

```

class FunctionBase
{
public:
    /* Polymorphic classes need virtual destructors. */
    virtual ~FunctionBase() {}

    /* Calls the stored function. */
    virtual Ret execute(const Arg& val) = 0;
    virtual FunctionBase* clone() const = 0;
};

/* Template derived class that executes a specific type of function. */
template<typename UnaryFunction> class FunctionImpl : public FunctionBase
{
public:
    explicit FunctionImpl(UnaryFunction f) : fn(f) {}
    virtual Ret execute(const Arg& val)
    {
        return fn(val);
    }
    virtual FunctionImpl* clone() const
    {
        return new FunctionImpl(*this);
    }
    UnaryFunction fn;
};

```

CS106L

Standard C++ Programming Laboratory

**Course Reader
Fall 2009**

Keith Schwarz
Stanford University

Acknowledgements

This course reader represents the culmination of two years' work on CS106L and its course handouts. Neither the class nor this reader would have been possible without Julie Zelenski's support and generosity during CS106L's infancy. I strongly encourage you to take one of Julie's classes – you will not be disappointed.

I'd also like to extend thanks to all of the CS106L students I've had the pleasure of teaching over the years. It is truly a joy to watch students light up when they see exactly what C++ can do. The long hours that went into this course reader would not have been possible without the knowledge that students are genuinely interested in the material.

Additionally, I would like to thank the brave souls who were generous enough to proofread draft versions of this course reader. Yin Huang and Steven Wu offered particularly apt advice on style and grammar. Ilya Sherman gave wonderful suggestions on typesetting and layout and caught many errors that slipped under my radar. Kyle Knutson helped double-check the correctness of the code in the extended examples. David Goldblatt helped me stay on top of recent developments in C++0x and pointed out how to make many of the STL practice problems truer to the spirit of the library. Sam Schreiber provided excellent advice about the overall structure of the reader and was the inspiration for the “Critiquing Class Design” chapter. Leonid Shamis astutely suggested that I expand the section on development environments. Brittney Fraser's amazing feedback made many of the examples easier to understand and prevented several major errors from making it into this reader.

This is the first edition of this course reader. While much of the content is taken directly from previous quarters' course handouts, over half of this course reader is entirely new. There are certainly ^{layout} problems, typos, grammatically errors, and spelling mistakes that have made it into this version. If you have any comments, corrections, or suggestions, please send me an email at htiek@cs.stanford.edu.

This course reader and its contents, except for quotations from other sources, are all © 2009 Keith Schwarz. If you would like to copy this course reader or its contents, send me an email and I'd be glad to see how I can help out.

Table of Contents

Introduction.....	1
Chapter 0: What is C++?.....	5
Chapter 1: Getting Started.....	9
Chapter 2: C++ without genlib.h	17
Introduction to the C++ Standard Library.....	21
Chapter 3: Streams.....	25
Chapter 4: STL Containers, Part I.....	41
Chapter 5: Extended Example: Snake.....	53
Chapter 6: STL Iterators.....	81
Chapter 7: STL Containers, Part II.....	91
Chapter 8: Extended Example: Finite Automata.....	99
Chapter 9: STL Algorithms.....	113
Chapter 10: Extended Example: Palindromes.....	127
C++ Core Language Features.....	133
Chapter 11: Pointers and References.....	135
Chapter 12: C Strings.....	147
Chapter 13: The Preprocessor.....	157
Chapter 14: Introduction to Templates.....	183
Chapter 15: const	205
Chapter 16: Extended Example: UnionFind	225
Chapter 17: Member Initializer Lists.....	237
Chapter 18: static	245
Chapter 19: Conversion Constructors.....	255
Chapter 20: Copy Constructors and Assignment Operators.....	261
Chapter 21: Extended Example: Critiquing Class Design.....	279
Chapter 22: Operator Overloading.....	291
Chapter 23: Extended Example: SmartPointer	315
Chapter 24: Extended Example: DimensionType	331
Chapter 25: Extended Example: grid	345
Chapter 26: Functors.....	365
Chapter 27: Introduction to Exception Handling.....	391
Chapter 28: Extended Example: Gauss-Jordan Elimination.....	405
Chapter 29: Introduction to Inheritance.....	429
Chapter 30: Extended Example: Function	463
More to Explore.....	479
Chapter 31: C++0x.....	481
Chapter 32: Where to Go From Here.....	497
Appendices.....	501
Appendix 0: Moving from C to C++.....	503
Appendix 1: Solutions to Practice Problems.....	515
Bibliography.....	537
Index.....	539

Part Zero

Introduction

Suppose we want to write a function that computes the average of a list of numbers. One implementation is given here:

```
double GetAverage(double arr[], int numElems)
{
    double total = 0.0;
    for(int h = 0; h < numElems; ++h)
        total += arr[h] / numElems;

    return total;
}
```

An alternative implementation is as follows:

```
template <typename ForwardIterator>
double GetAverage(ForwardIterator begin, ForwardIterator end)
{
    return accumulate(begin, end, 0.0) / distance(begin, end);
}
```

Don't panic if you don't understand any of this code – you're not expected to at this point – but even without an understanding of how either of these functions work it's clear that they are implemented differently. Although both of these functions are valid C++ and accurately compute the average, experienced C++ programmers will likely prefer the second version to the first because it is safer, more concise, and more versatile. To understand why you would prefer the second version of this function requires a solid understanding of the C++ programming language. Not only must you have a firm grasp of how all the language features involved in each solution work, but you must also understand the benefits and weaknesses of each of the approaches and ultimately which is a more versatile solution.

The purpose of this course is to get you up to speed on C++'s language features and libraries to the point where you are capable of not only writing C++ code, but also critiquing your design decisions and arguing why the cocktail of language features you chose is appropriate for your specific application. This is an ambitious goal, but if you take the time to read through this reader and work out some of the practice problems you should be in excellent C++ shape.

Who this Course is For

This course is designed to augment CS106B/X by providing a working knowledge of C++ and its applications. C++ is an industrial-strength tool that can be harnessed to solve a wide array of problems, and by the time you've completed CS106B/X and CS106L you should be equipped with the skill set necessary to identify solutions to complex problems, then to precisely and efficiently implement those solutions in C++.

This course reader assumes a knowledge of C++ at the level at which it would be covered in the first two weeks of CS106B/X. In particular, I assume that you are familiar with the following:

0. How to print to the console (i.e. `cout` and `endl`)
1. Primitive variable types (`int`, `double`, etc.)
2. The `string` type.
3. `enums` and `structs`.
4. Functions and function prototypes.
5. Pass-by-value and pass-by-reference.
6. Control structures (`if`, `for`, `while`, `do`, `switch`).
7. CS106B/X-specific libraries (`genlib.h`, `simpio.h`, the ADTs, etc.)

If you are unfamiliar with any of these terms, I recommend reading the first chapter of *Programming Abstractions in C++* by Eric Roberts and Julie Zelenski, which has an excellent treatment of the material. These concepts are fundamental to C++ but aren't that particular to the language – you'll find similar constructs in C, Java, Python, and other languages – and so I won't discuss them at great length. In addition to the language prerequisites, you should have at least one quarter of programming experience under your belt (CS106A should be more than enough). We'll be writing a lot of code, and the more programming savvy you bring to this course, the more you'll take out of it.

How this Reader is Organized

The course reader is logically divided into four sections:

0. **Introduction:** This section motivates and introduces the material and covers information necessary to be a working C++ programmer. In particular, it focuses on the history of C++, how to set up a C++ project for compilation, and how to move away from the `genlib.h` training wheels we've provided you in CS106B/X.
1. **The C++ Standard Library:** C++ has a standard library chock-full of programming goodies. Before moving on to more advanced language features, we'll explore what the streams library and STL have to offer.
2. **C++ Core Language:** C++ is an enormous language that affords great flexibility and control over exactly how your programs execute. This section discusses what tools are at your disposal and provides guidelines for their proper usage.
3. **More to Explore:** Unfortunately, this course reader cannot cover the entire C++ programming language. This section helps set up your journey into C++ with a discussion of the future of C++ and relevant C++ resources.

Notice that this course reader focuses on C++'s standard libraries before embarking on a detailed tour of its language features. This may seem backwards – after all, how can you understand libraries written in a language you have not yet studied? – but from experience I believe this is the best way to learn C++. A comprehensive understanding of the streams library and STL requires a rich understanding of templates, inheritance, functors, and operator overloading, but even without knowledge of these techniques it's still possible to write nontrivial C++ programs that use these libraries. For example, after a quick tour of the streams library and basic STL containers, we'll see how to write an implementation of the game Snake with an AI-controlled player. Later, once we've explored the proper language features, we'll revisit the standard libraries and see how they're put together.

To give you a feel for how C++ looks in practice, this course reader contains ten extended examples that demonstrate how to harness the concepts of the previous chapters to solve a particular problem. I *strongly* suggest that you take the time to read over these examples and play around with the code. The extended examples showcase how to use the techniques developed in previous chapters, and by seeing how the different pieces of C++ work together you will be a much more capable coder. In addition, I've tried to conclude each chapter with a few practice problems. Take a stab at them – you'll get a much more nuanced view of the language if you do. Solutions to some of my favorite problems are given in Appendix One. Exercises with solutions are marked with a diamond (♦).

C++ is a large language and it is impossible to cover all of its features in a single course. To help guide further exploration into C++ techniques, most chapters contain a “More to Explore” section listing important topics and techniques that may prove useful in your future C++ career.

Supplemental Reading

This course reader is by no means a complete C++ reference and there are many libraries and language features that we simply do not have time to cover. However, the portions of C++ we do cover are among the most-commonly used and you should be able to pick up the remaining pieces on a need-to-know basis. If you are interested in a more complete reference text, Bjarne Stroustrup's *The C++ Programming Language, Third Edition* is an excellent choice. Be aware that *TC++PL* is *not* a tutorial – it's a reference – and so you will probably want to read the relevant sections from this course reader before diving into it. If you're interested in a hybrid reference/tutorial, I would recommend *C++ Primer, Fourth Edition* by Lippman, Lajoie, and Moo. As for on-line resources, the C++ FAQ Lite at www.parashift.com/c++-faq-lite/ has a great discussion of C++'s core language features. cplusplus.com has perhaps the best coverage of the C++ standard library on the Internet, though its discussion of the language as a whole is fairly limited.

Onward and Forward!

Chapter 0: What is C++?

Every programming language has its own distinct flavor influenced by its history and design. Before seriously studying a programming language, it's important to learn *why* the language exists and what its objectives are. This chapter covers a quick history of C++, along with some of its design principles.

An Abbreviated History of C++*

The story of C++ begins with Bjarne Stroustrup, a Danish computer scientist working toward his PhD at Cambridge University. For his research, Stroustrup developed a simulator which modeled computers communicating over a network. Stroustrup chose to work in a language called Simula, at the time one of the foremost object-oriented programming languages. As Stroustrup recalled, at first Simula seemed like the perfect tool for the job:

It was a pleasure to write that simulator. The features of Simula were almost ideal for the purpose, and I was particularly impressed by the way the concepts of the language helped me think about the problems in my application. The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language...

I had used Simula before... but was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. [Str94]

In Simula, it was possible to model a computer using a computer *object* and a network using a network *object*, and the way that physical computers sent packets over physical networks corresponded to the way computer objects sent and received messages from network objects. But while Simula made it easier for Stroustrup to develop the simulator, the resulting program was so slow that it failed to produce any meaningful results. This was not the fault of Stroustrup's implementation, but of Simula itself. Simula was bloated and language features Stroustrup didn't use in his program were crippling the simulator's efficiency. For example, Stroustrup found that eighty percent of his program time was being spent on garbage collection despite the fact that the simulation didn't create any garbage. [Str94] In other words, while Simula had *decreased* the time required to build the simulator, it dramatically *increased* the time required for the simulator to execute.

Stroustrup realized that his Simula-based simulator was going nowhere. To continue his research, Stroustrup scrapped his Simula implementation and rewrote the program in a language he knew ran quickly and efficiently: BCPL. BCPL has since gone the way of the dodo, but at the time was a widely used, low-level systems programming language. Stroustrup later recalled that writing the simulator in BCPL was “horrible.” [Str94] As a low-level language, BCPL lacked objects and to represent computers and networks Stroustrup had to manually lay out and manipulate the proper bits and bytes. However, BCPL programs were far more efficient than their Simula counterparts, and Stroustrup's updated simulator worked marvelously.

Stroustrup's experiences with the distributed systems simulator impressed upon him the need for a more suitable tool for constructing large software systems. Stroustrup sought a hybridization of the best features of Simula and BCPL – a language with both high-level constructs and low-level runtime efficiency. After receiving his PhD, Stroustrup accepted a position at Bell Laboratories and began to create such a language. Settling on C as a base language, Stroustrup incorporated high-level constructs in the style of Simula while still maintaining C's underlying efficiency.

* This section is based on information from *The Design and Evolution of C++* by Bjarne Stroustrup.

After several revisions, *C with Classes*, as the language was originally dubbed, accumulated other high-level features and was officially renamed C++. C++ was an overnight success and spread rapidly into the programming community; for many years the number of C++ programmers was doubling every seven months. In 2007, C++ achieved three million users worldwide. [Str09] What began as Stroustrup's project at Bell Laboratories became an ISO-standardized programming language found in a variety of applications.

C++ Today

C++ began as a hybrid of high- and low-level languages but has since evolved into a distinctive language with its own idioms and constructs. Many programmers treat C++ as little more than an object-oriented C, but this view obscures much of the magic of C++. C++ is a *multiparadigm* programming language, meaning that it supports several different programming styles. C++ supports *imperative* programming in the style of C, meaning that you can treat C++ as an upgraded C. C++ supports *object-oriented* programming, so you can construct elaborate class hierarchies that hide complexity behind simple interfaces. C++ supports *generic* programming, allowing you to write code reusable in a large number of contexts. Finally, C++ supports a limited form of *higher-order* programming, allowing you to write functions that construct and manipulate other functions at runtime.

Design Philosophy

C++ is a comparatively old language; its first release was in 1985. Since then numerous other programming languages have sprung up – Java, Python, C#, and Javascript, to name a few. How exactly has C++ survived so long when others have failed? C++ may be useful and versatile, but so were BCPL and Simula, neither of which are in widespread use today.

One of the main reasons that C++ is still in use (and evolving) today has been its core guiding principles. Stroustrup has maintained an active interest in C++ since its inception and has steadfastly adhered to a particular design philosophy. Here is a sampling of the design points, as articulated in Stroustrup's *The Design and Evolution of C++*.

- *C++'s evolution must be driven by real problems.* When existing programming styles prove insufficient for modern challenges, C++ adapts. For example, the introduction of exception handling provided a much-needed system for error recovery, and abstract classes allowed programmers to define interfaces more naturally.
- *Don't try to force people.* C++ supports multiple programming styles. You can write code similar to that found in pure C, design class hierarchies as you would in Java, or develop software somewhere in between the two. C++ respects and trusts you as a programmer, allowing you to write the style of code you find most suitable to the task at hand rather than rigidly locking you into a single pattern.
- *Always provide a transition path.* C++ is designed such that the programming principles and techniques developed at any point in its history are still applicable. With few exceptions, C++ code written ten or twenty years ago should still compile and run on modern C++ compilers. Moreover, C++ is designed to be mostly backwards-compatible with C, meaning that veteran C coders can quickly get up to speed with C++.

The Goal of C++

There is one quote from Stroustrup ([Str94]) I believe best sums up C++:

*C++ makes programming **more enjoyable** for **serious programmers**.*

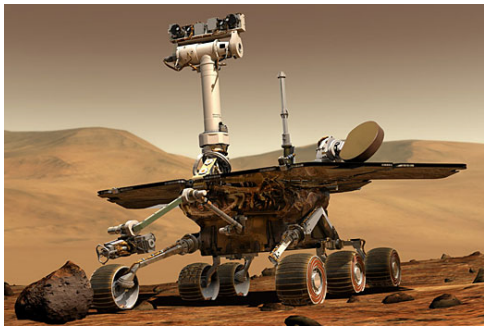
What exactly does this mean? Let's begin with what constitutes a *serious programmer*. Rigidly defining “serious programmer” is difficult, so instead I'll list some of the programs and projects written in C++ and leave it as an exercise to the reader to infer a proper definition. For example, you'll find C++ in:



Mozilla Firefox. The core infrastructure underlying all Mozilla projects is written predominantly in C++. While much of the code for Firefox is written in Javascript and XUL, these languages are executed by interpreters written in C++.

The WebKit layout engine used by Safari and Google Chrome is also written in C++. Although it's closed-source, I suspect that Internet Explorer is also written in C++. If you're browsing the web, you're seeing C++ in action.

Java HotSpot. The widespread success of Java is in part due to *HotSpot*, Sun's implementation of the Java Virtual Machine. HotSpot supports just-in-time compilation and optimization and is a beautifully engineered piece of software. It's also written in C++. The next time that someone engages you in a debate about the relative merits of C++ and Java, you can mention that if not for a well-architected C++ program Java would not be a competitive language.



NASA / JPL. The rovers currently exploring the surface of Mars have their autonomous driving systems written in C++. *C++ is on Mars!*

*C++ makes programming **more enjoyable** for serious programmers.* Not only does C++ power all of the above applications, it powers them *in style*. You can program with high-level constructs yet enjoy the runtime efficiency of a low-level language like C. You can choose the programming style that's right for you and work in a language that trusts and respects your expertise. You can write code once that you will reuse time and time again. This is what C++ is all about, and the purpose of this book is to get you up to speed on the mechanics, style, and just plain excitement of C++.

With that said, let's dive into C++. Our journey begins!

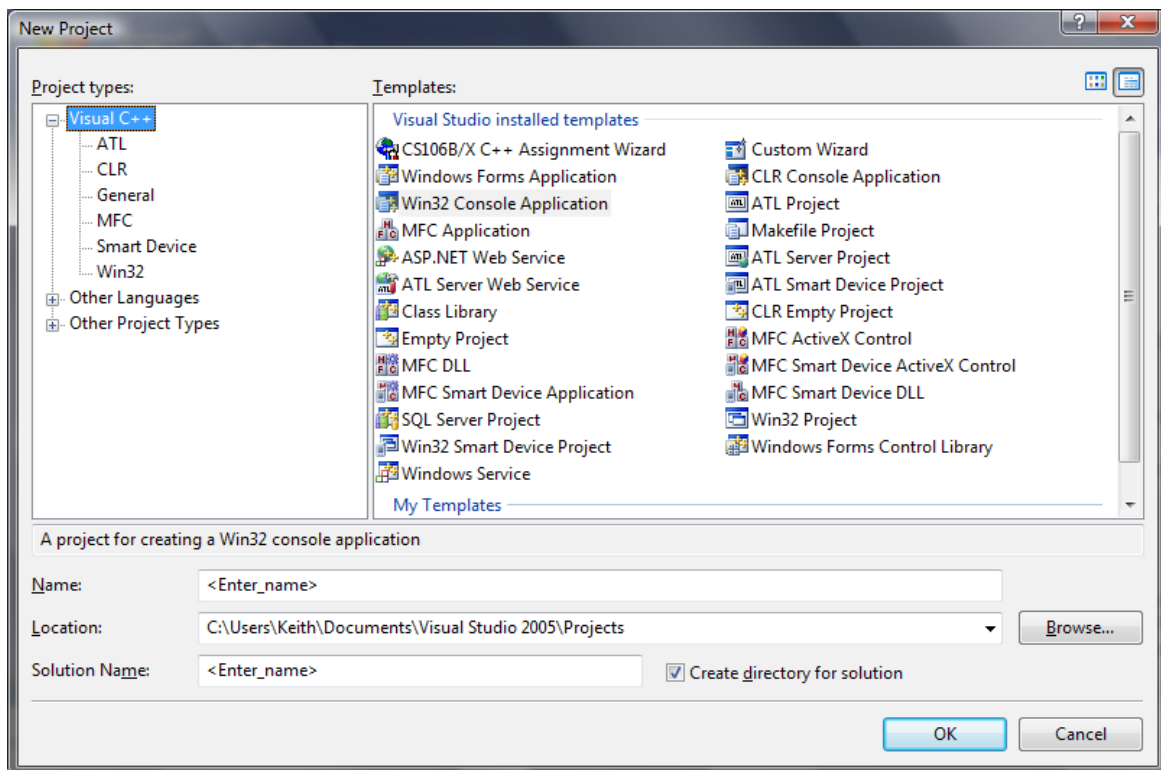
Chapter 1: Getting Started

Every journey begins with a single step, and in ours it's getting to the point where you can compile, link, run, and debug C++ programs. This depends on what operating system you have, so in this section we'll see how to get a C++ project up and running under Windows, Mac OS X, and Linux.

Compiling C++ Programs under Windows

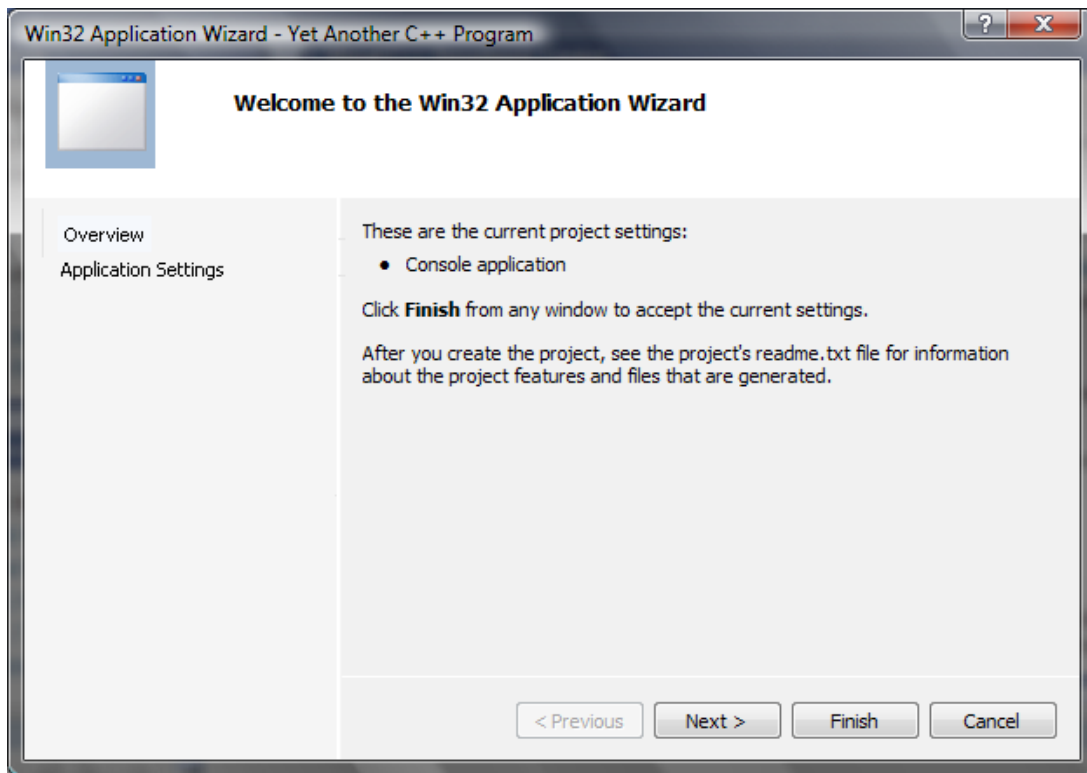
This section assumes that you are using Microsoft Visual Studio 2005 (VS2005). If you are a current CS106B/X student, you can follow the directions on the course website to obtain a copy. Otherwise, be prepared to shell out some cash to get your own copy, though it is definitely a worthwhile investment.* Alternatively, you can download Visual C++ 2008 Express Edition, a free version of Microsoft's development environment sporting a fully-functional C++ compiler. The express edition of Visual C++ lacks support for advanced Windows development, but is otherwise a perfectly fine C++ compiler. You can get Visual C++ 2008 Express Edition from <http://www.microsoft.com/express/vc/>. With only a few minor changes, the directions for using VS2005 should also apply to Visual C++ 2008 Express Edition, so this section will only cover VS2005.

VS2005 organizes C++ code into “projects,” collections of source and header files that will be built into a program. The first step in creating a C++ program is to get an empty C++ project up and running, then to populate it with the necessary files. To begin, open VS2005 and from the **File** menu choose **New > Project...** You should see a window that looks like this:



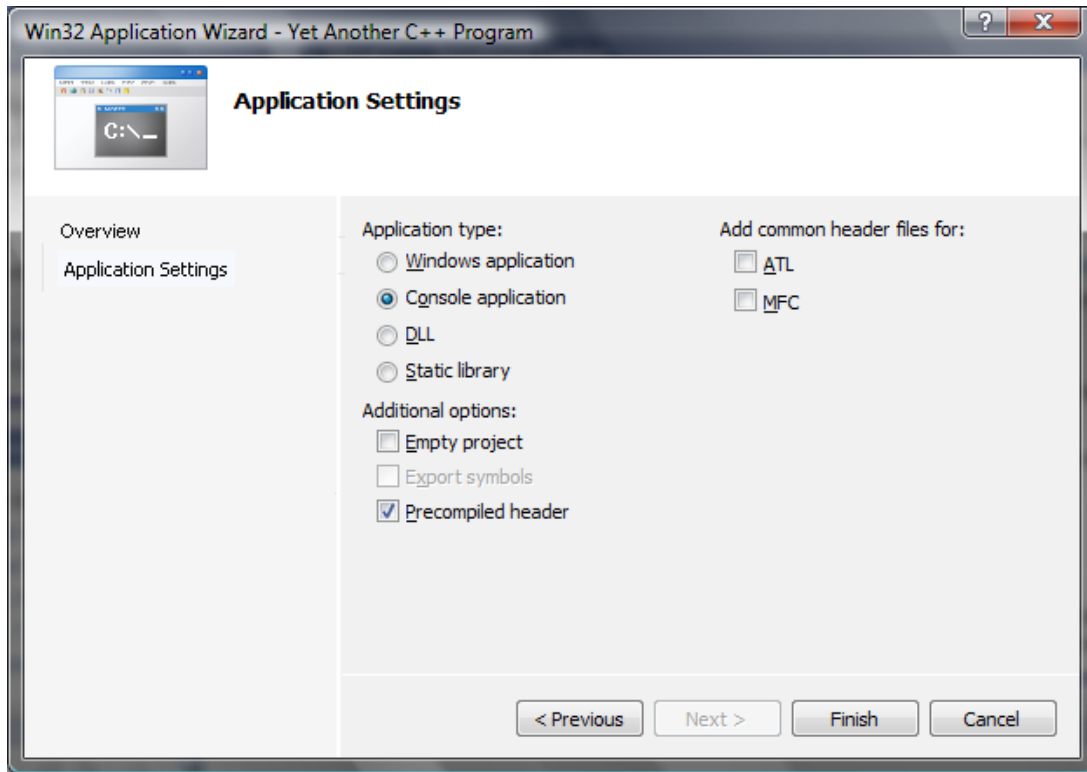
* I first began programming in C++ in 2001 using Microsoft Visual C++ 6.0, which cost roughly eighty dollars. I recently (2008) switched to Visual Studio 2005. This means that the compiler cost just over ten dollars a year. Considering the sheer number of hours I have spent programming, this was probably the best investment I have made.

As you can see, VS2005 has template support for all sorts of different projects, most of which are for Microsoft-specific applications such as dynamic-link libraries (DLLs) or ActiveX controls. We're not particularly interested in most of these choices – we just want a simple C++ program! To create one, find and choose **Win32 Console Application**. Give your project an appropriate name, then click **OK**. You should now see a window that looks like this, which will ask you to configure project settings:



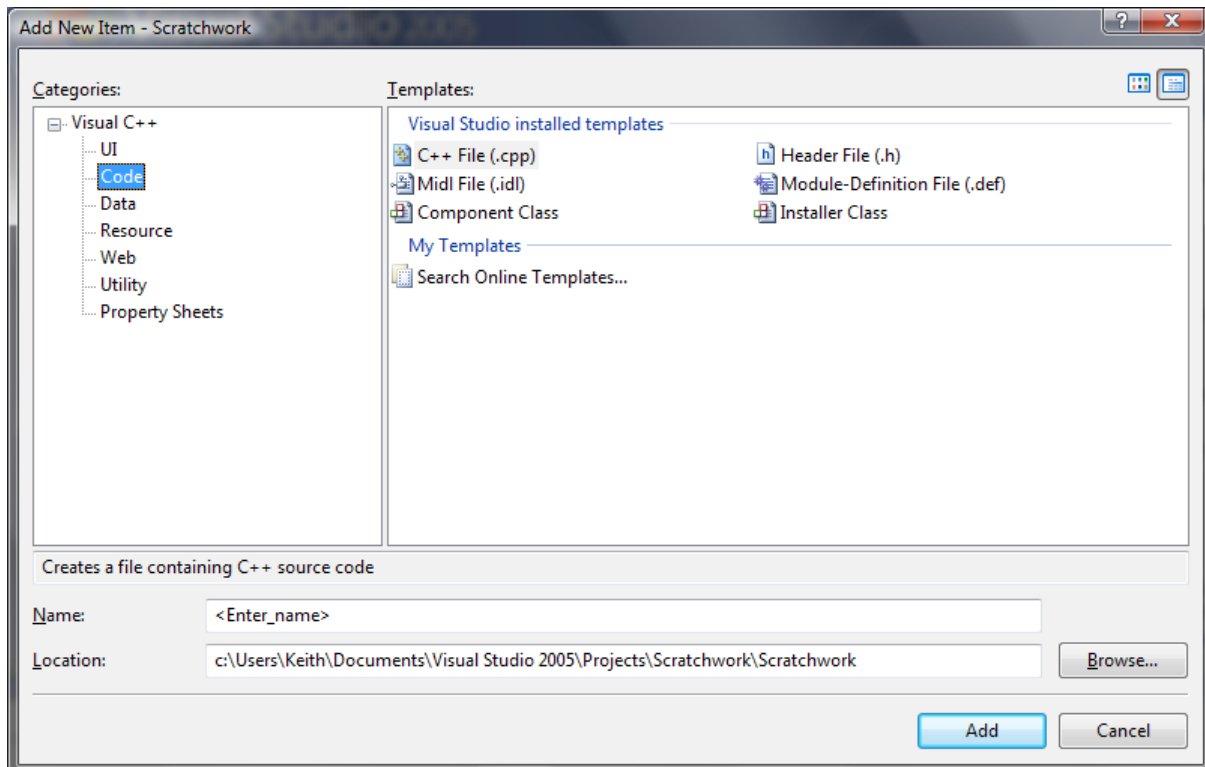
Note that the window title will have the name of the project you entered in the previous step in its title; “Yet Another C++ Program” is a placeholder.

At this point, you **do not** want to click **Finish**. Instead, hit **Next >** and you'll be presented with the following screen:



Keep all of the default settings listed here, but make sure that you check the box marked **Empty Project**. Otherwise VS2005 will give you a project with all sorts of Microsoft-specific features built into it. Once you've checked that box, click **Finish** and you'll have a fully functional (albeit empty) C++ project.

Now, it's time to create and add some source files to this project so that you can enter C++ code. To do this, go to **Project > Add New Item...** (or press CTRL+SHIFT+A). You'll be presented with the following dialog box:



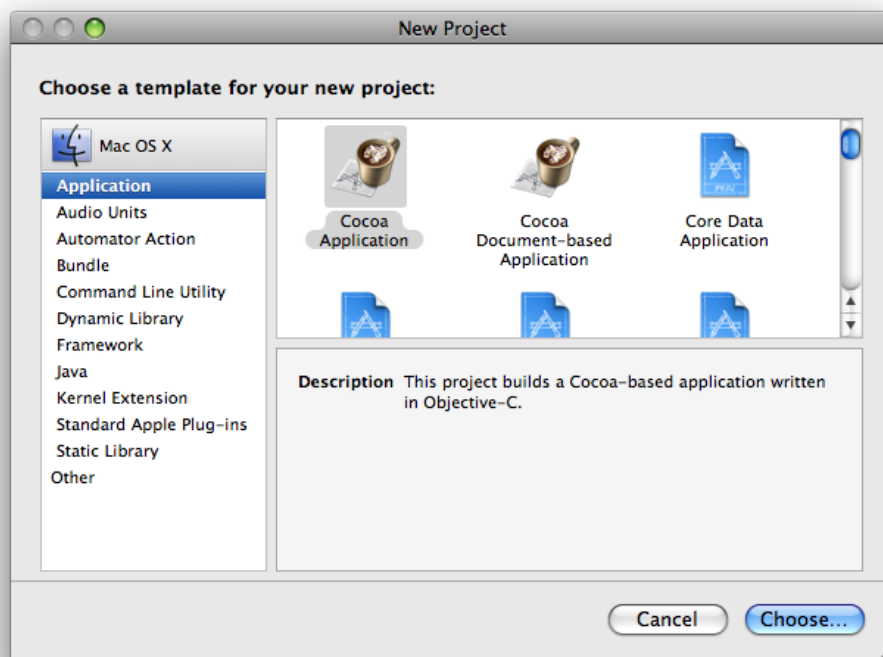
Choose **C++ File (.cpp)** and enter a name for it inside the Name field. VS2005 automatically appends .cpp to the end of the filename, so don't worry about manually entering the extension. Once you're ready, click **Add** and you should have your source file ready to go. Any C++ code you enter in here will be considered by the compiler and built into your final application.

Once you've written the source code, you can compile and run your programs by pressing **F5**, choosing **Debug > Start Debugging**, or clicking the green “play” icon. By default VS2005 will close the console window after your program finishes running, and if you want the window to persist after the program finishes executing you can run the program without debugging by pressing **CTRL+F5** or choosing **Debug > Start Without Debugging**. You should be all set to go!

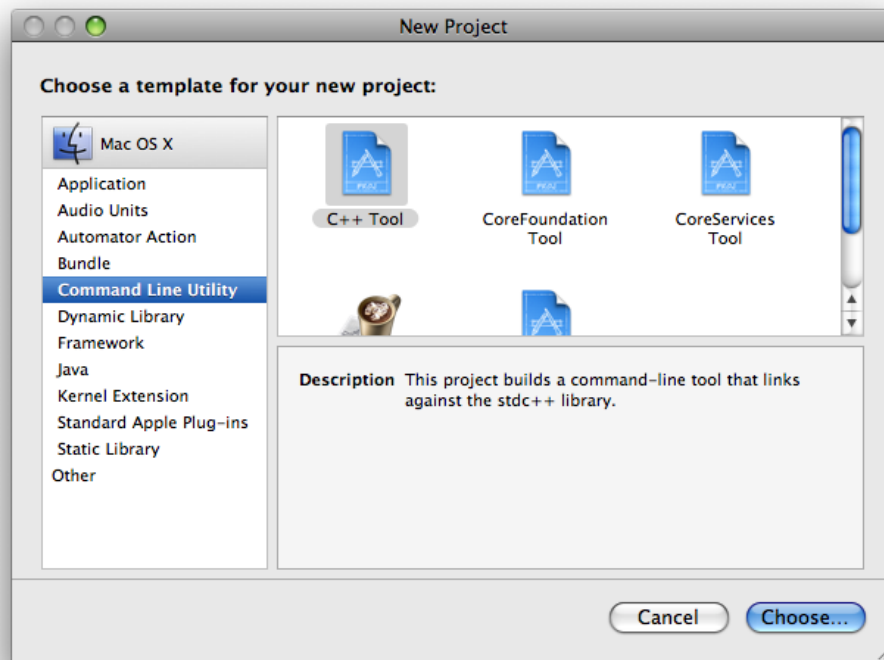
Compiling C++ Programs in Mac OS X

If you're developing C++ programs on Mac OS X, your best option is to use Apple's Xcode development environment. You can download Xcode free of charge from the Apple Developer Connection website at <http://developer.apple.com/>.

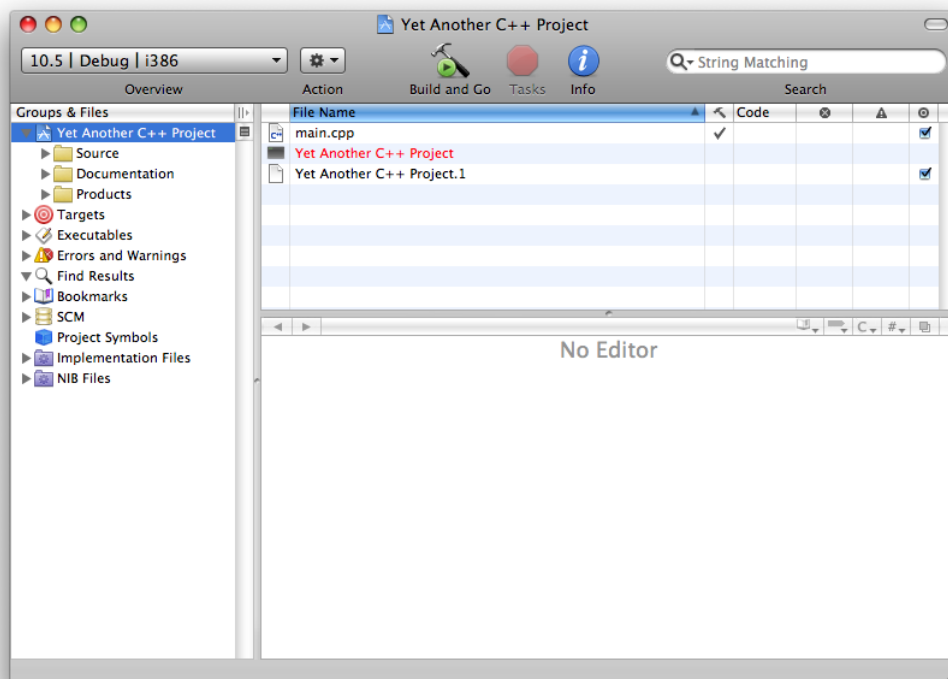
Once you've downloaded and installed Xcode, it's reasonably straightforward to create a new C++ project. Open Xcode. The first time that you run the program you'll get a nice welcome screen, which you're free to peruse but which you can safely dismiss. To create a C++ project, choose **File > New Project....** You'll be presented with a screen that looks like this:



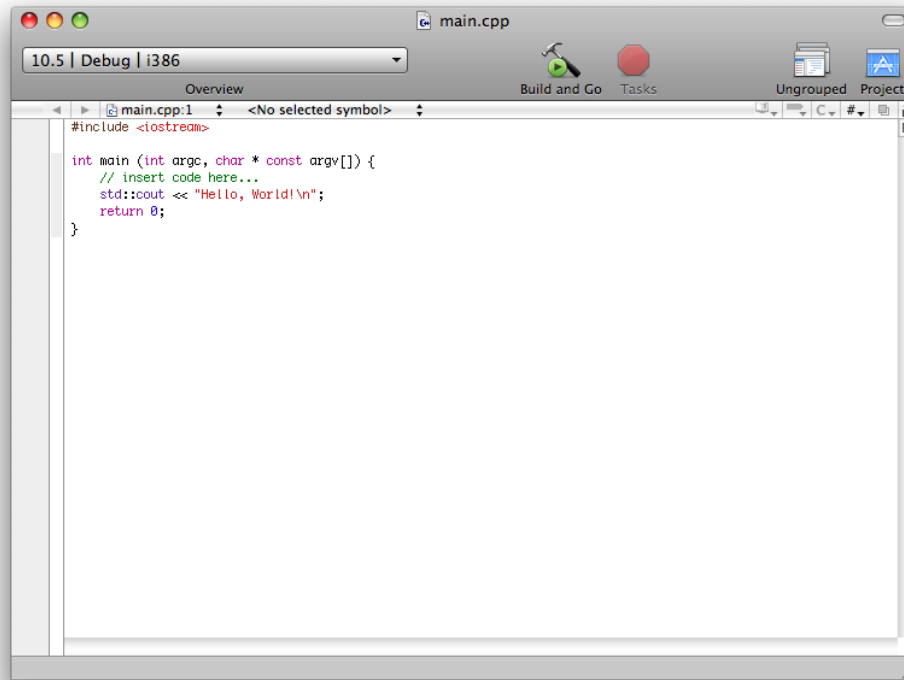
There are a lot of options here, most of which are Apple-specific or use languages other than C++ (such as Java or Objective-C). In the panel on the left side of the screen, choose **Command Line Utility** and you will see the following options:



Select **C++ Tool** and click the **Choose...** button. You'll be prompted for a project name and directory; feel free to choose whatever name and location you'd like. In this example I've used the name "Yet Another C++ Project," though I suggest you pick a more descriptive name. Once you've made your selection, you'll see the project window, which looks like this:

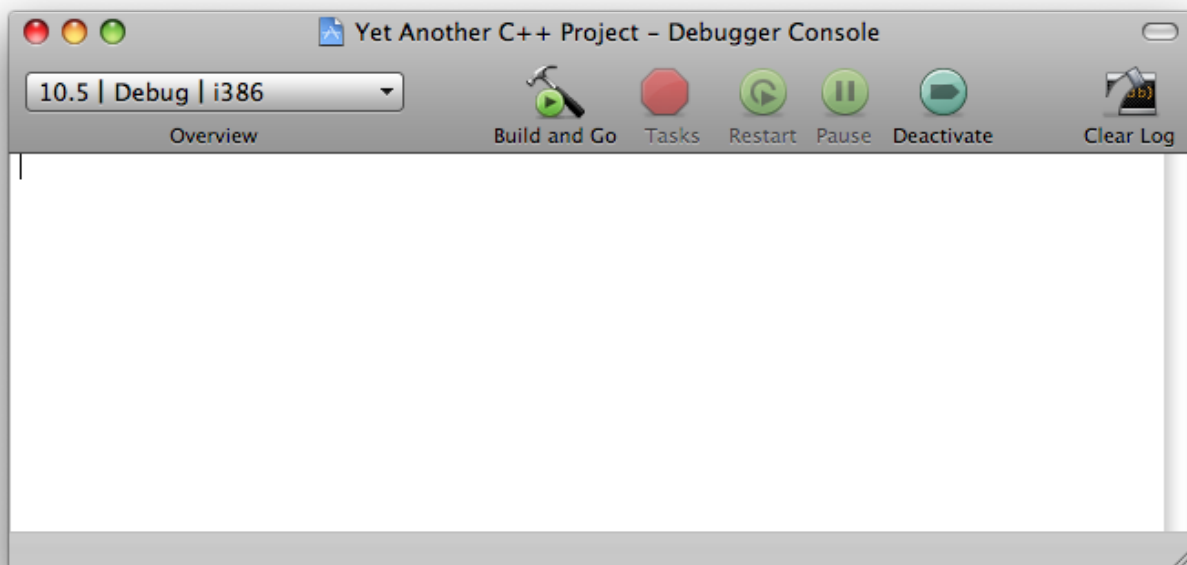


Notice that your project comes prepackaged with a file called `main.cpp`. This is a C++ source file that will be compiled and linked into the final program. By default, it contains a skeleton implementation of the Hello, World! program, as shown here:



Feel free to delete any of the code you see here and rewrite it as you see fit.

Because the program we've just created is a command-line utility, you will need to pull up the console window to see the output from your program. You can do this by choosing **Run > Console** or by pressing $\uparrow\uparrow\uparrow R$. Initially the console will be empty, as shown here:



Once you've run your program, the output will be displayed here in the console. You can run the program by clicking the **Build and Go** button (the hammer next to a green circle containing an arrow). That's it! You now have a working C++ project.

If you're interested in compiling programs from the Mac OS X terminal, you might find the following section on Linux development useful.

Compiling C++ Programs under Linux

For those of you using a Linux-based operating system, you're in luck – Linux is extremely developer-friendly and all of the tools you'll need are at your disposal from the command-line.

Unlike the Windows or Mac environments, when compiling code in Linux you won't need to set up a development environment using Visual Studio or Xcode. Instead, you'll just set up a directory where you'll put and edit your C++ files, then will directly invoke the GNU C++ Compiler (`g++`) from the command-line.

If you're using Linux I'll assume that you're already familiar with simple commands like `mkdir` and `chdir` and that you know how to edit and save a text document. When writing C++ source code, you'll probably want to save header files with the `.h` extension and C++ files with the `.cc`, `.cpp`, `.C`, or `.c++` extension. The `.cc` extension seems to be in vogue these days, though `.cpp` is also quite popular.

To compile your source code, you can execute `g++` from the command line by typing `g++` and then a list of the files you want to compile. For example, to compile `myfile.cc` and `myotherfile.cc`, you'd type

```
g++ myfile.cc myotherfile.cc
```

By default, this produces a file named `a.out`, which you can execute by entering `./a.out`. If you want to change the name of the program to something else, you can use `g++`'s `-o` switch, which produces an output file of a different name. For example, to create an executable called `myprogram` from the file `myfile.cc`, you could write

```
g++ myfile.cc -o myprogram
```

`g++` has a whole host of other switches (such as `-c` to compile but not link a file), so be sure to consult the `man` pages for more info.

It can get tedious writing out the commands to compile every single file in a project to form a finished executable, so most Linux developers use *makefiles*, scripts which allow you to compile an entire project by typing the `make` command. A full tour of makefiles is far beyond the scope of an introductory C++ text, but fortunately there are many good online tutorials on how to construct a makefile. The full manual for `make` is available online at <http://www.gnu.org/software/make/manual/make.html>.

Other Development Tools

If you are interested in using other development environments than the ones listed above, you're in luck. There are dozens of IDEs available that work on a wide range of platforms. Here's a small sampling:

- **NetBeans:** The NetBeans IDE supports C++ programming and is highly customizable. It also is completely cross-platform compatible, so you can use it on Windows, Mac OS X, and Linux.
- **MinGW:** MinGW is a port of common GNU tools to Microsoft Windows, so you can use tools like `g++` without running Linux. Many large software projects use MinGW as part of their build environment, so you might want to explore what it offers you.

- **Eclipse:** This popular Java IDE can be configured to run as a C++ compiler with a bit of additional effort. If you're using Windows you might need to install some additional software to get this IDE working, but otherwise it should be reasonably straightforward to configure.
- **Sun Studio:** If you're a Linux user and command-line hacking isn't your cup of tea, you might want to consider installing Sun Studio, Sun Microsystem's C++ development environment, which has a wonderful GUI and solid debugging support.

Chapter 2: C++ without `genlib.h`

When you arrived at your first CS106B/X lecture, you probably learned to write a simple “Hello, World” program like the one shown below:

```
#include "genlib.h"
#include <iostream>

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

Whether or not you have previous experience with C++, you probably realized that the first line means that the source code references an external file called `genlib.h`. For the purposes of CS106B/X, this is entirely acceptable (in fact, it's required!), but once you migrate from the educational setting to professional code you will run into trouble because `genlib.h` is *not* a standard header file; it's included in the CS106B/X libraries to simplify certain language features so you can focus on writing code, rather than appeasing the compiler.

In CS106L, none of our programs will use `genlib.h`, `simpio.h`, or any of the other CS106B/X library files. Don't worry, though, because none of the functions exported by these files are “magical.” In fact, in the next few chapters you will learn how to rewrite or supersede the functions and classes exported by the CS106B/X libraries.* If you have the time, I encourage you to actually open up the `genlib.h` file and peek around at its contents.

To write “Hello, World” without `genlib.h`, you'll need to add another line to your program. The “pure” C++ version of “Hello, World” thus looks something like this:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

We've replaced the header file `genlib.h` with the cryptic statement “`using namespace std;`” Before explaining exactly what this statement does, we need to take a quick diversion to lessons learned from development history. Suppose you're working at a company that produces two types of software: graphics design programs and online gunfighter duels (admittedly, this combination is pretty unlikely, but humor me for a while). Each project has its own source code files complete with a set of helper functions and classes. Here are some sample header files from each project, with most of the commenting removed:

* The exceptions are the graphics and sound libraries. C++ does not have natural language support for multimedia, and although many such libraries exist, we won't cover them in this text.

GraphicsUtility.h:

```
/* File: graphicsutility.h
 * Graphics utility functions.
 */

/* ClearScene: Clears the current scene. */
void ClearScene();

/* AddLine: Adds a line to the current scene. */
void AddLine(int x0, int y0, int x1, int y1);

/* Draw: Draws the current scene. */
void Draw();
```

GunfighterUtility.h:

```
/* File: gunfighterutility.h
 * Gunfighter utility functions.
 */

/* MarchTenPaces: Marches ten paces, animating each step. */
void MarchTenPaces(PlayerObject &toMove);

/* FaceFoe: Turns to face the opponent. */
void FaceFoe();

/* Draw: Unholsters and aims the pistol. */
void Draw();
```

Suppose the gunfighter team is implementing `MarchTenPaces` and needs to animate the gunfighters walking away from one another. Realizing that the graphics team has already implemented an entire library geared toward this, the gunfighter programmers import `graphicsutility.h` into their project, write code using the graphics functions, and try to compile. However, when they try to test their code, the linker reports errors to the effect of “error: function 'void Draw()' already defined.”

The problem is that the graphics and gunfighter modules each contain functions named `Draw()` with the same signature and the compiler can't distinguish between them. It's impractical for either team to rename their `Draw` function, both because the other programming teams expect them to provide functions named `Draw` and because their code is already filled with calls to `Draw`. Fortunately, there's an elegant resolution to this problem. Enter the C++ namespace keyword. A *namespace* adds another layer of naming onto your functions and variables. For example, if all of the gunfighter code was in the namespace “`Gunfighter`,” the function `Draw` would have the full name `Gunfighter::Draw`. Similarly, if the graphics programmers put their code inside namespace “`Graphics`,” they would reference the function `Draw` as `Graphics::Draw`. If this is the case, there is no longer any ambiguity between the two functions, and the gunfighter development team can compile their code.

But there's still one problem – other programming teams expect to find functions named `ClearScene` and `FaceFoe`, not `Graphics::ClearScene` and `Gunfighter::FaceFoe`. Fortunately, C++ allows what's known as a *using declaration* that lets you ignore fully qualified names from a namespace and instead use the shorter names.

Back to the Hello, World example, reprinted here:


```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

The statement “`using namespace std;`” following the `#include` directive tells the compiler that all of the functions and classes in the namespace `std` can be used without their fully-qualified names. This “`std`” namespace is the C++ *standard namespace* that includes all the library functions and classes of the standard library. For example, `cout` is truly named `std::cout`, and without the `using` declaration importing the `std` namespace, `Hello, World` would look something like this:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

While some programmers prefer to use the fully-qualified names when using standard library components, repeatedly writing `std::` can be a hassle. To eliminate this problem, in `genlib.h`, we included the `using` declaration for you. But now that we've taken the training wheels off and `genlib.h` is no more, you'll have to remember to include it yourself!

There's one more important part of `genlib.h`, the `string` type. Unlike other programming languages, C++ lacks a primitive string type.* Sure, there's the class `string`, but unlike `int` or `double` it's not a built-in type and must be included with a `#include` directive. Specifically, you'll need to write `#include <string>` at the top of any program that wants to use C++-style strings. And don't forget the `using` declaration, or you'll need to write `std::string` every time you want to use C++ strings!

* Technically speaking there are primitive strings in C++, but they aren't objects. See the chapter on C strings for more information.

Part One

Introduction to the C++ Standard Library

C++ has an enormous host of library functions, constants, and classes that simplify or obviate complex programming tasks. While much of the C++ standard library is beyond the scope of this class, a substantial portion of the library is accessible to beginning or intermediate-level C++ programmers. This chapter summarizes the library's contents and serves as a launching point for further exploration.

The C Runtime Library

C++ was originally designed as a superset of the C programming language – that is, with few exceptions, code that you write in pure C should compile and run in C++. Consequently, for backwards compatibility, C++ absorbed C's runtime library. You can identify libraries absorbed from C by the letter 'c' in the header file names. For example, to access the C library functions that let you access and manipulate the date and time, use the header file `<ctime>`, and for the core C standard library use `<cstdlib>`.

Much (but by no means all) of the C runtime library has been superseded by other portions of the C++ library. For example, C's input/output routines like `printf` and `scanf` can be replaced with C++'s safer and more robust `cout` and `cin` streams. However, there are many useful C runtime library functions (my personal favorite is `tmpnam`) and I recommend that you take some time to look over what's there.

Because some of the C runtime library is unsafe when mixed with standard C++ or requires an understanding of the language beyond the scope of this class, we will not cover the C runtime library in great depth. However, if you're interested in learning to use the C runtime library, there are some great resources online, such as

www.cppreference.com: A website covering both the C and C++ libraries. You might want to consider bookmarking this page as a quick reference because it's quite useful.

www.cplusplus.com/reference/clibrary/: A categorized overview of the C runtime library that includes code examples for most of the functions and macros. As with the above link, this is an extremely useful website and you might want to consider bookmarking it. For those of you with high-end cell phones, consider adding it to speed-dial.

The Streams Library

The streams library is C++'s way of reading and writing formatted input and output. The streams library includes functionality to read from and write to the console, files on disk, and even strings. In addition, it specifies a set of objects called *stream manipulators* that allows you to control the formatting and expression of data in a stream.

While our discussion of streams will cover a good deal of the library, we will not cover some topics such as binary file reading and random access, nor will we address some of the lower-level stream objects. For more information on these topics, you might want to refer to:

www.cplusplus.com/reference/iostream/: cplusplus.com has a very good overview of the streams library that includes a handy class library and even offers a peek into the inner workings of the classes.

The String Library

For those of you with a background in pure C, the C++ string library might seem like nothing short of a miracle. The C++ `string` is lightweight, fast, flexible, and powerful. In fact, it's so powerful and easy to use that it's one of the few standard C++ classes used in CS106B/X. For more information about `string`, refer to *Programming Abstractions in C++* and the handouts from CS106B/X.

The Standard Template Library (STL)

The STL is a collection of classes that store data (containers), objects to access data (iterators), functions that operate on data (algorithms), and objects that manipulate functions (functors). An understanding of the STL is critical to fully appreciate how powerful and versatile C++ is. However, as is bound to happen with any powerful programming library, the STL is complex and requires a strong understanding of the C++ language to fully use. Although we will dedicate several chapters to the STL, there simply isn't enough time to explore all of its facets and functionality.

If you're interested in exploring more topics in the STL, consider referring to these sources:

Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley. ISBN 0-201-74962-9. This is widely recognized as one of the most useful books concerning the STL. Rather than serving as an introductory text on the subject, Meyers' book describes how to maximize efficiency and get the most for your money out of the STL.

www.cplusplus.com/reference/: A great (and free!) online reference covering much of the STL, including code samples.

Numeric Libraries

The numeric libraries, mostly defined in the `<numeric>` and `<valarray>` headers, are classes and functions designed for computational or mathematical programming. For those of you with a background in Python, this header includes classes that let you access arrays via slices. We will not cover the numeric classes in this text, but those of you who are interested may want to consider looking into:

[msdn2.microsoft.com/en-us/library/fzkk3cy8\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/fzkk3cy8(VS.80).aspx): Microsoft's `valarray` reference, which is one of the better coverages I've found.

Yang, Daoqi. *C++ and Object-oriented Numeric Computing for Scientists and Engineers*. Springer. ISBN 0-387-98990-0. If you're interested in learning C++ in order to do computational programming, this is the book for you. It's a good introduction to C++ and includes many mathematical examples, such as solving ODEs. This book requires a mathematical background in linear algebra and calculus.

Memory Management Libraries

The C++ library also contains various objects and functions designed to help with memory allocation and deallocation. For example, the `auto_ptr` template class acts a “smart pointer” that automatically deallocates its memory, while the `set_new_handler` function can be used to set an emergency handler in case `operator new` can't find enough memory to satisfy a request.

While we will cover a subset of the memory management libraries in the second half of this course reader, a complete treatment of memory management is an advanced topic far beyond the scope of an introductory text. If you're interested in some practical applications of the memory libraries, consider reading:

www.gotw.ca/publications/using_auto_ptr_effectively.htm: The `auto_ptr` class can simplify your code and make it safer to use. However, it is not completely intuitive. This article is a great introduction to `auto_ptr` and offers several pointers and caveats.

Exception-Handling Libraries

C++ supports exception-handling with `try/throw/catch` blocks, as those of you familiar with Java might recognize. While we will cover exception handling in the second half of this course, you may still want to explore the libraries in more depth than what is covered here. If you're interested in exception-handling in general, these resources might be useful:

Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley. ISBN 0-201-61562-2. This book is an excellent resource on writing exception-safe code and is highly regarded in the C++ community. If you're interested in learning about writing code compatible with C++ exceptions, this is the book for you.

www.boost.org/more/generic_exception_safety.html: While this site is primarily aimed at writing exception-safe container classes, it nonetheless provides an excellent introduction to C++ exception safety.

Locale Libraries

Many programs are designed for an international audience where notational conventions vary (for example, \$1,234.56 in the United States might be written as 1.234,56 US elsewhere). The locale libraries offer a method for writing code that can easily be localized into other regions. Locale functions are far beyond the scope of an introductory text, but those of you who are interested may consider the following source useful:

www.cantrip.org/locale.html: This is one of the best introductions to locales that I've come across on the Internet. It uses advanced C++ syntax that might be confusing, so you might want to have a reference handy before proceeding.

Language Support Library

The C++ standard specifies the syntax and semantics of the C++ language, but it leaves many important decisions to individual implementers. For example, the size of an `int` or the behavior of a `double` holding too large a value vary on a system-by-system basis. To enable C++ programs to determine the configuration of their systems, C++ provides the language support library, a set of classes and constants that contain information about the particulars of the current C++ implementation.

In CS106B/X and CS106L, you will not need to worry about the language support library. However, if you plan on working on a cross-platform C++ project where these details matter, you may want to consider looking into:

http://www.unc.edu/depts/case/pgi/pgC++_lib/stdlibc/num_5679.htm: A complete reference for the `numeric_limits` class, which exports most of the information specific to a particular C++ implementation.

The Boost C++ Libraries

Although this course will only cover standard C++, if you plan to pursue C++ beyond this class you should strongly consider looking into the Boost C++ Libraries. Boost is the most prominent third-party C++ library, and several parts of Boost are being considered by the C++ Standards Committee for inclusion in the next release of C++. Once you've gotten the hang of the C++ standard library, you should strongly consider exploring Boost, especially since many parts of Boost seamlessly integrate with the existing libraries. You can find the Boost libraries at www.boost.org.

Third-Party Libraries

For those of you with experience in languages like Java or Python, the C++ standard library might seem limited. C++ lacks a graphics and sound package, has no support for networking, and does not natively support windowing. To use features like these, you'll need to rely on third-party libraries, like Microsoft's Win32 API or the X Window System.

There are several reasons C++ opted out of a “monolithic library” strategy. First, since the C++ libraries focus more on data manipulation than presentation, C++ works on more platforms than other languages; you can use C++ to program both web browsers and microcontrollers. Second, some features like multimedia and windowing vary greatly from system to system. Standardizing these features would result in a standard library catering to the lowest common denominator, something likely to please no one. Instead, C++ leaves libraries like these to third parties, so the resulting libraries are less portable but more powerful. Finally, libraries sometimes provide fundamentally different ways to approach programming in C++. Standardizing libraries like these would force C++ programmers comfortable with one programming style to uncomfortably adjust to another, and consequently libraries of this sort are left to third-parties.

Based on the sort of functionality you're looking for, the following third-party libraries might be worth exploring:

Networking: The Internet Sockets API is a widely-used set of types and functions used to write networking code in C and C++. The library is written in C, so you may want to build C++ wrappers around some of the core functionality. Socket programming works on most major operating systems, though the Windows version (Winsock) has a few syntactic differences from the rest of the sockets API. Alternatively, you can use the Boost library's `asio` package, which supports a wide array of networking protocols.

Graphics: There are literally hundreds of graphics packages written for C/C++, of which OpenGL (cross-platform) and DirectX (Microsoft-specific) are among the most well-known. These systems have a bit of a learning curve, but if you're interested you may want to take CS148 (Introduction to Computer Graphics).

Multithreading: There are numerous multithreading libraries available for C/C++. Microsoft's Win32 API has a rich set of threading capabilities, but is Windows-specific. Boost's threading classes are widely-used in industry and are cross-platform, but have a steep learning curve. For the truly brave among you, the pthreads library is a low-level C library that supports multithreading.

Windowing: All major operating systems provide some platform-specific interface that you can use to develop windowed software. If you're interested in a cross-platform windowing system, you might want to look into the GTK+ API. Microsoft's Win32 API is excellent if you want to learn to program Windows-specific applications.

Chapter 3: Streams

The streams library is C++'s way of formatting input and output to and from a variety of sources, including the console, files, and string buffers. However, like most parts of the standard library, the streams library has many features and idiosyncrasies that can take some time to adjust to. This chapter introduces the streams library and includes useful tips and tricks for practical programming.

`cout` and `cin`

As you've seen in CS106B/X, C++ provides a stream object called `cout` (**character output**) you can use to write formatted data to the console. For example, to print out a message to the user, you can write code that looks like this:

```
cout << "I'm sorry Dave. I'm afraid I can't do that." << endl;
```

Here, the `<<` operator is called the *stream insertion operator* and instructs C++ to push data into a stream.

To build a truly interactive program, however, we'll need to get input from the user. In CS106B/X, we provide the `simpio.h` header file, which exports the input functions `GetLine`, `GetInteger`, `GetReal`, and `GetLong`. Though useful, these functions are not part of the C++ standard library and will not be available outside of CS106B/X. Don't worry, though, because by the end of this chapter we'll see how to implement them using only standard C++.

The streams library exports another stream object called `cin` (**character input**) which lets you read values directly from the user. To read a value from `cin`, you use the *stream extraction operator* `>>`. Syntactically, the stream extraction operator mirrors the stream insertion operator. For example, here's a code snippet to prompt the user for an integer.

```
cout << "Please enter an integer: ";
int myInteger;
cin >> myInteger; // Value stored in myInteger
```

You can also read multiple values from `cin` by chaining together the stream extraction operator in the same way that you can write multiple values to `cout` by chaining the stream insertion operator:

```
int myInteger;
string myString;
cin >> myInteger >> myString; // Read an integer and string from cin
```

Note that when using `cin`, you should not read into `endl` the way that you write `endl` when using `cout`. Hence the following code is illegal:

```
int myInteger;
cin >> myInteger >> endl; // Error: Cannot read into endl.
```

In practice, it is not a good idea to read values directly from `cin`. Unlike `GetInteger` and the like, `cin` does not perform any safety checking of user input and if the user does not enter valid data `cin` will malfunction. We will cover how to fix these problems later in this chapter.

Reading and Writing Files

C++ provides a header file called `<fstream>` (**file stream**) that exports the `ifstream` and `ofstream` types, streams that perform file I/O. The naming convention is unfortunate – `ifstream` stands for **input file stream** (not “something that might be a stream”) and `ofstream` for **output file stream**. There is also a generic `fstream` class which can do both input and output, but we will not cover it in this chapter.

To create an `ifstream` that reads from a file, you can use this syntax:

```
ifstream myStream("myFile.txt");
```

This creates a new stream object named `myStream` which reads from the file `myFile.txt`, provided of course that the file exists. We can then read data from `myStream` just as we would from `cin`, as shown here:

```
ifstream myStream("myFile.txt");
int myInteger;
myStream >> myInteger; // Read an integer from myFile.txt
```

Notice that the final line looks almost identical to code that reads an integer from the console.

You can also open a file by using the `ifstream`'s `open` member function, as shown here:

```
ifstream myStream;           // Note: did not specify the file
myStream.open("myFile.txt"); // Now reading from myFile.txt
```

When opening a file using an `ifstream`, there is a chance that the specified file can't be opened. The filename might not specify an actual file, you might not have permission to read the file, or perhaps the file is locked. If you try reading data from an `ifstream` that is not associated with an open file, the read will fail and you will not get back meaningful data. After trying to open a file, you can check if the operation succeeded by using the `.is_open()` member function. For example, here's code to open a file and report an error to the user if a problem occurred:

```
ifstream input("myfile.txt");
if(!input.is_open())
    cerr << "Couldn't open the file myfile.txt" << endl;
```

Notice that we report the error to the `cerr` stream. `cerr`, like `cout`, is an output stream, but unlike `cout`, `cerr` is designed for error reporting and is sometimes handled differently by the operating system.

The output counterpart to `ifstream` is `ofstream`. As with `ifstream`, you specify which file to write to either by using the `.open()` member function or by specifying the file when you create the `ofstream`, as shown below:

```
ofstream myStream("myFile.txt"); // Write to myFile.txt
```

A word of warning: if you try writing to a nonexistent file with an `ofstream`, the `ofstream` will create the file for you. However, if you open a file that already exists, the `ofstream` will overwrite all of the contents of the file. Be careful not to write to important files without first backing them up!

The streams library is one of the older libraries in C++ and the `open` functions on the `ifstream` and `ofstream` classes predate the `string` type. If you have the name of a file stored in a C++ `string`, you will need to convert the `string` into a C-style string (covered in the second half of this book) before passing it as a parameter to `open`. This can be done using the `.c_str()` member function of the `string` class, as shown here:


```
ifstream input(myString.c_str()); // Open the filename stored in myString
```

When a file stream object goes out of scope, C++ will automatically close the file for you so that other processes can read and write the file. If you want to close the file prematurely, you can use the `.close()` member function. After calling `close`, reading or writing to or from the file stream will fail.

As mentioned above in the section on `cin`, when reading from or writing to files you will need to do extensive error checking to ensure that the operations succeed. Again, we'll see how to do this later.

Stream Manipulators

Consider the following code that prints data to `cout`:

```
cout << "This is a string!" << endl;
```

What exactly is `endl`? It's an example of a *stream manipulator*, an object that can be inserted into a stream to change some sort of stream property. `endl` is one of the most common stream manipulators, though others exist as well. To motivate some of the more complex manipulators, let's suppose that we have a file called `table-data.txt` containing four lines of text, where each line consists of an integer value and a real number. For example:

table-data.txt

137	2.71828
42	3.14159
7897987	1.608
1337	.01101010001

We want to write a program which reads in this data and prints it out in a table, as shown here:

-----	+	-----	+	-----
1		137		2.71828
2		42		3.14159
3		7897987		1.608
4		1337		0.01101

Here, the first column is the one-indexed line number, the second the integer values from the file, and the third the real-numbered values from the file.

Let's begin by defining a few constants to control what the output should look like. Since there are four lines in the file, we can write

```
const int NUM_LINES = 4;
```

And since there are three columns,

```
const int NUM_COLUMNS = 3;
```

Next, we'll pick an arbitrary width for each column. We'll choose twenty characters, though in principle we could pick any value as long as the data fit:

```
const int COLUMN_WIDTH = 20;
```

Now, we need to read in the table data and print out the formatted table. We'll decompose this problem into two smaller steps, resulting in the following source code:

```

#include <iostream>
#include <fstream>
using namespace std;

const int NUM_LINES = 4;
const int NUM_COLUMNS = 3;
const int COLUMN_WIDTH = 20;

int main()
{
    PrintTableHeader();
    PrintTableBody();
    return 0;
}

```

`PrintTableHeader` is responsible for printing out the top part of the table (the row of dashes and pluses) and `PrintTableBody` will load the contents of the file and print them to the console.

Despite the fact that `PrintTableHeader` precedes `PrintTableBody` in this program, we'll begin by implementing `PrintTableBody` as it illustrates exactly how much firepower we can get from the stream manipulators. We know that we need to open the file `table-data.txt` and that we'll need to read four lines of data from it, so we can begin writing this function as follows:

```

void PrintTableBody()
{
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k)
    {
        /* ... process data ... */
    }
}

```

You may have noticed that at the end of this `for` loop I've written `++k` instead of `k++`. There's a slight difference between the two syntaxes, but in this context they are interchangeable. When we talk about operator overloading in a later chapter we'll talk about why it's generally considered better practice to use the prefix increment operator instead of the postfix.

Now, we need to read data from the file and print it as a table. We can start by actually reading the values from the file, as shown here:

```

void PrintTableBody()
{
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
       * program.
       */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k)
    {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;
    }
}

```

Next, we need to print out the table row. This is where things get tricky. If you'll recall, the table is supposed to be printed as three columns, each a fixed width, that contain the relevant data. How can we ensure that when we print the values to `cout` that we put in the appropriate amount of whitespace? Manually writing space characters would be difficult, so instead we'll use a stream manipulator called `setw` (**set width**) to force `cout` to pad its output with the right number of spaces. `setw` is defined in the `<iomanip>` header file and can be used as follows:

```
cout << setw(10) << 137 << endl;
```

This tells `cout` that the next item it prints out should be padded with spaces so that it takes up at least ten characters. Similarly,

```
cout << setw(20) << "Hello there!" << endl;
```

Would print out `Hello there!` with sufficient leading whitespace.

By default `setw` pads the next operation with spaces on the left side. You can customize this behavior with the `left` and `right` stream manipulators, as shown here:

```

cout << '[' << left << setw(10) << "Hello!" << ']' << endl; // [      Hello!]
cout << '[' << right << setw(10) << "Hello!" << ']' << endl; // [Hello!    ]

```

Back to our example. We want to ensure that every table column is exactly `COLUMN_WIDTH` spaces across. Using `setw`, this is relatively straightforward and can be done as follows:

```

void PrintTableBody()
{
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
       * program.
       */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k)
    {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}

```

This produces the following output when run on the input file described above:

1	137	2.71828
2	42	3.14159
3	7897987	1.608
4	1337	0.01101

The body of the table looks great, and now we just need to print the table header, which looks like this:

```
-----+-----+-----
```

If you'll notice, this is formed by printing twenty dashes, then the pattern `--+`, another twenty dashes, the pattern `--+`, and finally another twenty dashes. We could thus implement `PrintTableHeader` like this:

```

void PrintTableHeader()
{
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column)
    {
        for(int k = 0; k < COLUMN_WIDTH; ++k)
            cout << '-';
        cout << "--+-";
    }

    /* Now print the ---...--- pattern for the last column. */
    for(int k = 0; k < COLUMN_WIDTH; ++k)
        cout << '-';

    /* Print a newline... there's nothing else on this line. */
    cout << endl;
}

```

As written there's nothing wrong with this code and the program will work just fine, but we can simplify the implementation by harnessing stream manipulators. Notice that at two points we need to print out `COLUMN_WIDTH` copies of the dash character. When printing out the table body, we were able to use the `setw` stream manipulator to print multiple copies of the space character; is there some way that we can use it here to print out multiple dashes? The answer is yes, thanks to `setfill`. The `setfill` manipulator accepts a parameter indicating what

character to use as a fill character for `setw`, then changes the stream such that all future calls to `setw` pad the stream with the specified character. For example:

```
cout << setfill('0') << setw(8) << 1000 << endl; // Prints 00001000
cout << setw(8) << 1000 << endl; // Prints 00001000 because of last setfill
```

Note that `setfill` does not replace all space characters with instances of some other character. It is only meaningful in conjunction with `setw`. For example:

```
cout << setfill('X') << "Some Spaces" << endl; // Prints Some Spaces
```

Using `setfill` and `setw`, we can print out `COLUMN_WIDTH` copies of the dash character as follows:

```
cout << setfill('-') << setw(COLUMN_WIDTH) << "" << setfill(' ');
```

This code is dense, so let's walk through it one step at a time. The first part, `setfill('-')`, tells `cout` to pad all output with dashes instead of spaces. Next, we use `setw` to tell `cout` that the next operation should take up at least `COLUMN_WIDTH` characters. The trick is the next step, printing the empty string. Since the empty string has length zero and the next operation will always print out at least `COLUMN_WIDTH` characters padded with dashes, this code prints out `COLUMN_WIDTH` dashes in a row. Finally, since `setfill` permanently sets the fill character, we use `setfill(' ')` to undo the changes we made to `cout`.

Using this code, we can rewrite `PrintTableHeader` as follows:

```
void PrintTableHeader()
{
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column)
        cout << setfill('-') << setw(COLUMN_WIDTH) << "" << "-+-";

    /* Now print the ---...--- pattern for the last column and a newline. */
    cout << setw(COLUMN_WIDTH) << "" << setfill(' ') << endl;
}
```

Notice that we only call `setfill(' ')` once, at the end of this function, since there's no reason to clear it at each step. Also notice that we've reduced the length of this function dramatically by having the library take care of the heavy lifting for us. The code to print out a table header is now three lines long!

There are many stream manipulators available in C++. The following table lists some of the more commonly-used ones:

<code>boolalpha</code>	<pre>cout << true << endl; // Output: 1 cout << boolalpha << true << endl; // Output: true</pre> Determines whether or not the stream should output boolean values as 1 and 0 or as “true” and “false.” The opposite manipulator is <code>noboolalpha</code>, which reverses this behavior.
<code>setw(n)</code>	<pre>cout << 10 << endl; // Output: 10 cout << setw(5) << 10 << endl; // Output: 10</pre> Sets the minimum width of the output for the next stream operation. If the data doesn't meet the minimum field requirement, it is padded with the default fill character until it is the proper size.

Common stream manipulators, contd.

hex, dec, oct	<pre>cout << 10 << endl; // Output: 10 cout << dec << 10 << endl; // Output: 10 cout << oct << 10 << endl; // Output: 12 cout << hex << 10 << endl; // Output: a cin >> hex >> x; // Reads a hexadecimal value.</pre> Sets the radix on the stream to either octal (base 8), decimal (base 10), or hexadecimal (base 16). This can be used either to format output or change the base for input.
ws	<pre>myStream >> ws >> value;</pre> Skips any whitespace stored in the stream. By default the stream extraction operator skips over whitespace, but other functions like <code>getline</code> do not. <code>ws</code> can sometimes be useful in conjunction with these other functions.

When Streams Go Bad

Because stream operations often involve transforming data from one form into another, stream operations are not always guaranteed to succeed. For example, consider the following code snippet, which reads integer values from a file:

```
ifstream in("input.txt"); // Read from input.txt
for(int i = 0; i < NUM_INTS; ++i)
{
    int value;
    in >> value;
    /* ... process value here ... */
}
```

If the file `input.txt` contains `NUM_INTS` consecutive integer values, then this code will work correctly. However, what happens if the file contains some other type of data, such as a string or a real number?

If you try to read stream data of one type into a variable of another type, rather than crashing the program or filling the variable with garbage data, the stream fails by entering an *error state* and the value of the variable will not change. Once the stream is in this error state, any subsequent read or write operations will automatically and silently fail, which can be a serious problem.

You can check if a stream is in an error state with the `.fail()` member function. Don't let the name mislead you – `fail` checks if a stream is in an error state, rather than putting the stream into that state. For example, here's code to read input from `cin` and check if an error occurred:

```
int myInteger;
cin >> myInteger;
if(cin.fail()) { /* ... error ... */ }
```

If a stream is in a fail state, you'll probably want to perform some special handling, possibly by reporting the error. Once you've fixed any problems, you need to tell the stream that everything is okay by using the `.clear()` member function to bring the stream out of its error state. Note that `clear` won't skip over the input that put the stream into an error state; you will need to extract this input manually.

Streams can also go into error states if a read operation fails because no data is available. This occurs most commonly when reading data from a file. Let's return to the table-printing example. In the `PrintTableData` function, we hardcoded the assumption that the file contains exactly four lines of data. But what if we want to print

out tables of arbitrary length? In that case, we'd need to continuously read through the file extracting and printing numbers until we exhaust its contents. We can tell when we've run out of data by checking the `.fail()` member function after performing a read. If `.fail()` returns true, something prevented us from extracting data (either because the file was malformed or because there was no more data) and we can stop looping.

Recall that the original code for reading data looks like this:

```
void PrintTableBody()
{
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k)
    {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}
```

The updated version of this code, which reads all of the contents of the file, is shown here:

```
void PrintTableBody()
{
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    int rowNumber = 0;
    while(true)
    {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        if(input.fail()) break;

        cout << setw(COLUMN_WIDTH) << (rowNumber + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;

        rowNumber++;
    }
}
```

Notice that we put the main logic into a `while(true)` loop that breaks when `input.fail()` returns true instead of a `while(!input.fail())` loop. These two structures may at first appear similar, but are quite different from one another. In a `while(!input.fail())` loop, we only check to see if the stream encountered an error after reading and processing the data in the body of the loop. This means that the loop will execute once more than it should, because we don't notice that the stream malfunctioned until the top of the loop. On the other hand, in the above loop structure (`while(true)` plus `break`), we stop looping as soon as the stream realizes that something has gone awry. Confusing these two loop structures is a common error, so be sure that you understand why to use the “loop-and-a-half” idiom rather than a simple `while` loop.

When Streams Do Too Much

Consider the following code snippet, which prompts a user for an age and hourly salary:

```
int age;
double hourlyWage;
cout << "Please enter your age: ";
cin >> age;
cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
```

As mentioned above, if the user enters a string or otherwise non-integer value when prompted for their age, the stream will enter an error state. There is another edge case to consider. Suppose the input is 2.71828. You would expect that, since this isn't an integer (it's a real number), the stream would go into an error state. However, this isn't what happens. The first call, `cin >> age`, will set `age` to 2. The next call, `cin >> hourlyWage`, rather than prompting the user for a value, will find the .71828 from the earlier input and fill in `hourlyWage` with that information. Despite the fact that the input was malformed for the first prompt, the stream was able to partially interpret it and no error was signaled.

As if this wasn't bad enough, suppose we have this program instead, which prompts a user for an administrator password and then asks whether the user wants to format her hard drive:

```
string password;
cout << "Enter administrator password: ";
cin >> password;
if(password == "password") // Use a better password, by the way!
{
    cout << "Do you want to erase your hard drive (Y or N)? ";
    char yesOrNo;
    cin >> yesOrNo;
    if(yesOrNo == 'y')
        EraseHardDrive();
}
```

What happens if someone enters password `y`? The first call, `cin >> password`, will read only `password`. Once we reach the second `cin` read, it automatically fills in `yesOrNo` with the leftover `y`, and there goes our hard drive! Clearly this is not what we intended.

As you can see, reading directly from `cin` is unsafe and poses more problems than it solves. In CS106B/X we provide you with the `simpio.h` library primarily so you don't have to deal with these sorts of errors. In the next section, we'll explore an entirely different way of reading input that avoids the above problems.

An Alternative: `getline`

Up to this point, we have been reading data using the stream extraction operator, which, as you've seen, can be dangerous. However, there are other functions that read data from a stream. One of these functions is `getline`, which reads characters from a stream until a newline character is encountered, then stores the read characters (minus the newline) in a `string`. `getline` accepts two parameters, a stream to read from and a `string` to write to. For example, to read a line of text from the console, you could use this code:

```
string myStr;
getline(cin, myStr);
```

No matter how many words or tokens the user types on this line, because `getline` reads until it encounters a newline, all of the data will be absorbed and stored in `myStr`. Moreover, because any data the user types in can

be expressed as a string, unless your input stream encounters a read error, `getline` will not put the stream into a fail state. No longer do you need to worry about strange I/O edge cases!

You may have noticed that the `getline` function acts similarly to the CS106B/X `GetLine` function. This is no coincidence, and in fact the `GetLine` function from `simpio.h` is implemented as follows:*

```
string GetLine()
{
    string result;
    getline(cin, result);
    return result;
}
```

At this point, `getline` may seem like a silver-bullet solution to our input problems. However, `getline` has a small problem when mixed with the stream extraction operator. When the user presses return after entering text in response to a `cin` prompt, the newline character is stored in the `cin` internal buffer. Normally, whenever you try to extract data from a stream using the `>>` operator, the stream skips over newline and whitespace characters before reading meaningful data. This means that if you write code like this:

```
int first, second;
cin >> first;
cin >> second;
```

The newline stored in `cin` after the user enters a value for `first` is eaten by `cin` before `second` is read. However, if we replace the second call to `cin` with a call to `getline`, as shown here:

```
int dummyInt;
string dummyString;
cin >> dummyInt;
getline(cin, dummyString);
```

`getline` will return an empty string. Why? Unlike the stream extraction operator, `getline` does *not* skip over the whitespace still remaining in the `cin` stream. Consequently, as soon as `getline` is called, it will find the newline remaining from the previous `cin` statement, assume the user has pressed return, and return the empty string.

To fix this problem, your best option is to replace all normal stream extraction operations with calls to library functions like `GetInteger` and `GetLine` that accomplish the same thing. Fortunately, with the information in the next section, you'll be able to write `GetInteger` and almost any `Get_____` function you'd ever need to use. When we cover templates and operator overloading in later chapters, you'll see how to build a generic read function that can parse any sort of data from the user.

A String Buffer: `stringstream`

Before we discuss writing `GetInteger`, we'll need to take a diversion to another type of C++ stream.

Often you will need to construct a string composed both of plain text and numeric or other data. For example, suppose you wanted to call this hypothetical function:

```
void MessageBoxAlert(string message);
```

* Technically, the implementation of `GetLine` from `simpio.h` is slightly different, as it checks to make sure that `cin` is not in an error state before reading.

and have it display a message box to the user informing her that the level number she wanted to warp to is out of bounds. At first thought, you might try something like

```
int levelNum = /* ... */;
MessageBoxAlert("Level " + levelNum + " is out of bounds."); // ERROR
```

For those of you with Java experience this might seem natural, but in C++ this isn't legal because you can't add numbers to strings (and when you can, it's almost certainly won't do what you expected; see the chapter on C strings).

One solution to this problem is to use another kind of stream object known as a *stringstream*, exported by the `<sstream>` header. Like console streams and file streams, *stringstreams* are stream objects and consequently all of the stream operations we've covered above work on *stringstreams*. However, instead of reading or writing data to an external source, *stringstreams* store data in temporary string buffers. In other words, you can view a *stringstream* as a way to create and read string data using stream operations.

For example, here is a code snippet to create a *stringstream* and put text data into it:

```
stringstream myStream;
myStream << "Hello!" << 137;
```

Once you've put data into a *stringstream*, you can retrieve the string you've created using the `.str()` member function. Continuing the above example, we can print out an error message as follows:

```
int levelNum = /* ... */;
stringstream messageText;
messageText << "Level " << levelNum << " is out of bounds.";
MessageBoxAlert(messageText.str());
```

stringstreams are an example of an *iostream*, a stream that can perform both input and output. You can both insert data into a *stringstream* to convert the data to a string and extract data from a *stringstream* to convert string data into a different format. For example:

```
stringstream myConverter;
int myInt;
string myString;
double myDouble;
myConverter << "137 Hello 2.71828"; // Insert string data
myConverter >> myInt >> myString >> myDouble; // Extract mixed data
```

The standard rules governing stream extraction operators still apply to *stringstreams*, so if you try to read data from a *stringstream* in one format that doesn't match the character data, the stream will fail. We'll exploit this functionality in the next section.

Putting it all together: Writing `GetInteger`

Using the techniques we covered in the previous sections, we can implement a set of robust user input functions along the lines of those provided by `simpio.h`. In this section we'll explore how to write `GetInteger`, which prompts the user to enter an integer and returns only after the user enters valid input.

Recall from the above sections that reading an integer from `cin` can result in two types of problems. First, the user could enter something that is not an integer, causing `cin` to fail. Second, the user could enter too much input, such as `137 246` or `Hello 37`, in which case the operation succeeds but leaves extra data in `cin` that can garble future reads. We can immediately eliminate these sorts of problems by using the `getline` function to

read input, since `getline` cannot put `cin` into a fail state and grabs all of the user's data, rather than just the first token.

The main problem with `getline` is that the input is returned as a `string`, rather than as formatted data. Fortunately, using a `stringstream`, we can convert this text data into another format of our choice. This suggests an implementation of `GetInteger`. We read data from the console using `getline` and funnel it into a `stringstream`. We then use standard stream manipulations to extract the integer from the `stringstream`, reporting an error and reprompting if unable to do so. We can start writing `GetInteger` as follows:

```
int GetInteger()
{
    while(true) // Read input until user enters valid data
    {
        stringstream converter;
        converter << GetLine();

        /* Process data here. On error: */
        cout << "Retry: "
    }
}
```

At this point, we've read in all of the data we need, and simply need to check that the data is in the proper format. As mentioned above, there are two sorts of problems we might run into – either the data isn't an integer, or the data contains leftover information that isn't part of the integer. We need to check for both cases. Checking for the first turns out to be pretty simple – because `stringstreams` are stream objects, we can see if the data isn't an integer by extracting an integer from our `stringstream` and checking if this puts the stream into a fail state. If so, we know the data is invalid and can alert the user to this effect.

The updated code for `GetInteger` is as follows:

```
int GetInteger()
{
    while(true) // Read input until user enters valid data
    {
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        converter >> result;
        if(!converter.fail())
        {
            /* ... check that there isn't any leftover data ... */
        }
        else cout << "Please enter an integer." << endl;
        cout << "Retry: "
    }
}
```

Finally, we need to check if there's any extra data left over. If so, we need to report to the user that something is wrong with the input, and can otherwise return the value we read. While there are several ways to check this, one simple method is to read in a single `char` from the `stringstream`. If it is possible to do so, then we know that there must have been something in the input stream that wasn't picked up when we extracted an integer and consequently that the input is bad. Otherwise, the stream must be out of data and will enter a fail state, signaling that the user's input was valid. The final code for `GetInteger`, which uses this trick, is shown here:

```

int GetInteger()
{
    while(true) // Read input until user enters valid data
    {
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        converter >> result;
        if(!converter.fail())
        {
            char remaining;
            converter >> remaining; // Check for stray input
            if(converter.fail()) // Couldn't read any more, so input is valid
                return result;
            else cout << "Unexpected character: " << remaining << endl;
        }
        else cout << "Please enter an integer." << endl;
        cout << "Retry: "
    }
}

```

More To Explore

C++ streams are extremely powerful and encompass a huge amount of functionality. While there are many more facets to explore, I highly recommend exploring some of these topics:

- **Random Access:** Most of the time, when performing I/O, you will access the data sequentially; that is, you will read in one piece of data, then the next, etc. However, in some cases you might know in advance that you want to look up only a certain piece of data in a file without considering all of the data before it. For example, a ZIP archive containing a directory structure most likely stores each compressed file at a different offset from the start of the file. Thus, if you wanted to write a program capable of extracting a single file from the archive, you'd almost certainly need the ability to jump to arbitrary locations in a file. C++ streams support this functionality with the `seekg`, `tellg`, `seekp`, and `tellp` functions (the first two for `istreams`, the latter for `ostreams`). Random access lets you quickly jump to single records in large data blocks and can be useful in data file design.
- **read and write:** When you write numeric data to a stream, you're actually converting them into sequences of characters that represent those numbers. For example, when you print out the four-byte value 78979871, you're using eight bytes to represent the data on screen or in a file – one for each character. These extra bytes can quickly add up, and it's actually possible to have on-disk representations of data that are more than twice as large as the data stored in memory. To get around this, C++ streams let you directly write data from memory onto disk without any formatting. All `ostreams` support a `write` function that writes unformatted data to a stream, and `istreams` support `read` to read unformatted data from a stream into memory. When used well, these functions can cut file loading times and reduce disk space usage. For example, The CS106B/X `Lexicon` class uses `read` to quickly load its data file into memory.

Practice Problems

Here are some questions to help you play around with the material from this chapter. Exercises with a diamond character have answers in Appendix One.

1. Write a function `ExtractFirstToken` that accepts a string and returns the first token from that string. For example, if you passed in the string “Eleanor Roosevelt,” the function should return “Eleanor.” For our purposes, define a token as a single continuous block of characters with no intervening whitespace. While it's possible to write this using the C library function `isspace` and a `for` loop, there's a much shorter solution leveraging off of a `stringstream`.
2. In common usage, numbers are written in *decimal* or *base 10*. This means that a string of digits is interpreted as a sum of multiples of powers of ten. For example, the number 137 is $1 \cdot 100 + 3 \cdot 10 + 7 \cdot 1$, which is the same as $1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$. However, it is possible to write numbers in other bases as well. For example, *octal*, or base 8, encodes numbers as sums of multiples of powers of eight. For example, 137 in octal would be $1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$ in decimal.* Similarly, *binary*, or base 2, uses powers of two.

When working in a particular base, we only use digits from 0 up to that base. Thus in base 10 we use the digits zero through nine, while in base five the only digits would be 0, 1, 2, 3, and 4. This means that 57 is not a valid base-five number and 93 is not a valid octal number. When working in bases numbered higher than ten, it is customary to use letters from the beginning of the alphabet as digits. For example, in *hexadecimal*, or base 16, one counts 0, 1, 2, ..., 9, A, B, C, D, E, F, 10. This means that 3D45E is a valid hexadecimal number, as is DEADBEEF or DEFACED.

Write a function `HasHexLetters` that accepts an `int` and returns whether or not that integer's hexadecimal representation contains letters. (*Hint: you'll need to use the `hex` and `dec` stream manipulators in conjunction with a `stringstream`. Try to solve this problem without brute-forcing it: leverage off the streams library instead of using loops.*) ♦

3. Modify the code for `GetInteger` to create a function `GetReal` that reads a real number from the user. How much did you need to modify to make this code work?
4. Using the code for `GetInteger` and the `boolalpha` stream manipulator, write a function `GetBoolean` that waits for the user to enter “true” or “false” and returns the corresponding boolean value.
5. Although the console does not naturally lend itself to graphics programming, it is possible to draw rudimentary approximations of polygons by printing out multiple copies of a character at the proper location. For example, we can draw a triangle by drawing a single character on one line, then three on the next, five on the line after that, etc. For example:

```

#
###
#####
#####
#####
#####

```

Using the `setw` and `setfill` stream manipulators, write a function `DrawTriangle` that takes in an `int` corresponding to the height of the triangle and a `char` representing a character to print, then draws a triangle of the specified height using that character. The triangle should be aligned so that the bottom row starts at the beginning of its line.

6. Write a function `OpenFile` that accepts as input an `ifstream` by reference and prompts the user for the name of a file. If the file can be found, `OpenFile` should return with the `ifstream` opened to read that file. Otherwise, `OpenFile` should print an error message and reprompt the user. (*Hint: If you try to open a nonexistent file with an `ifstream`, the stream goes into a fail state and you will need to use `.clear()` to restore it before trying again.*)

* Why do programmers always confuse Halloween and Christmas? Because 31 Oct = 25 Dec. ©

Chapter 4: STL Containers, Part I

In October of 1976 I observed that a certain algorithm – parallel reduction – was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures.

– Alex Stepanov, inventor of the STL. [Ste07]

The Standard Template Library (STL) is a programmer's dream. It offers efficient ways to store, access, manipulate, and view data and is designed for maximum extensibility. Once you've gotten over the initial syntax hurdles, you will quickly learn to appreciate the STL's sheer power and flexibility.

To give a sense of exactly where we're going, here are a few quick examples of code using the STL:

- We can create a list of random numbers, sort it, and print it to the console *in four lines of code!*

```
vector<int> myVector(NUM_INTS);
generate(myVector.begin(), myVector.end(), rand);
sort(myVector.begin(), myVector.end());
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"));
```

- We can open a file and print its contents *in two lines of code!*

```
ifstream input("my-file.txt");
copy(istreambuf_iterator<char>(input), istreambuf_iterator<char>(),
     ostreambuf_iterator<char>(cout));
```

- We can convert a string to upper case *in one line of code!*

```
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

If you aren't already impressed by the possibilities this library entails, keep reading. You will not be disappointed.

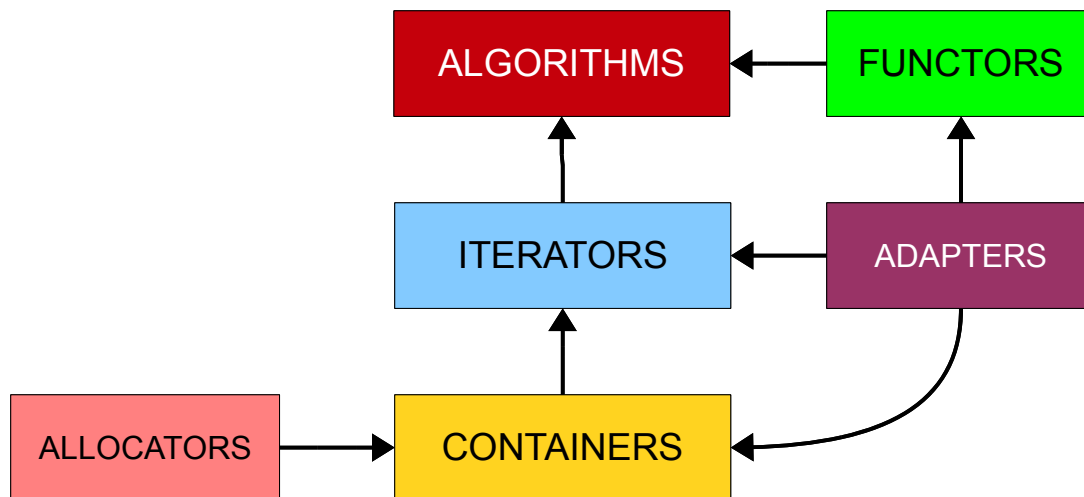
Overview of the STL

The STL is logically divided into six pieces, each consisting of generic components that interoperate with the rest of the library:

- **Containers.** At the heart of the STL are a collection of container classes, standard C++'s analog to the CS106B/X ADTs. For example, you can store an associative collection of key/value pairs in an STL `map`, or a growing list of elements in an STL `vector`.
- **Iterators.** Each STL container exports iterators, objects that view and modify ranges of stored data. Iterators have a common interface, allowing you to write code that operates on data stored in arbitrary containers.
- **Algorithms.** STL algorithms are functions that operate over ranges of data specified by iterators. The scope of the STL algorithms is staggering – there are algorithms for searching, sorting, reordering, permuting, creating, and destroying sets of data.

- **Adapters.** STL *adapters* are objects which transform an object from one form into another. For example, the `stack` adapter transforms a regular `vector` or `list` into a LIFO container, while the `istream_iterator` transforms a standard C++ stream into an STL iterator.
- **Functors.** Because so much of the STL relies on user-defined callback functions, the STL provides facilities for creating and modifying functions at runtime. We will defer our discussion of functors to the second half of this text, as they require operator overloading.
- **Allocators.** The STL allows clients of the container classes to customize how memory is allocated and deallocated, either for diagnostic or performance reasons. While allocators are an interesting topic worthy of discussion, they are beyond the scope of this course and we will not cover them.

Diagrammatically, these pieces are related as follows:



Here, the containers rely on the allocators for memory and produce iterators. Iterators can then be used in conjunction with the algorithms. Functors provide special functions for the algorithms, and adapters can produce functors, iterators, and containers. If this seems a bit confusing now, don't worry, you'll understand this relationship well by the time you've finished the next few chapters.

A Word on Safety

In a sense, the STL is like the CS106B/X ADT library – it provides a way for you to store and manipulate data. There are, of course, many differences between the libraries, of which the largest is safety. With the CS106B/X libraries, if you try to access the tenth element of a nine-element `Vector`, you'll get an error message alerting you to the fact and your program will terminate. With the STL, doing this results in *undefined behavior*. This means that anything could happen – your program might continue with garbage data, it might crash, or it might even cause yellow ducks to show up on the screen and start dancing. What's important to note, though, is that this means that the STL can easily introduce nasty bugs into your code. Most of the time, if you read one spot over the end in an STL `vector`, nothing serious will happen, but when something bad does come up you'll get crashes that can be tricky to track down. On the plus side, however, this lack of error checking means the STL container classes are much, *much* faster than anything you'll get from the CS106B/X libraries. In fact, the STL is so fast that it's reasonable to assume that if there's an STL and a non-STL way to accomplish a task, the STL way is going to be faster.

What does this all mean? Basically, you'll have to stop relying on user-friendly error messages to help debug your code. If you have bounds-checking issues with your STL containers, or try to pop an element off an empty `stack`, you'll probably be directed to a cryptic STL implementation file where the error occurred. It's the price you pay for efficiency.

A Word on Compiler Errors

The compiler errors you will encounter when making errors with the STL are nothing short of terrifying. Normally, if you leave a semicolon off a line, or try to assign a CS106B/X `Vector<string>` to a `Map<string>`, you'll get error messages that are somewhat comprehensible. Not so with the STL. In fact, STL errors are so difficult to read that it's going to take you a while before you'll have any idea what you're looking at. Here's a sample STL error message caused by passing a string into a function expecting an `int`:

```
c:\...\algorithm(897) : error C2446: '==' : no conversion from 'const char *' to
'int'
    There is no context in which this conversion is possible
    c:\...\algorithm(906) : see reference to function template instantiation
'void std::_Replace<std::_Vector_iterator<_Ty,_Alloc>,const
char[6]>(_FwdIt,_FwdIt,const char (&),const char (&))' being compiled
    with
    [
        _Ty=int,
        _Alloc=std::allocator<int>,
        _FwdIt=std::_Vector_iterator<int,std::allocator<int>>
    ]
    c:\...\main.cpp(11) : see reference to function template instantiation
'void std::replace<std::_Vector_iterator<_Ty,_Alloc>,const
char[6]>(_FwdIt,_FwdIt,const char (&),const char (&))' being compiled
    with
    [
        _Ty=int,
        _Alloc=std::allocator<int>,
        _FwdIt=std::_Vector_iterator<int,std::allocator<int>>
    ]
c:\...\algorithm(897) : error C2040: '==' : 'int' differs in levels of indirection
from 'const char [6]'
```

```
c:\...\algorithm(898) : error C2440: '=' : cannot convert from 'const char [6]' to
'int'
    There is no context in which this conversion is possible
```

When you get STL compiler errors, make sure to take the time to slowly read them and to compare the different types that they mention. If you have a type mismatch, it won't be immediately obvious, but it will be there. Similarly, if you try to call a member function that doesn't exist in a container class, be prepared to spend a few seconds reading what type the compiler reports before you even get to the member function that caused the error.

With that said, let's begin our trek into the heart of the STL! The remainder of this chapter focuses on *container classes*, objects like the CS106B/X ADTs that store data in a specialized format. We'll explore a sampling of the STL containers, how to harness their firepower, and what issues to be aware of when using them.

STL Containers vs CS106B/X ADTs

You're probably just getting used to the CS106B/X ADTs, so switching over to the STL container classes might be a bit jarring. Fortunately, for almost all of the CS106B/X ADTs there is a corresponding STL container class (the notable exception is `Grid`). The STL versions of the CS106B/X ADTs have the same name, except in lowercase: for example, the CS106 `Vector` is analogous to the STL `vector`. Furthermore, anything you can do with the CS106B/X ADTs you can also do with the STL container classes, although the syntax might be more cryptic. The main advantages of the STL might not be immediately apparent, but once you've finished the chapter on STL algorithms you'll see exactly how much the STL's capability dwarfs that of the CS106B/X ADT library.

To use the STL containers, you'll need to use a different set of headers than those you'll use for the CS106B/X ADTs. The header files for the STL use angle brackets (< and >) and don't end in .h. For example, the header for the STL `vector` is `<vector>`, instead of the CS106B/X `"vector.h"`.

stack

Perhaps the simplest STL container is the `stack`. The STL `stack` is similar to the CS106B/X `Stack` in most ways: it's a last-in, first-out (LIFO) container that supports `push` and `pop`; you can test to see if it's empty and, if not, how tall it is; and you can peek at the top element. However, there are a few subtle differences between the CS106B/X `Stack` and the STL `stack`, as you'll see in a minute.

The following table summarizes the member functions you can perform on a `stack`. We have not covered the `const` keyword yet, so for now feel free to ignore it.

CS106B/X <code>Stack</code>	STL <code>stack</code>	Differences
<code>int size()</code>	<code>size_type size() const</code>	Although the STL <code>stack</code> 's return type is <code>size_type</code> instead of <code>int</code> , these functions behave identically.
<code>bool isEmpty()</code>	<code>bool empty() const</code>	Don't get confused by the name: <code>empty</code> does <i>not</i> empty out the <code>stack</code> . The STL uses the function name <code>empty</code> instead of the CS106B/X <code>isEmpty</code> , but they do exactly the same thing.
<code>T& peek()</code>	<code>T& top()</code> <code>const T& top() const</code>	These functions do exactly the same thing, except that the STL <code>top</code> function does not perform any bounds-checking.
<code>void push(T toAdd)</code>	<code>void push(const T& toAdd)</code>	These functions are essentially identical.
<code>T pop();</code>	<code>void pop();</code>	See the next section.
<code>void clear();</code>	-	There is no STL <code>stack</code> equivalent of <code>clear</code> .

As you can see, most of the functions from the CS106B/X `Stack` work the same way in the STL `stack`, albeit with a few syntactic differences. The big difference, however, is the `pop` function. With the CS106B/X `Stack`, you can write code like this:

```
int topElem = myCS106Stack.pop();
```

This is *not* the case with the STL `stack`. The STL function `pop` removes the top element from the `stack` but *does not* return a value. Thus, an STL version of the above code might look something like this:

```
int topElem = mySTLStack.top();
mySTLStack.pop();
```

While this may make your code slightly less readable, it will improve your programs' runtime efficiency because the `top` function doesn't create a copy of the object on top of the `stack`. It's much faster to return a reference to an object than a copy of it, and if your `stack` stores non-primitive types the performance boost over the CS106B/X `Stack` is appreciable.

queue

Another simple STL container class is the `queue`, which behaves in much the same way as the CS106B/X `Queue`. Here's another summary table:

CS106B/X Queue	STL queue	Differences
<code>int size()</code>	<code>size_type size() const</code>	Although the STL <code>queue</code> 's return type is <code>size_type</code> instead of <code>int</code> , these functions behave identically.
<code>bool isEmpty()</code>	<code>bool empty() const</code>	<code>empty</code> does <i>not</i> empty out the <code>queue</code> . The STL uses the function name <code>empty</code> instead of the CS106B/X <code>isEmpty</code> , but they do exactly the same thing.
<code>T& peek()</code>	<code>T& front()</code> <code>const T& front() const</code>	These functions do exactly the same thing, except that the STL <code>front</code> function does not perform any bounds-checking.
<code>void enqueue(T toAdd)</code>	<code>void push(const T& toAdd)</code>	Although it's irksome that the <code>queue</code> uses “push” and “pop” instead of “enqueue” and “dequeue,” these functions are the same.
<code>T dequeue();</code>	<code>void pop();</code>	As with the <code>stack</code> , <code>pop</code> does not return the value it removes.
<code>void clear()</code>	-	There is no <code>queue</code> “clear” function.
-	<code>T& back();</code> <code>const T& back() const</code>	Returns a reference to the back of the <code>queue</code> , which is the last element that was pushed on.

Notice that the `enqueue/dequeue` functions you know from CS106B/X are now called `push` and `pop`. Although they are called `push` and `pop` instead of `enqueue` and `dequeue`, they behave identically to their CS106B/X `Queue` counterparts. This unfortunate syntax is one of the reasons that CS106B and CS106X don't use the STL.

vector

Fortunately (or unfortunately, depending on how you look at it) if you thought that the STL was exactly the same as the CS106B/X libraries with different function names, the only two “simple” STL containers are `stack` and `queue`.^{*} It's now time to deal with the STL `vector`, one of the STL's most ubiquitous containers. In this next section we'll talk about some basic `vector` functionality, concluding with a summary of important `vector` member functions.

Fixed-sized vectors

The simplest way to use a `vector` is to treat it as though it's a managed, fixed-size array. For now, we'll ignore the `vector`'s ability to dynamically resize itself and instead focus on the functions you can use to access the `vector`.

When you create a `vector`, you can set its default size as a parameter to the constructor like this:

```
vector<int> myIntVector;           // vector of size 0
vector<int> myIntVector(10);      // vector of size 10, elements set to zero.
vector<int> myIntVector(50, 137); // size 50, all elements are 137.
vector<string> myStrings(4, "blank"); // size 4, elements set to "blank"
```

With the CS106B/X `Vector`, if you write code like this:

```
Vector<int> myIntVector(100);
```

you are *not* creating a `Vector` with 100 elements. Instead you're providing a size hint about the expected size of the `Vector`. Thus, if you're porting code from CS106B/X to the STL equivalents, make sure to watch out for this discrepancy – it won't work the way you want it to!

* There is a third “simple” container, the `priority_queue`, which we won't cover here.

Interestingly, if you store primitive types like `int` or `double` in an STL `vector` and specify an initial size for the `vector`, the `vector` contents will default to zero. This is not the behavior you see in the CS106B/X ADTs, so be aware of the distinction.

Now, of course, a `vector` can't do much anything if you can't access any of its data. Fortunately, the STL `vector` is much like the CS106B/X `Vector` in that you can access its data in two ways, as shown below:

```
vector<int> myInts(10);
myInts[0] = 10;      // Set the first element to 10.
myInts.at(0) = 10;   // Set the first element to 10.
int ninthElement = myInts[9];
int ninthElement = myInts.at(9);
```

There is a subtle difference between the two syntaxes, but for now it's safe to ignore it. Just remember that in both cases, if you go off the end of the `vector`, you're likely to get your program to crash, so make sure to bounds-check your accesses!

As with the `stack` and `queue`, the `vector` uses `size` and `empty` to return the size and determine if the `vector` is empty, respectively. For example:

```
vector<int> myVector(10);
for(vector<int>::size_type h = 0; h < myVector.size(); ++h)
    myVector[h] = h;
```

In this above code, note that we used the type `vector<int>::size_type` for our iteration variable instead of the more traditional `int`. All STL container classes define a special `size_type` type that holds only positive integers because containers cannot hold negative numbers of elements. In nearly every case it's completely safe to ignore the distinction and use `ints` for tracking these variables, but it's good to know the distinction since your compiler might complain about “implicit type conversions with possible losses of data.”

If you want to clear out all of the entries of a `vector`, use the `clear` member function, which behaves the same way as the CS106B/X ADT `clear` member functions:

```
vector<int> myVector(1000);
cout << myVector.size() << endl;    // Prints 1000
myVector.clear();
cout << myVector.size() << endl;    // Prints 0
```

Variable-sized vectors

These above functions are indeed useful, but they treat the `vector` as though it were simply a fixed-sized array. However, the real power of the `vector` shows up when you `resize` it. The simplest way to `resize` a `vector` is with the `resize` member function. `resize` has the same syntax as the `vector` constructor; that is, you can specify only a target size, or both a target size and a default value for any inserted elements. `resize` can be used both to expand and shrink a `vector`. If the `vector` expands, any new elements are appended to the end. If it shrinks, they're deleted off the end.

(For the examples below, assume we have a `PrintVector` function that takes in a `vector<int>` and prints it to the screen.*)

* For now you can just use a simple for loop to do this. Later, we'll see a much cooler way that involves iterators and algorithms.

```
vector<int> myVector;    // Defaults to empty vector
PrintVector(myVector);  // Output: [nothing]
myVector.resize(10);    // Grow the vector, setting new elements to 0
PrintVector(myVector);  // Output: 0 0 0 0 0 0 0 0 0 0
myVector.resize(5);     // Shrink the vector
PrintVector(myVector);  // Output: 0 0 0 0 0
myVector.resize(7, 1);  // Grow the vector, setting new elements to 1
PrintVector(myVector);  // Output: 0 0 0 0 0 1 1
myVector.resize(1, 7);  // The second parameter is effectively ignored.
PrintVector(myVector);  // Output: 0
```

Of course, sometimes you only want to add one element to the `vector`. If that new element is at the end of the `vector`, you can use the `push_back` method to “push” an element onto the back of the `vector`. This is equivalent to the CS106B/X `add` function. For example:

```
vector<int> myVector;
for(int i = 0; i < 10; ++i)
    myVector.push_back(i);
PrintVector(myVector); // Output: 0 1 2 3 4 5 6 7 8 9
```

Similarly, to take an element off the back of a `vector`, you can use `pop_back`, with behavior almost identical to that of the `stack`'s `pop` member function:

```
vector<int> myVector(10, 0);
while(!myVector.empty())
    myVector.pop_back();
```

The `vector` also has a member function `back` that returns a reference to the last element in the `vector`.

`push_back` and `pop_back` are nice, but what if you want to insert an element into an arbitrary point in the `vector`? This is possible, but the syntax is a bit quirky. Once you've finished the chapter on STL iterators, however, this code will seem fine. To insert an element into a `vector` at an arbitrary point, use this syntax:

```
myVector.insert(myVector.begin() + n, element);
```

Here, `n` represents the index at which you want to insert the element. So, for example, to insert the number 10 at the very beginning of an integer `vector`, you'd write

```
myVector.insert(myVector.begin(), 10);
```

Just as you can use `resize` to grow a `vector` and fill in the newly added elements, you can use `insert` to insert multiple copies of a single element by using this syntax:

```
myVector.insert(myVector.begin() + n, numCopies, element);
```

For example, to insert five copies of the number 137 at the start of a `vector`, you could use the syntax

```
myVector.insert(myVector.begin(), 5, 137).
```

To remove a single element from a random point in a `vector`, use the `erase` method as follows:

```
myVector.erase(myVector.begin() + n);
```

where n represents the index of the element to erase. If you want to remove elements in the range $[start, stop)$, you can alternatively use this syntax:*

```
myVector.erase(myVector.begin() + start, myVector.begin() + stop);
```

The following table summarizes most of the important member functions of the `vector`:

Again, we haven't covered `const` yet, so it's safe to ignore it for now. We also haven't covered iterators yet, so don't feel scared when you see iterator functions.

Constructor: <code>vector<T> ()</code>	<code>vector<int> myVector;</code> Constructs an empty vector.
Constructor: <code>vector<T> (size_type size)</code>	<code>vector<int> myVector(10);</code> Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: <code>vector<T> (size_type size, const T& default)</code>	<code>vector<string> myVector(5, "blank");</code> Constructs a vector of the specified size where each element is equal to the specified default value.
<code>size_type size() const;</code>	<code>for(int i = 0; i < myVector.size(); ++i) { ... }</code> Returns the number of elements in the vector.
<code>bool empty() const;</code>	<code>while(!myVector.empty()) { ... }</code> Returns whether the vector is empty.
<code>void clear();</code>	<code>myVector.clear();</code> Erases all the elements in the vector and sets the size to zero.
<code>T& operator [] (size_type position);</code> <code>const T& operator [] (size_type position) const;</code> <code>T& at(size_type position);</code> <code>const T& at(size_type position) const;</code>	<code>myVector[0] = 100;</code> <code>int x = myVector[0];</code> <code>myVector.at(0) = 100;</code> <code>int x = myVector.at(0);</code> Returns a reference to the element at the specified position. The bracket notation <code>[]</code> does not do any bounds checking and has undefined behavior past the end of the data. The <code>at</code> member function will throw an exception if you try to access data beyond the end. We will cover exception handling in a later chapter.
<code>void resize(size_type newSize);</code> <code>void resize(size_type newSize, T fill);</code>	<code>myVector.resize(10);</code> <code>myVector.resize(10, "default");</code> Resizes the vector so that it's guaranteed to be the specified size. In the second version, the <code>vector</code> elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the <code>vector</code> , so you can't use <code>resize</code> to add elements to or remove elements from the start of the <code>vector</code> .

* The notation $[A, B)$ denotes the interval between A and B including A but excluding B . Since the domains considered in this text are usually discrete, the range $[A, B)$ corresponds to all elements between A and B , including A but excluding B . Thus $[0, 3)$ contains 0, 1, and 2 but not 3.

vector functions, contd.

<code>void push_back();</code>	<code>myVector.push_back(100);</code> Appends an element to the vector.
<code>T& back();</code> <code>const T& back() const;</code>	<code>myVector.back() = 5;</code> <code>int lastElem = myVector.back();</code> Returns a reference to the last element in the vector.
<code>T& front();</code> <code>const T& front() const;</code>	<code>myVector.front() = 0;</code> <code>int firstElem = myVector.front();</code> Returns a reference to the first element in the vector.
<code>void pop_back();</code>	<code>myVector.pop_back();</code> Removes the last element from the vector.
<code>iterator begin();</code> <code>const_iterator begin() const;</code>	<code>vector<int>::iterator itr = myVector.begin();</code> Returns an iterator that points to the first element in the vector.
<code>iterator end();</code> <code>const_iterator end() const;</code>	<code>while(itr != myVector.end());</code> Returns an iterator to the element <i>after</i> the last. The iterator returned by <code>end</code> does not point to an element in the vector.
<code>iterator insert(iterator position,</code> <code>const T& value);</code> <code>void insert(iterator start,</code> <code>size_type numCopies,</code> <code>const T& value);</code>	<code>myVector.insert(myVector.begin() + 4, "Hello");</code> <code>myVector.insert(myVector.begin(), 2, "Yo!");</code> The first version inserts the specified value into the vector, and the second inserts <code>numCopies</code> copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.
<code>iterator erase(iterator position);</code> <code>iterator erase(iterator start,</code> <code>iterator end);</code>	<code>myVector.erase(myVector.begin());</code> <code>myVector.erase(startItr, endItr);</code> The first version erases the element at the position pointed to by <code>position</code> . The second version erases all elements in the range <code>[startItr, endItr)</code> . Note that this does not erase the element pointed to by <code>endItr</code> . All iterators after the remove point are invalidated. If using this member function on a <code>deque</code> (see below), all iterators are invalidated.

As you can see, the `vector` is very powerful and has a whole bunch of useful functions. Others exist as well, so be sure to consult a reference.

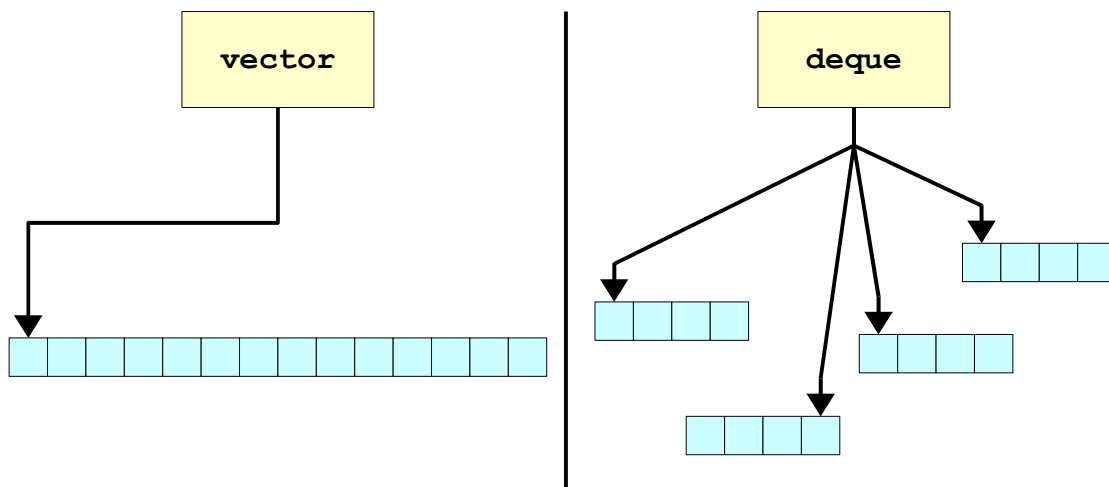
deque: Another sequence container

The STL `vector` is a linear sequence that grows and shrinks dynamically. While it's possible to add and remove elements anywhere in a `vector`, insertions and deletions are fastest at the end of the `vector` and are costly at the beginning. If we want to work with a sequence of elements that grows and shrinks at both ends, we could do so using a `vector`, but would suffer a severe performance hit. Is there some way that we can model a linear sequence with efficient insertion and deletion at both ends?

As mentioned at the start of this chapter, each of the CS106B/X ADTs (excluding `Grid`) has a corresponding STL container. However, the STL has several containers which have no analog in the CS106B/X ADTs. One such container is the `deque`. `deque` (pronounced “deck,” as in a deck of cards, and not “dequeue,” the queue's

remove element operation) stands for “double-ended queue” and is a sequence container like `vector` that supports efficient insertion and removal of elements at both the beginning and the end of its sequence. What's interesting about the `deque` is that all operations supported by `vector` are also provided by `deque`. Thus we can `resize` a `deque`, use the bracket syntax to access individual elements, and erase elements at arbitrary positions. However, `deque`s also support two more functions, `push_front` and `pop_front`, which work like the `vector`'s `push_back` and `pop_back` except that they insert and remove elements at the front of the `deque`. These four functions (`push_back`, `pop_back`, `push_front`, `pop_front`) allow the `deque` to mimic other data structures we've discussed this chapter. For example, a `deque` can act like a `stack` if we use `push_front` and `pop_front` to insert and remove elements at the beginning of its sequence, and similarly could model a `queue` by calling `pop_front` to remove elements and `push_back` to add them. This is not a coincidence – the `stack` is actually implemented as a wrapper class that manipulates a `deque`,* as is the `queue`. Because the `stack` and `queue` “adapt” the `deque` from one data type to another, they are not technically containers, but “container adapters.”

If `deque` has more functionality than `vector`, why use `vector`? The main reason is speed. `deque`s and `vectors` are implemented in two different ways. Typically, a `vector` stores its elements in contiguous memory addresses. `deque`s, on the other hand, maintain a list of different “pages” that store information. This is shown here:



These different implementations impact the efficiency of the `vector` and `deque` operations. In a `vector`, because all elements are stored in consecutive locations, it is possible to locate elements through simple arithmetic: to look up the n th element of a `vector`, find the address of the first element in the `vector`, then jump forward n positions. In a `deque` this lookup is more complex: the `deque` has to figure out which page the element will be stored in, then has to search that page for the proper item. However, inserting elements at the front of a `vector` requires the `vector` to shuffle all existing elements down to make room for the new element (slow), while doing the same in the `deque` only requires the `deque` to rearrange elements in a single page (fast).

If you're debating about whether to use a `vector` or a `deque` in a particular application, you might appreciate this advice from the C++ ISO Standard (section 23.1.1.2):

vector is the type of sequence that should be used by default... deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

* One might even say that it's stacking the `deque`. ☺

If you ever find yourself about to use a `vector`, check to see what you're doing with it. If you need to optimize for fast access, keep using a `vector`. If you're going to be inserting or deleting elements at the beginning or end of the container frequently, consider using a `deque` instead.

More To Explore

There's so much to explore with the STL that we could easily fill the rest of the course reader with STL content. If you're interested in some more advanced topics relating to this material and the STL in general, consider reading on these topics:

1. **valarray**: The `valarray` class is similar to a `vector` in that it's a managed array that can hold elements of any type. However, unlike `vector`, `valarray` is designed for numerical computations. `valarrays` are fixed-size and have intrinsic support for mathematical operators. For example, you can use the syntax `myValArray *= 2` to multiply all of the entries in a `valarray` by two. If you're interested in numeric or computational programming, consider looking into the `valarray`.
2. There's an excellent article online comparing the performances of the `vector` and `deque` containers. If you're interested, you can see it at http://www.codeproject.com/vcpp/stl/vector_vs_deque.asp.

Practice Problems

Everything we've covered this chapter primarily concerns the basic syntax of STL containers and thus doesn't lend itself to “practice problems” in the traditional sense. However, I highly recommend that you try some of the following exercises:

1. Take some of the code you've seen from CS106B/X that uses the ADT library and rewrite it using the STL. What are the major differences between the two libraries? Is one easier to work with than another?
2. Deliberately write code with the STL that contains syntax errors and look at the error messages you get. See if you have any idea what they say, and, if not, practice until you can get a feel for what's going on. If you want a real mess, try making mistakes with `vector<string>`.
3. As mentioned above, the `deque` outperforms the `vector` when inserting and removing elements at the end of the container. However, the `vector` has a useful member function called `reserve` that can be used to increase its performance against the `deque` in certain circumstances. The `reserve` function accepts an integer as a parameter and acts as a sort of “size hint” to the `vector`. Behind the scenes, `reserve` works by allocating additional storage space for the `vector` elements, reducing the number of times that the `vector` has to ask for more storage space. Once you have called `reserve`, as long as the size of the `vector` is less than the number of elements you have reserved, calls to `push_back` and `insert` on the `vector` will execute more quickly than normal. Once the `vector` hits the size you reserved, these operations revert to their original speed.*

Write a program that uses `push_back` to insert a large number of strings into two different `vector`s – one which has had `reserve` called on it and one which hasn't – as well as a `deque`. The exact number and content of strings is up to you, but large numbers of long strings will give the most impressive results. Use the `clock()` function exported by `<ctime>` to compute how long it takes to finish inserting the strings; consult a reference for more information. Now repeat this trial, but insert the elements at the beginning of the container rather than the end. Did calling `reserve` help to make the `vector` more competitive against the `deque`?

4. Compare the performances of the `vector`'s bracket operator syntax (e.g. `myVector[0]`) to the `vector`'s `at` function (e.g. `myVector.at(0)`). Given that the `at` function is bounds-checked, how serious are the runtime penalties of bounds-checking?

* Calling `push_back` n times always takes $O(n)$ time, whether or not you call `reserve`. However, calling `reserve` reduces the constant term in the big- O to a smaller value, meaning that the overall execution time is lower.

5. In this next problem we'll explore a simple encryption algorithm called the *Vigenère cipher* and how to implement it using the STL containers.

One of the oldest known ciphers is the *Caesar cipher*, named for Julius Caesar, who allegedly employed it. The idea is simple. We pick a secret number between 1 and 26, inclusive, then encrypt the input string by replacing each letter with the letter that comes that many spaces after it. If this pushes us off the end of the alphabet, we wrap around to the start of the alphabet. For example, if we were given the string “The cookies are in the fridge” and picked the number 1, we would end up with the resulting string “Uif dppljft bsf jo uif gsje hf.” To decrypt the string, we simply need to push each letter backwards by one spot.

The Caesar cipher is an extremely weak form of encryption; it was broken in the ninth century by the Arab polymath al-Kindi. The problem is that the cipher preserves the relative frequencies of each of the letters in the source text. Not all letters appear in English with equal frequency – e and t are far more common than q or w, for example – and by looking at the relative letter frequencies in the encrypted text it is possible to determine which letter in the encrypted text corresponds to a letter in the source text and to recover the key.

The problem with the Caesar cipher is that it preserves letter frequencies because each letter is transformed using the same key. But what if we were to use *multiple* keys while encrypting the message? That is, we might encrypt the first letter with one key, the second with another, the third with yet another, etc. One way of doing this is to pick a sequence of numbers, then cycle through them while encrypting the text. For example, let's suppose that we want to encrypt the above message using the key string 1, 3, 7. Then we would do the following:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7
U	K	L	D	R	V	L	L	L	T	D	Y	F	L	U	U	K	L	G	U	P	E	J	L

Notice that the letters KIE from COOKIES are all mapped to the letter L, making cryptanalysis much more difficult. This particular encryption system is the Vigenère cipher.

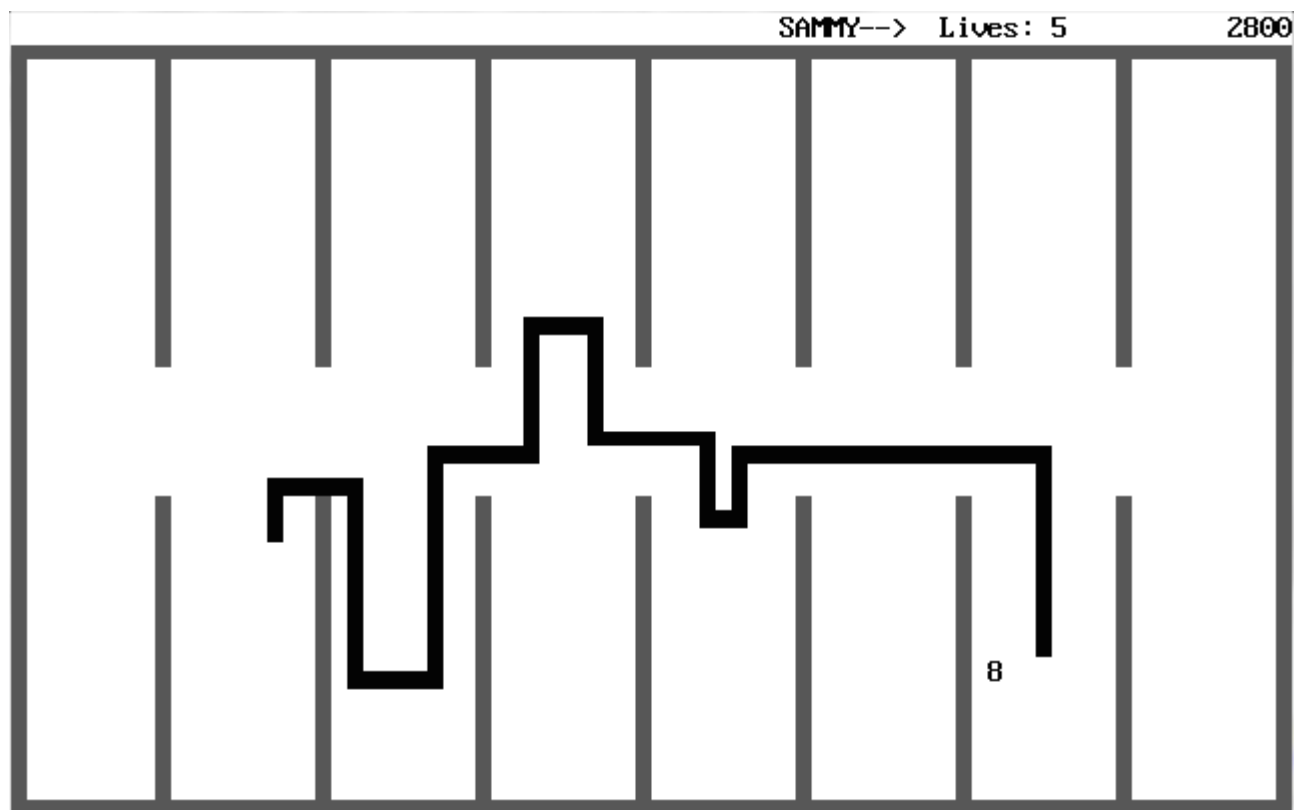
Now, let's consider what would happen if we wanted to implement this algorithm in C++ to work on arbitrary strings. Strings in C++ are composed of individual `chars`, which can take on (typically) one of 256 different values. If we had a list of integer keys, we could encrypt a string using the Vigenère cipher by simply cycling through those keys and incrementing the appropriate letters of the string. In fact, the algorithm is quite simple. We iterate over the characters of the string, at each point incrementing the character by the current key and then rotating the keys one cycle.

- a. Suppose that we want to represent a list of integer keys that can easily be cycled; that is, we want to efficiently support moving the first element of the list to the back. Of the containers and container adapters covered in the above section (`stack`, `queue`, `vector`, `deque`), which have the best support for this operation? ♦
- b. Based on your decision, implement a function `VigenereEncrypt` that accepts a `string` and a list of `int` keys stored in the container (or container adapter) of your choice, then encrypts the `string` using the Vigenère cipher. ♦

Chapter 5: Extended Example: Snake

Few computer games can boast the longevity or addictive power of *Snake*. Regardless of your background, chances are that you have played Snake or one of its many variants. The rules are simple – you control a snake on a two-dimensional grid and try to eat food pellets scattered around the grid. You lose if you crash into the walls. True to Newton's laws, the snake continues moving in a single direction until you explicitly change its bearing by ninety degrees. Every time the snake eats food, a new piece of food is randomly placed on the grid and the snake's length increases. Over time, the snake's body grows so long that it becomes an obstacle, and if the snake collides with itself the player loses.

Here's a screenshot from QBasic Nibbles, a Microsoft implementation of Snake released with MS-DOS version 5.0. The snake is the long black string at the bottom of the map, and the number 8 is the food:



Because the rules of Snake are so simple, it's possible to implement the entire game in only a few hundred lines of C++ code. In this extended example, we'll write a Snake program in which the *computer* controls the snake according to a simple AI. In the process, you'll gain experience with the STL `vector` and `deque`, the streams library, and a sprinkling of C library functions. Once we've finished, you'll have a rather snazzy program that can serve as a launching point for further C++ exploration.

Our Version of Snake

There are many variants of Snake, so to avoid confusion we'll explicitly spell out the rules of the game we're implementing:

1. The snake moves by extending its head in the direction it's moving and pulling its tail in one space.
2. The snake wins if it eats twenty pieces of food.
3. The snake loses if it crashes into itself or into a wall.
4. If the snake eats a piece of food, its length grows by one and a new piece of food is randomly placed.
5. There is only one level, the starting level.

While traditionally Snake is played by a human, our Snake will be computer-controlled so that we can explore some important pieces of the C runtime library. We'll discuss the AI we'll use when we begin implementing it.

Representing the World

In order to represent the Snake world, we need to keep track of the following information:

1. The size and layout of the world.
2. The location of the snake.
3. How many pieces of food we've consumed.

Let's consider this information one piece at a time. First, how should we represent the world? Since the world is two-dimensional, we will need to store this information in something akin to the CS106B/X `Grid`. Unfortunately, the STL doesn't have a container class that encapsulates a multidimensional array, but we can emulate this functionality with an STL `vector` of `vectors`. For example, if we represent each square with an object of type `WorldTile`, we could use a `vector<vector<WorldTile>>`. Note that there is a space between the two closing angle brackets – this is deliberate and is an unfortunate bug in the C++ specification. If we omit the space, C++ would interpret the closing braces on `vector<vector<WorldTile>>>` as the stream extraction operator `>>`, as in `cin >> myValue`. Although most compilers will accept code that uses two adjacent closing braces, it's bad practice to write it this way.

While we could use a `vector<vector<WorldTile>>`, there's actually a simpler option. Since we need to be able to display the world to the user, we can instead store the world as a `vector<string>` where each `string` encodes one row of the board. This also simplifies displaying the world; given a `vector<string>` representing all the world information, we can draw the board by outputting each string on its own line. Moreover, since we can use the bracket operator `[]` on both `vector` and `string`, we can use the familiar syntax `world[row][col]` to select individual locations. The first brackets select the `string` out of the `vector` and the second the character out of the `string`.

We'll use the following characters to encode game information:

- A space character (' ') represents an empty tile.
- A pound sign (' # ') represents a wall.
- A dollar sign (' \$ ') represents food.
- An asterisk (' * ') represents a tile occupied by a snake.

For simplicity, we'll bundle all the game data into a single `struct` called `gameT`. This will allow us to pass all the game information to functions as a single parameter. Based on the above information, we can begin writing this `struct` as follows:

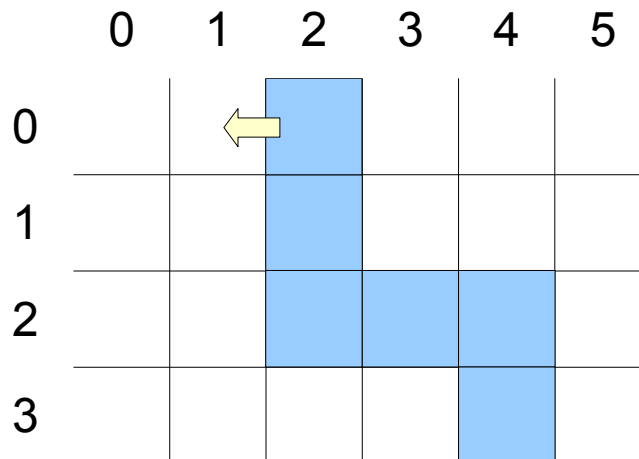
```
struct gameT
{
    vector<string> world;
    /* ... */
};
```

We also will need quick access to the dimensions of the playing field, since we will need to be able to check whether the snake is out of bounds. While we could access this information by checking the dimensions of the `vector` and the strings stored in it, for simplicity we'll store this information explicitly in the `gameT` struct, as shown here:

```
struct gameT
{
    vector<string> world;
    int numRows, numCols;
    /* ... */
};
```

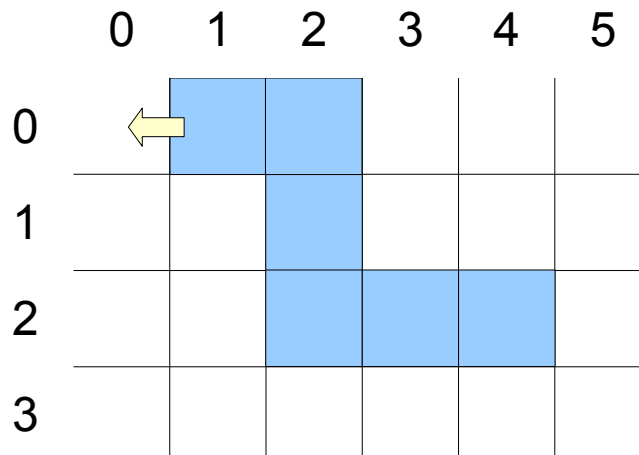
For consistency, we'll access elements in the `vector<string>` treating the first index as the row and the second as the column. Thus `world[3][5]` is row three, column five (where indices are zero-indexed).

Now, we need to settle on a representation for the snake. The snake lives on a two-dimensional grid and moves at a certain velocity. Because the grid is discrete, we can represent the snake as a collection of its points along with its velocity vector. For example, we can represent the following snake:



As the points (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3) and the velocity vector (-1, 0).

The points comprising the snake body are ordered to determine how the snake moves. When the snake moves, the first point (the *head*) moves one step in the direction of the velocity vector. The second piece then moves into the gap left by the first, the third moves into the gap left by the second piece, etc. This leaves a gap where the tail used to be. For example, after moving one step, the above snake looks like this:



To represent the snake in memory, we thus need to keep track of its velocity and an ordered list of the points comprising it. The former can be represented using two `ints`, one for the Δx component and one for the Δy component. But how should we represent the latter? In the previous chapter we learned about the `stack`, `queue`, `vector`, and `deque`, each of which could represent the snake. To see what the best option is, let's think about how we might implement snake motion. We can think of snake motion in one of two ways – first, as the head moving forward a step and the rest of the points shifting down one spot, and second as the snake getting a new point in front of its current head and losing its tail. The first approach requires us to update every element in the body and is not particularly efficient. The second approach can easily be implemented with a `deque` through an appropriate combination of `push_front` and `pop_back`. We will thus use a `deque` to encode the snake body.

If we want to have a `deque` of points, we'll first need some way of encoding a point. This can be done with this struct:

```
struct pointT
{
    int row, col;
};
```

Taking these new considerations into account, our new `gameT` struct looks like this:

```
struct gameT
{
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;

    /* ... */
};
```

Finally, we need to keep track of how many pieces of food we've munched so far. That can easily be stored in an `int`, yielding this final version of `gameT`:

```
struct gameT
{
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;

    int numEaten;
};
```

The Skeleton Implementation

Now that we've settled on a representation for our game, we can start thinking about how to organize the program. There are two logical steps – setup and gameplay – leading to the following skeleton implementation:

```

#include <iostream>
#include <string>
#include <deque>
#include <vector>
using namespace std;

/* Number of food pellets that must be eaten to win. */
const int kMaxFood = 20;

/* Constants for the different tile types. */
const char kEmptyTile = ' ';
const char kWallTile = '#';
const char kFoodTile = '$';
const char kSnakeTile = '*';

/* A struct encoding a point in a two-dimensional grid. */
struct pointT
{
    int row, col;
};

/* A struct containing relevant game information. */
struct gameT
{
    vector<string> world; // The playing field
    int numRows, numCols; // Size of the playing field

    deque<pointT> snake; // The snake body
    int dx, dy; // The snake direction

    int numEaten; // How much food we've eaten.
};

/* The main program.  Initializes the world, then runs the simulation. */
int main()
{
    gameT game;
    InitializeGame(game);
    RunSimulation(game);
    return 0;
}

```

Atop this program are the necessary `#includes` for the functions and objects we're using, followed by a list of constants for the game. The `pointT` and `gameT` structs are identical to those described above. `main` creates a `gameT` object, passes it into `InitializeGame` for initialization, and finally hands it to `RunSimulation` to play the game.

We'll begin by writing `InitializeGame` so that we can get a valid `gameT` for `RunSimulation`. But how should we initialize the game board? Should we use the same board every time, or let the user specify a level of their choosing? Both of these are reasonable, but for this extended example we'll choose the latter. In particular, we'll specify a level file format, then let the user specify which file to load at runtime.

There are many possible file formats to choose from, but each must contain at least enough information to populate a `gameT` struct; that is, we need the world dimensions and layout, the starting position of the snake, and the direction of the snake. While I encourage you to experiment with different structures, we'll use a simple file format that encodes the world as a list of strings and the rest of the data as integers in a particular order. Here is one possible file:

level.txt

```
15 15
1 0
#####
#$          $#
#   #       #   #
#   #       #   #
#   # $     #   #
#   #       #   #
#   #       #   #
#       *     #
#   #       #   #
#   #       #   #
#   # $     #   #
#   #       #   #
#   #       #   #
#$          $#
#####
```

The first two numbers encode the number of rows and columns in the file, respectively. The next line contains the initial snake velocity as Δx , Δy . The remaining lines encode the game board, using the same characters we settled on for the `world` vector. We'll assume that the snake is initially of length one and its position is given by a `*` character.

There are two steps necessary to let the user choose the level layout. First, we need to prompt the user for the name of the file to open, reprompting until she chooses an actual file. Second, we need to parse the contents of the file into a `gameT` struct. In this example we won't check that the file is formatted correctly, though in professional code we would certainly need to check this. If you'd like some additional practice with the streams library, this would be an excellent exercise.

Let's start writing the function responsible for loading the file from disk, `InitializeGame`. Since we need to prompt the user for a filename until she enters a valid file, we'll begin writing:

```
void InitializeGame(gameT& game)
{
    ifstream input;
    while(true)
    {
        cout << "Enter filename: ";
        string filename = GetLine();

        /* ... */
    }
    /* ... */
}
```

The `while(true)` loop will continuously prompt the user until she enters a valid file. Here, we assume that `GetLine()` is the version defined in the chapter on streams. Also, since we're now using the `ifstream` type, we'll need to `#include <fstream>` at the top of our program.

Now that the user has given us the a filename, we'll try opening it using the `.open()` member function. If the file opens successfully, we'll break out of the loop and start reading level data:


```

void InitializeGame(gameT& game)
{
    ifstream input;
    while(true)
    {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See the chapter on streams for .c_str().
        if(input.is_open()) break;

        /* ... */
    }
    /* ... */
}

```

If the file did *not* open, however, we need to report this to the user. Additionally, we have to make sure to reset the stream's error state, since opening a nonexistent file causes the stream to fail. Code for this is shown here:

```

void InitializeGame(gameT& game)
{
    ifstream input;
    while(true)
    {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See the chapter on streams for .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    /* ... */
}

```

Now we need to parse the file data into a `gameT` struct. Since this is rather involved, we'll decompose it into a helper function called `LoadWorld`, then finish `InitializeGame` as follows:

```

void InitializeGame(gameT& game)
{
    ifstream input;
    while(true)
    {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See the chapter on streams for .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    LoadWorld(game, input);
}

```

Notice that except for the call to `LoadWorld`, nothing in the code for `InitializeGame` actually pertains to our Snake game. In fact, the code we've written is a generic routine for opening a file specified by the user. We'll

thus break this function down into two functions – `OpenUserFile`, which prompts the user for a filename, and `InitializeGame`, which opens the specified file, then hands it off to `LoadWorld`. This is shown here:

```
void OpenUserFile(ifstream& input)
{
    while(true)
    {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See the chapter on streams for .c_str().
        if(input.is_open()) return;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
}

void InitializeGame(gameT& game)
{
    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}
```

Let's begin working on `LoadWorld`. The first line of our file format encodes the number of rows and columns in the world, and we can read this data directly into the `gameT` struct, as seen here:

```
void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    /* ... */
}
```

We've also resized the vector to hold `game.numRows` strings, guaranteeing that we have enough strings to store the entire world. This simplifies the implementation, as you'll see momentarily.

Next, we'll read the starting velocity for the snake, as shown here:

```
void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    /* ... */
}
```

At this point, we've read in the parameters of the world, and need to start reading in the actual world data. Since each line of the file contains one row of the grid, we'll use `getline` for the remaining read operations. There's a catch, however. Recall that `getline` does not mix well with the stream extraction operator (`>>`), which we've used exclusively so far. In particular, the first call to `getline` after using the stream extraction operator will return the empty string because the newline character delimiting the data is still waiting to be read. To prevent

this from gumming up the rest of our input operations, we'll call `getline` here on a dummy string to flush out the remaining newline:

```
void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    /* ... */
}
```

Now we're ready to start reading in world data. We'll read in `game.numRows` lines from the file directly into the `game.world` vector. Since earlier we resized the vector, there already are enough strings to hold all the data we'll read. The reading code is shown below:

```
void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row)
    {
        getline(input, game.world[row]);
        /* ... */
    }

    /* ... */
}
```

Recall that somewhere in the level file is a single `*` character indicating where the snake begins. To make sure that we set up the snake correctly, after reading in a line of the world data we'll check to see if it contains a star and, if so, we'll populate the `game.snake` deque appropriately. Using the `.find()` member function on the string simplifies this task, as shown here:

```

void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row)
    {
        getline(input, game.world[row]);
        int col = game.world[row].find(kSnakeTile);
        if(col != string::npos)
        {
            pointT head;
            head.row = row;
            head.col = col;
            game.snake.push_back(head);
        }
    }

    /* ... */
}

```

The syntax for creating and filling in the `pointT` data is a bit bulky here. When we cover classes in the second half of this course you'll see a much better way of creating this `pointT`. In the meantime, we can write a helper function to clean this code up, as shown here:

```

pointT MakePoint(int row, int col)
{
    pointT result;
    result.row = row;
    result.col = col;
    return result;
}

void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row)
    {
        getline(input, game.world[row]);
        int col = game.world[row].find(kSnakeTile);
        if(col != string::npos)
            game.snake.push_back(MakePoint(row, col));
    }

    /* ... */
}

```

There's one last step to take care of, and that's to ensure that we set the `numEaten` field to zero. This edit completes `LoadWorld` and the final version of the code is shown here:

```
void LoadWorld(gameT& game, ifstream& input)
{
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row)
    {
        getline(input, game.world[row]);
        int col = game.world[row].find(kSnakeTile);
        if(col != string::npos)
            game.snake.push_back(MakePoint(row, col));
    }

    game.numEaten = 0;
}
```

Great! We've just finished setup and it's now time to code up the actual game. We'll begin by coding a skeleton of `RunSimulation` which displays the current state of the game, runs the AI, and moves the snake:

```
void RunSimulation(gameT& game)
{
    /* Keep looping while we haven't eaten too much. */
    while(game.numEaten < kMaxFood)
    {
        PrintWorld(game);           // Display the board
        PerformAI(game);           // Have the AI choose an action

        if(!MoveSnake(game)) // Move the snake and stop if we crashed.
            break;

        Pause();                  // Pause so we can see what's going on.
    }
    DisplayResult(game);         // Tell the user what happened
}
```

We'll implement the functions referenced here out of order, starting with the simplest and moving to the most difficult. First, we'll begin by writing `Pause`, which stops for a short period of time to make the game seem more fluid. The particular implementation of `Pause` we'll use is a *busy loop*, a `while` loop that does nothing until enough time has elapsed. Busy loops are frowned upon in professional code because they waste CPU power, but for our purposes are perfectly acceptable.

The `<ctime>` header exports a function called `clock()` that returns the number of “clock ticks” that have elapsed since the program began. The duration of a clock tick varies from system to system, so C++ provides the constant `CLOCKS_PER_SEC` to convert clock ticks to seconds. We can use `clock` to implement a busy loop as follows:

1. Call `clock()` to get the current time in clock ticks and store the result.
2. Continuously call `clock()` and compare the result against the cached value. If enough time has passed, stop looping.

This can be coded as follows:

```
const double kWaitTime = 0.1; // Pause 0.1 seconds between frames
void Pause()
{
    clock_t startTime = clock(); // clock_t is a type designed to hold clock ticks.

    /* This loop does nothing except loop and check how much time is left. Note
     * that we have to typecast startTime from clock_t to double so that the
     * division is correct. The static_cast<double>(...) syntax is the preferred
     * C++ way of performing a typecast of this sort; see the chapter on
     * inheritance for more information.
     */
    while(static_cast<double>(clock() - startTime) / CLOCKS_PER_SEC < kWaitTime);
}
```

Next, let's implement the `PrintWorld` function, which displays the current state of the world. We chose to represent the world as a `vector<string>` to simplify this code, and as you can see this design decision pays off well:

```
void PrintWorld(gameT& game)
{
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

This implementation of `PrintWorld` is fine, but every time it executes it adds more text to the console instead of clearing what's already there. This makes it tricky to see what's happening. Unfortunately, standard C++ does not export a set of routines for manipulating the console. However, every major operating system exports its own console manipulation routines, primarily for developers working on a command line. For example, on a Linux system, typing `clear` into the console will clear its contents, while on Windows the command is `CLS`.

C++ absorbed C's standard library, including the `system` function (header file `<cstdlib>`). `system` executes an operating system-specific instruction as if you had typed it into your system's command line. This function can be very dangerous if used incorrectly,* but also greatly expands the power of C++. We will not cover how to use `system` in detail since it is platform-specific, but one particular application of `system` is to call the appropriate operating system function to clear the console. We can thus upgrade our implementation of `PrintWorld` as follows:

```
/* The string used to clear the display before printing the game board. Windows
 * systems should use "CLS"; Mac OS X or Linux users should use "clear" instead.
 */
const string kClearCommand = "CLS";

void PrintWorld(gameT& game)
{
    system(kClearCommand.c_str());
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

* In particular, calling `system` without checking that the parameters have been sanitized can let malicious users completely compromise your system. Take CS155 for more information on what sorts of attacks are possible.

Because `system` is from the days of pure C, we have to use `.c_str()` to convert the `string` parameter into a C-style string before we can pass it into the function.

The final quick function we'll write is `DisplayResult`, which is called after the game has ended to report whether the computer won or lost. This function is shown here:

```
void DisplayResult(gameT& game)
{
    PrintWorld(game);
    if(game.numEaten == kMaxFood)
        cout << "The snake ate enough food and wins!" << endl;
    else
        cout << "Oh no! The snake crashed!" << endl;
}
```

Now, on to the two tricky functions – `PerformAI`, which determines the snake's next move, and `MoveSnake`, which moves the snake and processes collisions. We'll begin with `PerformAI`.

Designing an AI that plays Snake intelligently is far beyond the scope of this class. However, it is feasible to build a rudimentary AI that plays reasonably well. Our particular AI works as follows: if the snake is about to collide with an object, the AI will turn the snake out of danger. Otherwise, the snake will continue on its current path, but has a percent chance to randomly change direction.

Let's begin by writing the code to check whether the snake will turn; that is, whether we're about to hit a wall or if the snake randomly decides to veer in a direction. We'll write a skeletal implementation of this code, then will implement the requisite functions. Our initial code is

```
const double kTurnRate = 0.2; // 20% chance to turn each step.
void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        /* ... */
    }
}
```

Here we're calling three functions we haven't written yet – `GetNextPosition`, which computes the position of the head on the next iteration; `Crashed`, which returns whether the snake would crash if its head was in the given position; and `RandomChance`, which returns true with probability equal to the parameter. Before implementing the rest of `PerformAI`, let's knock these functions out so we can focus on the rest of the task at hand. We begin by implementing `GetNextPosition`. This function accepts as input the game state and returns the point that we will occupy on the next frame if we continue moving in our current direction. This function isn't particularly complex and is shown here:

```

pointT GetNextPosition(gameT& game)
{
    /* Get the head position. */
    pointT result = game.snake.front();

    /* Increment the head position by the current direction. */
    result.row += game.dy;
    result.col += game.dx;
    return result;
}

```

The implementation of `Crashed` is similarly straightforward. The snake has crashed if it has gone out of bounds or if its head is on top of a wall or another part of the snake:

```

bool Crashed(pointT headPos, gameT& game)
{
    return !InWorld(headPos, game) ||
        game.world[headPos.row][headPos.col] == kSnakeTile ||
        game.world[headPos.row][headPos.col] == kWallTile;
}

```

Here, `InWorld` returns whether the point is in bounds and is defined as

```

bool InWorld(pointT& pt, gameT& game)
{
    return pt.col >= 0 &&
        pt.row >= 0 &&
        pt.col < game.numCols &&
        pt.row < game.numRows;
}

```

Next, we need to implement `RandomChance`. In CS106B/X we provide you a header file, `random.h`, that exports this function. However, `random.h` is not a standard C++ header file and thus we will not use it here. Instead, we will use C++'s `rand` and `srand` functions, also exported by `<cstdlib>`, to implement `RandomChance`. `rand()` returns a pseudorandom number in the range $[0, \text{RAND\_MAX}]$, where `RAND_MAX` is usually $2^{15} - 1$. `srand` seeds the random number generator with a value that determines which values are returned by `rand`. One common technique is to use the `time` function, which returns the current system time, as the seed for `srand` since different runs of the program will yield different random seeds. Traditionally, you will only call `srand` once per program, preferably during initialization. We'll thus modify `InitializeGame` so that it calls `srand` in addition to its other functionality:

```

void InitializeGame(gameT& game)
{
    /* Seed the randomizer. The static_cast converts the result of time(NULL) from
     * time_t to the unsigned int required by srand. This line is idiomatic C++.
     */
    srand(static_cast<unsigned int>(time(NULL)));

    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}

```

Now, let's implement `RandomChance`. To write this function, we'll call `rand` to obtain a value in the range $[0, \text{RAND\_MAX}]$, then divide it by `RAND_MAX + 1.0` to get a value in the range $[0, 1)$. We can then return whether this value is less than the input probability. This yields true with the specified probability; try

convincing yourself that this works if it doesn't immediately seem obvious. This is a common technique and in fact is how the CS106B/X `RandomChance` function is implemented.

`RandomChance` is shown here:

```
bool RandomChance(double probability)
{
    return (rand() / (RAND_MAX + 1.0)) < probability;
}
```

The `static_cast` is necessary here to convert the return value of `rand()` from an `int` to a `double`. Forgetting to do so will cause rounding errors that make the function most certainly nonrandom.

Phew! Apologies for the lengthy detour – let's get back to writing the AI! Recall that we've written this code so far:

```
void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        /* ... */
    }
}
```

We now need to implement the logic for turning the snake left or right. First, we'll figure out in what positions the snake's head would be if we turned left or right. Then, based on which of these positions are safe, we'll pick a direction to turn. To avoid code duplication, we'll modify our implementation of `GetNextPosition` so that the caller can specify the direction of motion, rather than relying on the `gameT`'s stored direction. The modified version of `GetNextPosition` is shown here:

```
pointT GetNextPosition(gameT& game, int dx, int dy)
{
    /* Get the head position. */
    gameT result = game.snake.front();

    /* Increment the head position by the specified direction. */
    result.row += dy;
    result.col += dx;
    return result;
}
```

We'll need to modify `PerformAI` to pass in the proper parameters to `GetNextPosition`, as shown here:

```

void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        /* ... */
    }
}

```

Now, let's write the rest of this code. Given that the snake's velocity is $(\text{game.dx}, \text{game.dy})$, what velocities would we move at if we were heading ninety degrees to the left or right? Using some basic linear algebra,* if our current heading is along dx and dy , then the headings after turning left and right from our current heading are given by

$$\begin{aligned} \text{dx}_{\text{left}} &= -\text{dy} \\ \text{dy}_{\text{left}} &= \text{dx} \\ \text{dx}_{\text{right}} &= \text{dy} \\ \text{dy}_{\text{right}} &= -\text{dx} \end{aligned}$$

Using these equalities, we can write the following code, which determines what bearings are available and whether it's safe to turn left or right:

```

void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx, leftDy), game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy), game);

        /* ... */
    }
}

```

Now, we'll decide which direction to turn. If we can only turn one direction, we will choose that direction. If we can't turn at all, we will do nothing. Finally, if we can turn either direction, we'll pick a direction randomly. We will store which direction to turn in a boolean variable called `willTurnLeft` which is `true` if we will turn left and `false` if we will turn right. This is shown here:

* This is the result of multiplying the vector $(\text{dx}, \text{dy})^T$ by a rotation matrix for either $+\pi/2$ or $-\pi/2$ radians.

```

void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),  game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy), game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        /* ... */
    }
}

```

Finally, we'll update our direction vector based on our choice:

```

void PerformAI(gameT& game)
{
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate))
    {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),  game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy), game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        game.dx = willTurnLeft? leftDx : rightDx;
        game.dy = willTurnLeft? leftDy : rightDy;
    }
}

```

If you're not familiar with the `? :` operator, the syntax is as follows:

expression ? result-if-true : result-if-false

Here, this means that we'll set `game.dx` to `leftDx` if `willTurnLeft` is true and to `rightDx` otherwise.

We now have a working version of `PerformAI`. Our resulting implementation is not particularly dense, and most of the work is factored out into the helper functions.

There is one task left – implementing `MoveSnake`. Recall that `MoveSnake` moves the snake one step forward on its path. If the snake crashes, the function returns `false` to indicate that the game is over. Otherwise, the function returns `true`.

The first thing to do in `MoveSnake` is to figure out where the snake's head will be after taking a step. Thanks to `GetNextPosition`, this has already been taken care of for us:

```

bool MoveSnake(gameT& game)
{
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    /* ... */
}

```

Now, if we crashed into something (either by falling off the map or by hitting an object), we'll return `false` so that the main loop can terminate:

```
bool MoveSnake(gameT& game)
{
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    /* ... */
}
```

Next, we need to check to see if we ate some food. We'll store this in a `bool` variable for now, since the logic for processing food will come a bit later:

```
bool MoveSnake(gameT& game)
{
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    /* ... */
}
```

Now, let's update the snake's head. We need to update the `world` vector so that the user can see that the snake's head is in a new square, and also need to update the `snake deque` so that the snake's head is now given by the new position. This is shown here:

```
bool MoveSnake(gameT& game)
{
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    /* ... */
}
```

Finally, it's time to move the snake's tail forward one step. However, if we've eaten any food, we will leave the tail as-is so that the snake grows by one tile. We'll also put food someplace else on the map so the snake has a new objective. The code for this is shown here:

```

bool MoveSnake(gameT& game)
{
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    if(!isFood)
    {
        game.world[game.snake.back().row][game.snake.back().col] = kEmptyTile;
        game.snake.pop_back();
    }
    else
    {
        ++game.numEaten;
        PlaceFood(game);
    }
    return true;
}

```

We're nearing the home stretch – all that's left to do is to implement `PlaceFood` and we're done! This function is simple – we'll just sit in a loop picking random locations on the board until we find an empty spot, then will put a piece of food there. To generate a random location on the board, we'll scale `rand()` down to the proper range using the modulus (%) operator. For example, on a world with four rows and ten columns, we'd pick as a row `rand() % 4` and as a column `col() % 10`. The code for this function is shown here:

```

void PlaceFood(gameT& game)
{
    while(true)
    {
        int row = rand() % game.numRows;
        int col = rand() % game.numCols;

        /* If the specified position is empty, place the food there. */
        if(game.world[row][col] == kEmptyTile)
        {
            game.world[row][col] = kFoodTile;
            return;
        }
    }
}

```

More to Explore

Congratulations on making it this far! We now have a complete working implementation of the Snake game with a computerized AI. The maps and world are completely customizable and you can create your own levels if you so choose. Moreover, in the process we've covered the STL, streams libraries, and a scattering of other libraries. However, this code has a lot of potential for extensions. If you're up for a challenge or just want to flex your newfound C++ muscles, try playing around with some of these ideas:

1. **Multiple snakes.** Some variants of Snake have multiple snakes each trying to collect as much food as possible. Each snake moves independently of the others and can be hurt by crashing into any snake, not just itself. Converting the above code to use multiple snakes would be a great way to play around with the STL and shouldn't be too difficult if you put the snake data into an STL `vector`.
2. **Bouncing balls.** One Snake variant which I am particularly fond of is *Rattler Race*, which was bundled with the Microsoft Entertainment Pack in 1990. In addition to snakes, this game featured bouncing balls which moved at 45° angles and reflected off of walls at 90° angles. The balls were harmless if they hit the snake's body, but were lethal if they hit the snake's head. Try adding bouncing balls to this implementation of snake. If you store the balls in a `vector` or `deque` appropriately, this will not be a difficult update.
3. **Multiple levels.** Many versions of Snake have multiple levels. Once the snake gathers a certain number of food pieces, it shrinks down to size one and then advances to the next level. See if you can think of a way to incorporate multiple levels to this Snake variant. One possibility would be to have the user specify a “puzzle file” at the start of the game containing a list of puzzle files, then to iterate through them as the user advances.
4. **Improved growing.** In our implementation of Snake, the snake grows by length one every time it eats food. Some Snake variants have the snake grow by progressively greater lengths as the size of the snake increases. Try modifying the code so that the snake's length increases by more than one.

Complete Snake Listing

Here is the complete source for the Snake example. Comments have been added to the relevant sections.

```
#include <iostream>
#include <string>
#include <deque>
#include <vector>
#include <fstream>
#include <cstdlib> // rand, srand, system
#include <ctime>   // clock, clock_t, CLOCKS_PER_SEC, time
using namespace std;

/* Probability of turning at each step. */
const double kTurnRate = 0.2;

/* Time to wait, in seconds, between frames. */
const double kWaitTime = 0.1;

/* Number of food pellets that must be eaten to win. */
const int kMaxFood = 20;

/* Constants for the different tile types. */
const char kEmptyTile = ' ';
const char kWallTile  = '#';
const char kFoodTile  = '$';
const char kSnakeTile = '*';

/* The string used to clear the display before printing the game board. Windows
 * systems should use "CLS", Mac OS X or Linux users should use "clear."
 */
const string kClearCommand = "CLS";

/* A struct encoding a point in a two-dimensional grid. */
struct pointT
{
    int row, col;
};

/* A struct containing relevant game information. */
struct gameT
{
    vector<string> world; // The playing field
    int numRows, numCols; // Size of the playing field

    deque<pointT> snake; // The snake body
    int dx, dy; // The snake direction

    int numEaten; // How much food we've eaten.
};

/* Reads a line of text from the user. */
string GetLine()
{
    string result;
    getline(cin, result);
    return result;
}
```



```
/* A helper function constructing a point at a given position. */
pointT MakePoint(int row, int col)
{
    pointT result;
    result.row = row;
    result.col = col;
    return result;
}

/* Returns true with probability probability. This works by scaling
 * down rand() by RAND_MAX + 1.0 so that we have a value in [0, 1) and returning
 * whether the value is less than the set probability.
 */
bool RandomChance(double probability)
{
    return (rand() / (RAND_MAX + 1.0)) < probability;
}

/* Places a piece of food randomly on the board. This assumes that there
 * is some free space remaining.
 */
void PlaceFood(gameT& game)
{
    while(true)
    {
        int row = rand() % game.numRows;
        int col = rand() % game.numCols;

        /* If there the specified position is empty, place the food there. */
        if(game.world[row][col] == kEmptyTile)
        {
            game.world[row][col] = kFoodTile;
            return;
        }
    }
}

/* Clears the display and prints the game board. */
void PrintWorld(gameT& game)
{
    /* Use a system call to clear the display. */
    system(kClearCommand.c_str());

    /* Print each row. */
    for(int i = 0; i < game.world.size(); ++i)
        cout << game.world[i] << endl;

    cout << "Food eaten: " << game.numEaten << endl;
}
```

```

/* Given an ifstream to a file containing CORRECTLY-FORMATTED world data,
 * loads in the world.
 *
 * The format used is as follows:
 * Line 1: numRows numCols
 * Line 2: dx dy
 * Rest:   World data
 *
 * We assume that the world is correctly-sized and that there is a single
 * '*' character in the world that's the starting point for the snake.
 */
void LoadWorld(gameT& game, ifstream& input)
{
    /* Read in the number of rows and columns. */
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    /* Read in the starting location. */
    input >> game.dx >> game.dy;

    /* Because we're going to be using getline() to read in the world
     * data, we need to make sure that we consume the newline character
     * at the end of the line containing the input data. We'll use
     * getline() to handle this. See the chapter on streams for why
     * this is necessary.
     */
    string dummy;
    getline(input, dummy);

    /* Read in the rows. */
    for(int row = 0; row < game.numRows; ++row)
    {
        getline(input, game.world[row]);

        /* Check to see if the * character (snake start position)
         * is in this line. If so, make the snake.
         */
        int col = game.world[row].find('*');
        if(col != string::npos)
            game.snake.push_back(MakePoint(row, col));
    }

    /* Set numEaten to zero - this needs to get done somewhere! */
    game.numEaten = 0;
}

/* Helper function which returns whether a point is contained in the game
 * grid.
 */
bool InWorld(pointT& pt, gameT& game)
{
    return pt.col >= 0 &&
           pt.row >= 0 &&
           pt.col < game.numCols &&
           pt.row < game.numRows;
}

```

```
/* Returns whether, if the snake head is at position head, the snake
 * has crashed.
 */
bool Crashed(pointT head, gameT& game)
{
    /* We crashed if the head is out of bounds, on a wall, or on another
     * snake piece.
     */
    return !InWorld(head, game) ||
        game.world[head.row][head.col] == kSnakeTile ||
        game.world[head.row][head.col] == kWallTile;
}

/* Returns the next position occupied by the head if the snake is moving
 * in the direction dx, dy.
 */
pointT GetNextPosition(gameT& game, int dx, int dy)
{
    /* Get the head. */
    pointT nextSpot = game.snake.front();

    /* Update it. */
    nextSpot.col += dx;
    nextSpot.row += dy;

    return nextSpot;
}
```

```

/* Performs AI logic to control the snake. The behavior is as follows:
 * 1. If we are going to crash, we try to turn.
 * 2. Independently, we have a percent chance to turn at each step.
 * 3. If we do have to turn, we always turn in a safe direction, and if we have
 *    multiple options we pick one randomly.
 */
void PerformAI(gameT& game)
{
    /* Look where we're going to be next step. */
    pointT nextSpot = GetNextPosition(game, game.dx, game.dy);

    /* If this crashes us or we just feel like turning, turn. */
    if(Crashed(nextSpot, game) || RandomChance(kTurnRate))
    {
        /* Compute what direction we'd be facing if we turned left or
         * right. From linear algebra we have the following:
         *
         * For a left turn:
         * |x'|   |0 -1||x| --> x' = -y
         * |y'| = |1  0||y| --> y' =  x
         *
         * For a right turn:
         * |x'|   |0  1||x| --> x' =  y
         * |y'| = |-1 0||y| --> y' = -x
         */
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft = !Crashed(GetNextPosition(game, leftDx, leftDy), game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy), game);

        /* Now determine which direction to turn based on what direction
         * we're facing. If we can choose either direction, pick one
         * randomly. If we can't turn, don't.
         */
        bool willTurnLeft;
        if(!canLeft && !canRight)
            return;
        else if(canLeft && !canRight)
            willTurnLeft = true;
        else if(!canLeft && canRight)
            willTurnLeft = false;
        else
            willTurnLeft = RandomChance(0.5);

        /* Based on the direction, turn appropriately. */
        game.dx = willTurnLeft? leftDx : rightDx;
        game.dy = willTurnLeft? leftDy : rightDy;
    }
}

```

```

/* Moves the snake one step in its current direction and handles collisions
 * and eating food. Returns true if we didn't crash, false if we did.
 */
bool MoveSnake(gameT& game)
{
    /* Compute new head. */
    pointT nextSpot = GetNextPosition(game, game.dx, game.dy);

    /* Check for dead. */
    if(Crashed(nextSpot, game))
        return false;

    /* Remember whether we just ate food. */
    bool isFood = (game.world[nextSpot.row][nextSpot.col] == kFoodTile);

    /* Update the display. */
    game.world[nextSpot.row][nextSpot.col] = kSnakeTile;

    /* Push new head. */
    game.snake.push_front(nextSpot);

    /* If we got food, pick a new spot and don't remove the tail. This causes us
     * to extend by one spot.
     */
    if(isFood)
    {
        PlaceFood(game);
        ++game.numEaten;
    }
    else
    {
        /* Clear the tail and remove it from the snake. */
        game.world[game.snake.back().row][game.snake.back().col] = kEmptyTile;
        game.snake.pop_back();
    }
    return true;
}

/* Pauses for a few milliseconds so we can see what's happening. This is
 * implemented using a busy loop, which is less-than-optimal but doesn't
 * require platform-specific features.
 */
void Pause()
{
    clock_t start = clock();
    while(static_cast<double>(clock() - start) / CLOCKS_PER_SEC < kWaitTime);
}

/* Displays the result of the game. */
void DisplayResult(gameT& game)
{
    PrintWorld(game);
    if(game.numEaten == kMaxFood)
        cout << "Yay! The snake won!" << endl;
    else
        cout << "Oh no! The snake crashed!" << endl;
}

```

```
/* Prompts the user for a filename, then opens the specified file. */
void OpenUserFile(istream& input)
{
    while(true)
    {
        cout << "Enter level file: ";
        input.open(GetLine().c_str());

        if(!input.fail()) return;

        cout << "Sorry, I can't open that file." << endl;
        input.clear();
    }
}

/* Initializes the game and loads the level file. */
void InitializeGame(gameT& game)
{
    /* Seed the randomizer. */
    srand(static_cast<int>(time(NULL)));

    istream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}

/* Runs the simulation and displays the result. */
void RunSimulation(gameT& game)
{
    /* Keep looping while we haven't eaten too much. */
    while(game.numEaten < kMaxFood)
    {
        PrintWorld(game);
        PerformAI(game);

        /* Move the snake and abort if we crashed. */
        if(!MoveSnake(game))
            break;
        Pause();
    }
    DisplayResult(game);
}

/* The main program.  Initializes the world, then runs the simulation. */
int main()
{
    gameT game;
    InitializeGame(game);
    RunSimulation(game);
    return 0;
}
```

Chapter 6: STL Iterators

The STL comprises several different container types, from the linear `vector` to the associative `map`. To provide a unified means of accessing and modifying elements in different container types, the STL uses *iterators*, objects that traverse ranges of data. While the benefits of iterators might not be apparent right now, when we cover algorithms in a later chapter you will see exactly how much power and flexibility iterators afford. This chapter explores a number of issues pertaining to iterators, beginning with basic syntax and ending with the iterator hierarchy and iterator adapters.

A Word on Safety, Syntax, and Readability

The iterators you've been exposed to in CS106B/X are Java-style iterators, which support functions called `next` and `hasNext` to return values and identify whether they may safely advance. Unfortunately, STL iterators, though providing similar functionality, have a completely different syntax. STL iterators are designed to look like raw C++ pointers (see the chapter on C strings for more details), and like pointers do not perform any bounds checking. However, as with every other part of the STL, this design decision leads to impressive performance gains. Iterators are optimized for speed, and in some cases you can increase the performance of your code by switching to iterator loops over traditional `for` loops.

A Word on Templates

All STL iterators have a common interface – that is, operations on an iterator for a `deque<string>` have the same syntax as operations on an iterator for a `set<int>`. This commonality paves the way for STL algorithms, which we'll cover in an upcoming chapter. However, because iterators are so commonly used in template functions, you are likely to encounter some very nasty compiler messages if you make syntax errors with iterators. A single mistake can cause a cascade of catastrophic compiler complaints that can really ruin your day. If you encounter these messages, be prepared to sift through them for a while before you find the root of your problem.

Now, without further ado, let's start talking about iterators!

Basic STL Iterators

Rather than jumping head-first into full-fledged iterator syntax, we'll start off at the basics and take smaller steps until we end up with the idiomatic STL iterator loop.

All STL container classes, except for `stack` and `queue`, define the `iterator` type.* For example, to declare an iterator for a `vector<int>`, we'd use the syntax

```
vector<int>::iterator myItr; // Correct, but iterator is uninitialized.
```

Note that unlike in the CS106B/X libraries, the word `iterator` is in lowercase. Also, keep in mind that this iterator can only iterate over a `vector<int>`; if we wanted to iterate over a `vector<string>` or `deque<int>`, we'd have to declare iterators of the type `vector<string>::iterator` and `deque<int>::iterator`, respectively.

* As mentioned in the chapter on STL containers, `stack` and `queue` are not technically containers (they're *container adapters*) and therefore do not have `iterators`.

STL iterators are designed to work like C and C++ pointers. Thus, to get the value of the element pointed at by an iterator, you “dereference” it using the `*` operator. Unlike CS106B/X iterators, STL iterators can modify the contents of the container they are iterating over. Here is some code using iterators to read and write values:

```
/* Assumes myItr is of type vector<int>::iterator */
int value = *myItr; // Read the element pointed at by the iterator.
*myItr = 137;      // Set the element pointed to by the iterator to 137.
```

When working with containers of objects, classes, or structs, you can use the `->` operator to select a data member or member function of an object pointed at by an iterator. For example, here's some code to print out the length of a string in a `vector<string>`:

```
/* Assumes myItr is of type vector<string>::iterator */
cout << myItr->length() << endl; // Print length of the current string.
```

Using an uninitialized iterator causes *undefined behavior*, which commonly manifests as a crash. Before you work with an iterator, make sure to initialize it to point to the elements of a container. Commonly you'll use the STL containers' `begin` member functions, which return iterators to the first elements of the containers. For example:

```
vector<int>::iterator itr = myVector.begin(); // Initialize the iterator
*itr = 137; // myVector[0] is now set to 137.
```

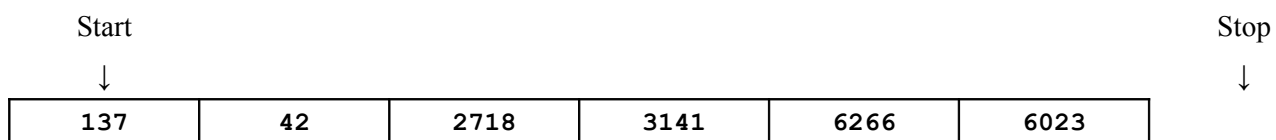
Note that unlike CS106B/X iterators' `next` function, getting the value of an STL iterator using `*` does not advance it to the next item in the container. To advance an STL iterator, use the `++` operator, as shown below.

```
vector<int>::iterator itr = myVector.begin();
*itr = 137; // myVector[0] is now 137.
++itr;     // Advance to next element.
*itr = 42; // myVector[1] is now 42.
```

You might be wondering why the above code uses `++itr` rather than `itr++`. For technical reasons involving operator overloading (covered in a later chapter of the course reader), although both `++itr` and `itr++` advance the iterator forward one position, the first version is slightly faster than the second.

We now know how to set up an iterator, read and write values, and advance the iterator forward, but we have not yet described how to tell when an iterator has iterated over all of the elements in a container. With CS106B/X iterators, you can easily tell when you've hit the end of a container by checking to see if `hasNext` returns `false`. However, STL iterators don't have a `hasNext` function or even rudimentary functionality like it. With STL iterators you need *two* iterators to define a range – one for the beginning and one for the end. While this makes the syntax a bit trickier, it makes STL iterators more flexible than CS106B/X iterators because it's possible to iterate over an arbitrary range instead of the entire container.

Each STL container class defines an `end` function that returns an iterator to the element *one past the end* of the container. Put another way, `end` returns the first iterator that is not in the container. While this seems confusing, it's actually quite useful because it lets you use iterators to define ranges of the type `[start, stop)`. You can see this visually below:



To best see how to use the `end` function, consider the following code snippet, which is the idiomatic “loop over a container” for loop:

```
for(vector<int>::iterator itr = myVector.begin(); itr != myVector.end(); ++itr)
    cout << *itr << endl;
```

Here, we simply crawl over the container by stepping forward one element at a time until the iterator reaches `end`. Notice that the end condition is `itr != myVector.end()`, whether the iterator traversing the range has hit the end of the sequence.

If you'll remember from our discussion of the `vector`, the syntax to remove elements from the `vector` is `myVector.erase(myVector.begin() + n)`, where `n` represented the index of the element to remove. The reason this code works correctly is because you can add integer values to `vector` iterators to obtain iterators that many positions past the current iterator. Thus to get an iterator that points to the `n`th element of the `vector`, we simply get an iterator to the beginning of the `vector` with `begin` and add `n` to it. This same trick works with a `deque`.

Iterator Generality

As mentioned in the previous chapter, the `deque` class supports the same functionality as the `vector`. This holds true for iterators, and in fact `deque` iterators have identical syntax and semantics to `vector` iterators. For example, here's some basic code to print out all elements of a `deque`:

```
for(deque<int>::iterator itr = myDeque.begin(); itr != myDeque.end(); ++itr)
    cout << *itr << endl;
```

`deques` and `vectors` are implemented differently – the `vector` with a single contiguous array, the `deque` with many chained arrays. Despite these differences, the iterator loops to crawl over the two containers have exactly the same structure. Somehow each iterator “knows” how to get from one element to the next, even if those elements aren't stored in consecutive memory locations. This is the real beauty of STL iterators – no matter how the data is stored, the iterators will access them correctly and with minimal syntax changes on your end. This is the magic of *operator overloading*, a technique we'll cover in the second half of this text.

Using Iterators to Define Ranges

The STL relies heavily on iterators to define ranges within containers. All of the examples we've seen so far have iterated exclusively over the range `[begin(), end())`, but it is possible and often useful to iterate over smaller ranges. For example, suppose you want to write a loop that will print the first ten values of a `deque` to a file. Using iterators, this can be accomplished easily as follows:

```
for(deque<int>::iterator itr = myDeque.begin();
    itr != myDeque().begin() + 10; // Ten steps down from the beginning.
    ++itr)
    myStream << *itr << endl;
```

Similarly, all containers support functions that let you access, manipulate, or add a range of data defined by iterators. For example, each container class provides an `insert` function that accepts a range of iterators and inserts all values in that range into the container. Here's an example:

```

/* Create a vector, fill it in with the first NUM_INTS integers. */
vector<int> myVector;
for(vector<int>::size_type h = 0; h < NUM_INTS; ++h)
    myVector.push_back(h);

/* Copy the first five elements of the vector into the deque */
deque<int> myDeque;
myDeque.insert(myDeque.begin(), // Start location of where to insert
               myVector.begin(), myVector.begin() + 5); // Values to insert

```

Even though the vector's iterators are of type `vector<int>::iterator` and not `deque<int>::iterator`, the code will compile. This is your first real glimpse of the magic of the STL: the fact that all iterators have the same interface means that the deque can accept iterators from any container, not just other deques.

What's even more interesting is that you can specify a range of iterators as arguments to the constructors of the associative containers (map and set, which are covered in the next chapter). When the new container is constructed, it will contain all of the elements specified by the range. This next code snippet fills in a set with the contents of a vector:

```

vector<int> myVector;
/* ... initialize myVector ... */

set<int> mySet(myVector.begin(), myVector.end());

```

This is really where the “template” in “Standard Template Library” begins to show its strength, and the upcoming chapter on algorithms will demonstrate exactly how powerful the common iterator interface can be.

Iterating Backwards

At some point you might want to traverse the elements of a container backwards. All STL containers define a type called `reverse_iterator` that represents an iterator that responds to the normal `++` operator backwards. For example, the statement `++myReverseItr` would result in `myReverseItr` pointing to the element that came *before* the current one. Similarly, containers have functions `rbegin` and `rend` that act as reversed versions of the traditional `begin` and `end`. Note, however, that `rbegin` does *not* point one past the end as does `end` – instead it points to the very last element in the container. Similarly, `rend` points one position *before* the first element, not to the same element as `begin`.

Here's some code showing off reverse iterators:

```

/* Print a vector backwards */
for(vector<int>::reverse_iterator itr = myVector.rbegin();
     itr != myVector.rend(); ++itr)
    cout << *itr << endl;

```

Iterator Categories

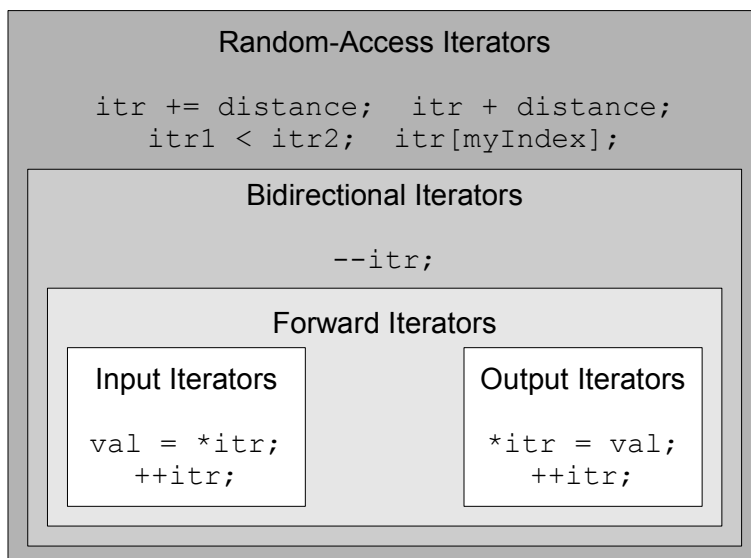
If you'll recall from the discussion of the vector and deque insert functions, to specify an iterator to the *n*th element of a vector, we used the syntax `myVector.begin() + n`. Although this syntax is legal in conjunction with vector and deque, it is illegal to use `+` operator with iterators for other container classes like map and set. At first this may seem strange – after all, there's nothing intuitively wrong with moving a set iterator forward multiple steps, but when you consider how the set is internally structured the reasons become more obvious. Unlike vector and deque, the elements in a map or set are not stored sequentially (usually they're kept in a balanced binary tree). Consequently, to advance an iterator *n* steps forward, the map or set iterator must take *n* individual steps forward. Contrast this with a vector iterator, where advancing forward *n* steps is a simple addition (since all of the vector's elements are stored contiguously). Since the runtime complexity of

advancing a `map` or `set` iterator forward n steps is linear in the size of the jump, whereas advancing a `vector` iterator is a constant-time operation, the STL disallows the `+` operator for `map` and `set` iterators to prevent subtle sources of inefficiency.

Because not all STL iterators can efficiently or legally perform all of the functions of every other iterator, STL iterators are categorized based on their relative power. At the high end are *random-access iterators* that can perform all of the possible iterator functions, and at the bottom are the *input* and *output* iterators which guarantee only a minimum of functionality. There are five different types of iterators, each of which is discussed in short detail below.

- **Output Iterators.** Output iterators are one of the two weakest types of iterators. With an output iterator, you can write values using the syntax `*myItr = value` and can advance the iterator forward one step using the `++` operator. However, you cannot read a value from an output iterator using the syntax `value = *myItr`, nor can you use the `+=` or `-` operators.
- **Input Iterators.** Input iterators are similar to output iterators except that they read values instead of writing them. That is, you can write code along the lines of `value = *myItr`, but not `*myItr = value`. Moreover, input iterators cannot iterate over the same range twice.
- **Forward Iterators.** Forward iterators combine the functionality of input and output iterators so that most intuitive operations are well-defined. With a forward iterator, you can write both `*myItr = value` and `value = *myItr`. Forward iterators, as their name suggests, can only move forward. Thus `++myItr` is legal, but `--myItr` is not.
- **Bidirectional Iterators.** Bidirectional iterators are the iterators exposed by `map` and `set` and encompass all of the functionality of forward iterators. Additionally, they can move backwards with the decrement operator. Thus it's possible to write `--myItr` to go back to the last element you visited, or even to traverse a list in reverse order. However, bidirectional iterators cannot respond to the `+` or `+=` operators.
- **Random-Access Iterators.** Don't get tripped up by the name – random-access iterators don't move around randomly. Random-access iterators get their name from their ability to move forward and backward by arbitrary amounts at any point. These are the iterators employed by `vector` and `deque` and represent the maximum possible functionality, including iterator-from-iterator subtraction, bracket syntax, and incrementation with `+` and `+=`.

If you'll notice, each class of iterators is progressively more powerful than the previous one – that is, the iterators form a functionality hierarchy. This means that when a library function requires a certain class of iterator, you can provide it any iterator that's at least as powerful. For example, if a function requires a forward iterator, you can provide either a forward, bidirectional, or random-access iterator. The iterator hierarchy is illustrated below:



string Iterators

The C++ `string` class exports its own iterator type and consequently is a container just like the `vector` or `map`. Like `vector` and `deque` iterators, `string` iterators are random-access iterators, so you can write expressions like `myString.begin() + n` to get an iterator to the *n*th element of a `string`. Most of the `string` member functions that require a start position and a length can also accept two iterators that define a range. For example, to replace characters three and four in a `string` with the string “STL,” you can write

```
myString.replace(myString.begin() + 3, myString.begin() + 5, "STL");
```

The `string` class also has several member functions similar to those of the `vector`, so be sure to consult a reference for more information.

Iterator Adapters

The STL uses the idea of an iterator range extensively, and anywhere that a range of elements is expected the STL will accept it using a pair of iterators. Interestingly, however, the STL never verifies that the iterators it receives actually correspond to iterators over a container. In fact, it's possible to build objects which look like iterators – for example, they can be dereferenced to a value with `*` and advanced forward with `++` – but which don't actually store elements inside an STL container. We can then plug these objects into the STL to “trick” the STL into performing complex operations behind-the-scenes. Such objects are called *iterator adapters* and are one of the cornerstones of advanced STL programming. To use the iterator adapters, you'll need to `#include` the `<iterator>` header.

One of the more common iterator adapters is `ostream_iterator`, which writes values to a stream. For example, consider the following code which uses an `ostream_iterator` to print values to `cout`:

```
ostream_iterator<int> myItr(cout, " ");
*myItr = 137; // Prints 137 to cout
++myItr;
*myItr = 42;  // Prints 42 to cout
++myItr
```

If you compile and run this code, you will notice that the numbers 137 and 42 get written to the console, separated by spaces. Although it *looks* like you're manipulating the contents of a container, you're actually writing characters to the `cout` stream.

`ostream_iterator` is a template type that requires you to specify what type of element you'd like to write to the stream. In the constructor, you must also specify an `ostream` to write to, which can be any output stream, including `cout`, `ofstreams` and `stringstreams`. The final argument to the constructor specifies an optional string to print out between elements. You may omit this if you want the contents to run together.

Another useful iterator adapter is the `back_insert_iterator`. With `back_insert_iterator`, you can append elements to a container using iterator syntax. For example, the following code creates a `vector<int>` and uses a `back_insert_iterator` to fill it in:

```
vector<int> myVector; /* Initially empty */
back_insert_iterator<vector<int> > itr(myVector); // Note that template argument
                                                    // is vector<int>.

for(int i = 0; i < 10; ++i)
{
    *itr = i; // "Write" to the back_insert_iterator, appending the value.
    ++itr;
}

for(vector<int>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    cout << *itr << endl; // Prints numbers zero through nine
```

Although we never explicitly added any elements to the `vector`, through the magic of iterator adapters we were able to populate the `vector` with data.

The syntax `back_insert_iterator<vector<int> >` is a bit clunky, and in most cases when you're using `back_insert_iterators` you'll only need to create a temporary object. For example, when using STL algorithms, you'll most likely want to create a `back_insert_iterator` only in the context of an algorithm. To do this, you can use the `back_inserter` function, which takes in a container and returns an initialized `back_insert_iterator` for use on that object.

Internally, `back_insert_iterator` calls `push_back` whenever it's dereferenced, so you can't use `back_insert_iterators` to insert elements into containers that don't have a `push_back` member function, such as `map` or `set`.

All of these examples are interesting, but why would you ever want to use an iterator adapter? After all, you can just write values directly to `cout` instead of using an `ostream_iterator`, and you can always call `push_back` to insert elements into containers. But iterator adapters have the advantage that they are iterators – that is, if a function expects an iterator, you can pass in an iterator adapter instead of a regular iterator. Suppose, for example, that you want to use an STL algorithm to perform a computation and print the result directly to `cout`. Unfortunately, STL algorithms aren't designed to write values to `cout` – they're written to store results in ranges defined by iterators. But by using an iterator adapter, you can trick the algorithm into “thinking” it's storing values but in reality is printing them to `cout`. Iterator adapters thus let you customize the behavior of STL algorithms by changing the way that they read and write data.

The following table lists some of the more common iterator adapters and provides some useful context. You'll likely refer to this table most when writing code that uses algorithms.

<code>back_insert_iterator<Container></code>	<pre>back_insert_iterator<vector<int> > itr(myVector); back_insert_iterator<deque<char> > itr = back_inserter(myDeque);</pre> An output iterator that stores elements by calling <code>push_back</code> on the specified container. You can declare <code>back_insert_iterators</code> explicitly, or can create them with the function <code>back_inserter</code>.

Common iterator adapters, contd.

<code>front_insert_iterator<Container></code>	<pre>front_insert_iterator<deque<int> > itr(myIntDeque); front_insert_iterator<deque<char> > itr = front_inserter(myDeque);</pre> An output iterator that stores elements by calling <code>push_front</code> on the specified container. Since the container must have a <code>push_front</code> member function, you cannot use a <code>front_insert_iterator</code> with a vector. As with <code>back_insert_iterator</code>, you can create <code>front_insert_iterators</code> with the <code>front_inserter</code> function.
<code>insert_iterator<Container></code>	<pre>insert_iterator<set<int> > itr(mySet, mySet.begin()); insert_iterator<set<int> > itr = inserter(mySet, mySet.begin());</pre> An output iterator that stores its elements by calling <code>insert</code> on the specified container to insert elements at the indicated position. You can use this iterator type to insert into any container, especially <code>set</code>. The special function <code>inserter</code> generates <code>insert_iterators</code> for you.
<code>ostream_iterator<Type></code>	<pre>ostream_iterator<int> itr(cout, " "); ostream_iterator<char> itr(cout); ostream_iterator<double> itr(myStream, "\n");</pre> An output iterator that writes elements into an output stream. In the constructor, you must initialize the <code>ostream_iterator</code> to point to an <code>ostream</code>, and can optionally provide a separator string written after every element.
<code>istream_iterator<Type></code>	<pre>istream_iterator<int> itr(cin); // Reads from cin istream_iterator<int> endItr; // Special end value</pre> An input iterator that reads values from the specified <code>istream</code> when dereferenced. When <code>istream_iterators</code> reach the end of their streams (for example, when reading from a file), they take on a special “end” value that you can get by creating an <code>istream_iterator</code> with no parameters. <code>istream_iterators</code> are susceptible to stream failures and should be used with care.
<code>ostreambuf_iterator<char></code>	<pre>ostreambuf_iterator<char> itr(cout); // Write to cout</pre> An output iterator that writes raw character data to an output stream. Unlike <code>ostream_iterator</code>, which can print values of any type, <code>ostreambuf_iterator</code> can only write individual characters. <code>ostreambuf_iterator</code> is usually used in conjunction with <code>istreambuf_iterator</code>.
<code>istreambuf_iterator<char></code>	<pre>istreambuf_iterator<char> itr(cin); // Read data from cin istreambuf_iterator<char> endItr; // Special end value</pre> An input iterator that reads unformatted data from an input stream. <code>istreambuf_iterator</code> always reads in character data and will not skip over whitespace. Like <code>istream_iterator</code>, <code>istreambuf_iterators</code> have a special iterator constructed with no parameters which indicates “end of stream.” <code>istreambuf_iterator</code> is used primarily to read raw data from a file for processing with the STL algorithms.

More to Explore

This chapter covers most of the iterator functions and scenarios you're likely to encounter in practice. While there are many other interesting iterator topics, most of them concern implementation techniques and though fascinating are far beyond the scope of this class. However, there are a few topics that might be worth looking into, some of which I've listed here:

1. **advance and distance:** Because not all iterators support the `+=` operator, the STL includes a nifty function called `advance` that efficiently advances any iterator by the specified distance. Using a technique known as *template metaprogramming*, `advance` will always choose the fastest possible means for advancing the iterator. For example, calling `advance` on a `vector` iterator is equivalent to using the `+=` operator, while advancing a `set` iterator is equivalent to a `for` loop that uses `++`. Similarly, there is a function called `distance` that efficiently computes the number of elements in a range defined by two iterators.
2. **reverse_iterator:** `reverse_iterator` is an iterator adapter that converts an iterator moving in one direction to an iterator moving in the opposite direction. The semantics of `reverse_iterator` are a bit tricky – for example, constructing a `reverse_iterator` from a regular STL iterator results in the `reverse_iterator` pointing to the element one *before* the element the original iterator points at – but in many cases `reverse_iterator` can be quite useful.
3. **The Boost Iterators:** The Boost C++ Libraries have many iterator adapters that perform a wide variety of tasks. For example, the `filter_iterator` type iterates over containers but skips over elements that don't match a certain predicate function. Also, the `transform_iterator` reads and writes elements only after first applying a transformation to them. If you're interesting in supercharging your code with iterator adapters, definitely look into the Boost libraries.

Practice Problems

1. Some STL containers contain member functions to return iterators to specific elements. As a sentinel value, these functions return the value of the container's `end()` function. Why do you think this is?
2. Write a function `DuplicateReversed` that accepts a `vector<int>` and returns a new `vector<int>` with the same values as the original `vector` but in reverse order.
3. What iterator category does `back_insert_iterator` belong to?
4. Suppose you want to write a template function that iterates over a container and doubles each element in-place. What is the least powerful iterator category that would be required for this function to work?
5. Using iterator adapters, write a function `LoadAllTokens` that, given a filename, returns a `vector` consisting of all of the tokens in that file. For our purposes, define a token to be a series of characters separated by any form of whitespace. While you can do this with a standard `ifstream` and a `while` loop, try to use `istream_iterators` instead. ♦

Chapter 7: STL Containers, Part II

Three chapters ago we talked about `vector` and `deque`, the STL's two managed array classes. However, the STL offers many other containers, two of which, `set` and `map`, we will cover in this chapter. Because `set` and `map` are associative containers, they require a decent understanding of iterators and iterator syntax. At the end, we'll quickly talk about two special container classes, the `multimap` and `multiset`, which have no counterparts in the CS106B/X ADT library.

typedef

The containers introduced in this chapter make extensive use of iterators and you may find yourself typing out lengthy type declarations frequently. For example, if you have a `set<string>`, the return type of its `insert` function will be `pair<set<string>::iterator, bool>`. This can become a nuisance at times and you may want to use the `typedef` keyword to simplify your code. `typedef` is a sort of “const for types” that lets you define shorthands for types that already exist. The syntax for `typedef` is

```
typedef type-name replacement
```

For example, you could define `IntVecItr` as a shorthand for `vector<int>::iterator` by writing

```
typedef vector<int>::iterator IntVecItr;
```

You can then iterate over a `vector<int>` by writing

```
for(IntVecItr itr = v.begin(); itr != v.end(); ++itr) { /* ... */ }
```

Keep `typedef` in the back of your mind as you read the rest of this chapter – you will undoubtedly find a host of uses for it as you continue your journey into the STL.

set

The STL `set`, like the CS106B/X `Set`, is an associative container representing an abstract collection of objects. You can test a `set` for inclusion, and can also insert and remove elements. However, unlike the CS106B/X `Set`, the STL `set` does not have intrinsic support for union, intersection, and difference, though in the upcoming chapter on algorithms we'll see how you can obtain this functionality.

Like the CS106B/X `Set`, when storing non-primitive types, the `set` requires you to specify a callback function that compares two values of the type being stored. Unfortunately, the syntax to provide a comparison callback for the `set` either uses complicated, error-prone syntax or requires an understanding of operator overloading, a language feature we haven't covered yet. Thus, for now, you should avoid storing custom data types in `sets`.

The most basic operation you can perform on a `set` is insertion using the `insert` function. Unlike the `deque` and `vector` `insert` functions, you do not need to specify a location for the new element because the `set` automatically orders its elements. Here is some sample code using `insert`:

```
set<int> mySet;
mySet.insert(137); // Now contains: 137
mySet.insert(42); // Now contains: 42 137 (in that order)
mySet.insert(137); // Now contains: 42 137 (note no duplicates)
```

To check whether the `set` contains an element, you can use the `find` function. `find` searches the `set` for an element and, if found, returns an iterator to it. Otherwise, `find` returns the value of the container's `end` function as a sentinel indicating the element wasn't found. For example, given the `set` initialized above:

```
if(mySet.find(137) != mySet.end())
    cout << "The set contains 137." << endl; // This is printed.

if(mySet.find(0) == mySet.end())
    cout << "The set doesn't contain 0." << endl; // Also printed.
```

Instead of `find`, you can also use the `count` function, which returns 1 if the element is contained in the `set` and 0 otherwise. Using C++'s automatic conversion of nonzero values into `true` and zero values to `false`, you can sometimes write cleaner code using `count`. For example:

```
if(mySet.count(137))
    cout << "137 is in the set." << endl; // Printed
if(!mySet.count(500))
    cout << "500 is not in the set." << endl; // Printed
```

If `count` is simpler than `find`, why use `find`? The reason is that sometimes you'll want an iterator to an element of a `set` for reasons other than determining membership. You might want to erase the value, for example, or perhaps you'll want to obtain iterators to a range of values for use with an STL algorithm. Thus, while `count` is quite useful, `find` is much more common in practice.

You can remove elements from a `set` using `erase`. There are several versions of `erase`, one of which removes the value pointed at by the specified iterator, and another that removes the specified value from the `set` if it exists. For example:

```
mySet.erase(137); // Removes 137, if it exists.
set<int>::iterator itr = mySet.find(42);
if(itr != mySet.end())
    mySet.erase(itr);
```

The STL `set` stores its elements in sorted order, meaning that if you iterate over a `set` you will see the values it contains in sorted order. For example, if we have a `set` containing the first five odd integers, the following code will print them out in ascending order:

```
for(set<int>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

Because `sets` store their elements in sorted order, it is possible to perform queries on an STL `set` that you cannot make on a C# `Set`. In particular, the `set` exports two functions, `lower_bound` and `upper_bound`, that can be used to iterate over the elements in a `set` that are within a certain range. `lower_bound` accepts a value, then returns an iterator to the first element in the `set` greater than or equal to that value. `upper_bound` similarly accepts a value and returns an iterator to the first element in the `set` that is strictly greater than the specified element. Given a closed range `[lower, upper]`, we can iterate over that range by using `lower_bound` to get an iterator to the first element no less than `lower` and iterating until we reach the value returned by `upper_bound`, the first element strictly greater than `upper`. For example, the following loop iterates over all elements in the `set` in the range `[10, 100]`:

```
set<int>::iterator stop = mySet.upper_bound(100);
for(set<int>::iterator itr = mySet.lower_bound(10); itr != stop; ++itr)
    /* ... perform tasks... */
```

A word of caution: you should not modify items pointed to by `set` iterators. Since `sets` are internally stored in sorted order, if you modify a value in a `set` in-place, you risk ruining the internal ordering. With the elements no longer in sorted order, the `set`'s searching routines may fail, leading to difficult-to-track bugs. If you want to apply a transformation to a `set`'s elements, unless you're certain the transformation will leave the elements in forward-sorted order, you should strongly consider creating a second `set` and storing the result there. In fact, some implementations of the STL won't even let you modify the elements of a `set`, an issue we'll address in the chapter on STL algorithms.

The following table lists some of the most important `set` functions, though there are more. The entries in this table are also applicable for the `map`, albeit with the appropriate modifications (see the next few sections for more information).

We haven't covered `const` yet, so for now it's safe to ignore it. We also haven't covered `const_iterators`, but for now you can just treat them as iterators that can't write any values.

Constructor: <code>set<T>()</code>	<pre>set<int> mySet;</pre> Constructs an empty <code>set</code>.
Constructor: <code>set<T>(const set<T>& other)</code>	<pre>set<int> myOtherSet = mySet;</pre> Constructs a <code>set</code> that's a copy of another <code>set</code>.
Constructor: <code>set<T>(InputIterator start, InputIterator stop)</code>	<pre>set<int> mySet(myVec.begin(), myVec.end());</pre> Constructs a <code>set</code> containing copies of the elements in the range <code>[start, stop)</code>. Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source.
<code>size_type size() const</code>	<pre>int numEntries = mySet.size();</pre> Returns the number of elements contained in the <code>set</code>.
<code>bool empty() const</code>	<pre>if(mySet.empty()) { ... }</pre> Returns whether the <code>set</code> is empty.
<code>void clear()</code>	<pre>mySet.clear();</pre> Removes all elements from the <code>set</code>.
<code>iterator begin()</code> <code>const_iterator begin() const</code>	<pre>set<int>::iterator itr = mySet.begin();</pre> Returns an iterator to the start of the <code>set</code>. Be careful when modifying elements in-place.
<code>iterator end()</code> <code>const_iterator end()</code>	<pre>while(itr != mySet.end()) { ... }</pre> Returns an iterator to the element one past the end of the final element of the <code>set</code>.
<code>pair<iterator, bool> insert(const T& value)</code> <code>void insert(InputIterator begin, InputIterator end)</code>	<pre>mySet.insert(4); mySet.insert(myVec.begin(), myVec.end());</pre> The first version inserts the specified value into the <code>set</code>. The return type is a <code>pair</code> containing an iterator to the element and a <code>bool</code> indicating whether the element was inserted successfully (<code>true</code>) or if it already existed (<code>false</code>). The second version inserts the specified range of elements into the <code>set</code>, ignoring duplicates.

set functions, contd.

<pre>iterator find(const T& element) const_iterator find(const T& element) const</pre>	<pre>if(mySet.find(0) != mySet.end()) { ... }</pre> Returns an iterator to the specified element if it exists, and <code>end</code> otherwise.
<pre>size_type count(const T& item) const</pre>	<pre>if(mySet.count(0)) { ... }</pre> Returns 1 if the specified element is contained in the <code>set</code>, and 0 otherwise.
<pre>size_type erase(const T& element) void erase(iterator itr); void erase(iterator start, iterator stop);</pre>	<pre>if(mySet.erase(0)) {...} // 0 was erased mySet.erase(mySet.begin()); mySet.erase(mySet.begin(), mySet.end());</pre> Removes an element from the <code>set</code>. In the first version, the specified element is removed if found, and the function returns 1 if the element was removed and 0 if it wasn't in the <code>set</code>. The second version removes the element pointed to by <code>itr</code>. The final version erases elements in the range <code>[start, stop)</code>.
<pre>iterator lower_bound(const T& value)</pre>	<pre>itr = mySet.lower_bound(5);</pre> Returns an iterator to the first element that is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with <code>upper_bound</code>.
<pre>iterator upper_bound(const T& value)</pre>	<pre>itr = mySet.upper_bound(100);</pre> Returns an iterator to the first element that is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to <code>upper_bound</code> to obtain all elements less than or equal to the parameter.

pair

Before introducing `map`, let us first quickly discuss `pair`, a template struct that stores two elements of mixed types. `pair` (defined in `<utility>`) accepts two template arguments and is declared as `pair<TypeOne, TypeTwo>`. `pair` has two fields, named `first` and `second`, which store the values of the two elements of the `pair`; `first` is a variable of type `TypeOne`, `second` of type `TypeTwo`.

You can create a `pair` explicitly using this syntax:

```
pair<int, string> myPair;
myPair.first = myInteger;
myPair.second = myString;
```

Alternatively, you can initialize the `pair`'s members in the constructor as follows:

```
pair<int, string> myPair(myInteger, myString);
```

In some instances, you will need to create a `pair` on-the-fly to pass as a parameter (especially to the `map`'s `insert`). You can therefore use the `make_pair` function as follows:

```
pair<int, string> myPair = make_pair(137, "string!");
```

Interestingly, even though we didn't specify what type of `pair` to create, the `make_pair` function was able to deduce the type of the `pair` from the types of the elements. This has to do with how C++ handles function templates and we'll explore this in more detail later.

map

The `map` is one of the STL's most useful containers. Like the CS106B/X `Map`, the STL `map` is an associative container which lets you query which value is associated with a given key. Unlike the CS106B/X `Map`, the STL `map` lets you choose what key type you want to use. That is, the STL `map` can map strings to ints, ints to strings, or even `vector<int>`s to `deque<double>`s. Also unlike the CS106B/X `Map`, the STL `map` stores its elements in sorted order. That is, if you iterate over the keys in a `map<string, int>`, you'll get all of the string keys back in alphabetical order. As a consequence, if you want to use nonstandard types as keys in a `map` – for example, your own custom structs – you will need to provide a comparison function. Again, like `set`, providing this function is quite difficult and we will not discuss how until we cover operator overloading.

To declare a `map`, use the following syntax:

```
map<KeyType, ValueType> myMap;
```

For example, to create a `map` similar to the CS106B/X `Map<int>`, you'd write `map<string, int>`, a mapping from strings to ints.

Unfortunately, while the STL `map` is powerful, its syntax is considerably more complicated than that of other containers because its functions and iterators need to accept, return, and manipulate two values. The `map`'s iterators act as pointers to `pair<const KeyType, ValueType>`s, pair of an immutable key and a mutable value. For example, to iterate over a `map<string, int>` and print out all its key/value pairs, you can use the following loop:

```
for(map<string, int>::iterator itr = myMap.begin(); itr != myMap.end(); ++itr)
    cout << itr->first << ": " << itr->second << endl;
```

Note that the key is `itr->first` and the value is `itr->second`. While you cannot change the value of the key during iteration,* you can modify its associated value.

As with the `set`, you can use `find` to return an iterator to an element in a `map` given a key. For example, to check to see if “STL” is a key in a `map<string, int>`, you can use this syntax:

```
if(myMap.find("STL") != myMap.end()) { ... }
```

Alternatively, you can use the `count` function as you would with a `set`.

Because the `map` stores everything as pairs, inserting a new value into a `map` is a syntax headache. You must manually create the pair to insert using the `make_pair` function discussed earlier. Thus, if you wanted to insert the key/value pair (“STL”, 137) into a `map<string, int>`, you must use this syntax:

```
myMap.insert(make_pair("STL", 137));
```

Unlike the CS106B/X `Map`, if you try to overwrite the value of an existing item with the `insert` function, the STL `map` won't modify the value. Thus, if you write

* As with the `set`, changing the values of the keys in a `map` could destroy the internal ordering and render the `map` non-functional. However, unlike the `set`, if you try to modify a key with an iterator, you'll get a compile-time error.

```
myMap.insert(make_pair("STL", 137)); // Inserted
myMap.insert(make_pair("STL", 42)); // Whoops! Not overwritten.
```

The value associated with the key “STL” will still be the value 137. To check whether your insertion actually created a new value, the map's `insert` function returns a `pair<iterator, bool>` that contains information about the result of the operation. The first element of the pair is an iterator to the key/value pair you just inserted, and the second element is a `bool` indicating whether the `insert` operation set the value appropriately. If this value is `false`, the value stored in the map was not updated. Thus, you can write code like this to insert a key/value pair and manually update the value if the key already existed:

```
/* Try to insert normally. */
pair<map<string, int>::iterator, bool> result =
    myMap.insert(make_pair("STL", 137));

/* If insertion failed, manually set the value. */
if(!result.second)
    result.first->second = 137;
```

In the last line, the expression `result.first->second` is the value of the existing entry, since `result.first` yields an iterator pointing to the entry, so `result.first->second` is the value field of the iterator to the entry. As you can see, the pair can make for tricky, unintuitive code.

There is an alternate syntax you can use to insert and access elements involving the bracket operator. As with the CS106B/X map, you can access individual elements by writing `myMap[key]`. If the element doesn't already exist, it will be created. Furthermore, if the key already exists, the value will be updated. For example, you can use the bracket operator to insert a new element into a `map<string, int>` as follows:

```
myMap["STL"] = 137;
```

You can also use the bracket operator to read a value from the map. As with insertion, if the element doesn't exist, it will be created. For example:

```
int myValue = myMap["STL"];
```

If the bracket notation is so much more convenient, why even bother with `insert`? As with everything in the STL, the reason is efficiency. Suppose you have a `map<string, vector<int>>`. If you insert a key/value pair using `myMap.insert(make_pair("STL", myVector))` the newly created `vector<int>` will be initialized to the contents of `myVector`. With `myMap["STL"] = myVector`, the bracket operation will first create an empty `vector<int>`, then assign `myVector` on top of it. As you'll see later in the chapter on class construction, this second version is noticeably slower than the first. As an STL programmer, you are working with professional libraries. If you want to go for raw speed, yes, you will have to deal with more complicated syntax, but if you want to just get the code up and running, then the STL won't have any reservations.

multimap and multiset

The STL provides two special “multicontainer” classes, `multimap` and `multiset`, that act as maps and sets except that the values and keys they store are not necessarily unique. That is, a `multiset` could contain several copies of the same value, while a `multimap` might have duplicate keys associated with different values.

`multimap` and `multiset` (declared in `<map>` and `<set>`, respectively) have identical syntax to `map` and `set`, except that some of the functions work slightly differently. For example, the `count` function will return the number of copies of an element in a multicontainer, not just a binary zero or one. Also, while `find` will still return an iterator to an element if it exists, the element it points to is not guaranteed to be the only copy of that ele-

ment in the multicontainer. Finally, the `erase` function will erase *all* copies of the specified key or element, not just the first it encounters.

One function that's quite useful for the multicontainers is `equal_range`. `equal_range` returns a `pair<iterator, iterator>` that represents the span of entries equal to the specified value. For example, given a `multimap<string, int>`, you could use the following code to iterate over all entries with key “STL”:

```
/* Store the result of the equal_range */
pair<multimap<string, int>::iterator, multimap<string, int>::iterator>
myPair = myMultiMap.equal_range("STL");

/* Iterate over it! */
for(multimap<string, int>::iterator itr = myPair.first;
    itr != myPair.second; ++itr)
    cout << itr->first << ": " << itr->second << endl;
```

As you can see, you might find yourself needing to `typedef` your iterator types a bit more with the multicontainers, but nonetheless they are quite useful.

More to Explore

In this chapter we covered `map` and `set`, which combined with `vector` and `deque` are the most commonly-used STL containers. However, there are several others we didn't cover, a few of which might be worth looking into. Here are some topics you might want to read up on:

1. **list:** `vector` and `deque` are sequential containers that mimic built-in arrays. The `list` container, however, models a sequence of elements without indices. `list` supports several nifty operations, such as merging, sorting, and splicing, and has quick insertions at almost any point. If you're planning on using a linked list for an operation, the `list` container is perfect for you.
2. **The Boost Containers:** The Boost C++ Libraries are a collection of functions and classes developed to augment C++'s native library support. Boost offers several new container classes that might be worth looking into. For example, `multi_array` is a container class that acts as a Grid in any number of dimensions. Also, the `unordered_set` and `unordered_map` act as replacements to the `set` and `map` that use hashing instead of tree structures to store their data. If you're interested in exploring these containers, head on over to www.boost.org.

Practice Problems

Some of these problems are programming exercises, others just cool things to think about. Play around with them a bit to get some practice with `map` and `set`!

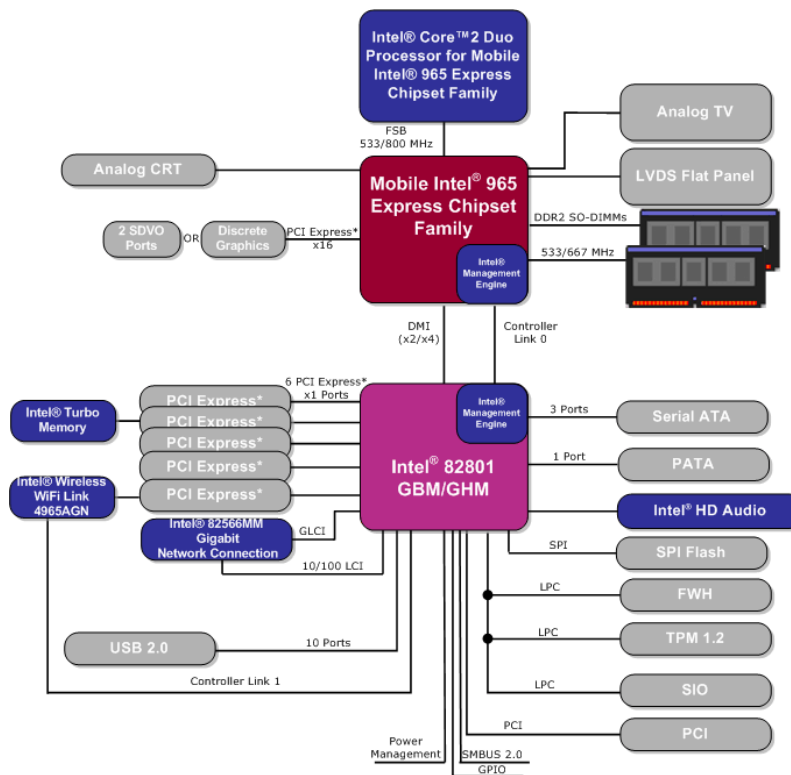
1. Using the timing code suggested in the practice problems from the first chapter on STL containers, time the speed differences of the `map`'s `insert` function versus the bracket syntax in a `map<int, int>`, then repeat the trial for `map<int, string>`. What does this tell you about the time it takes to copy objects? ♦
2. Write a function `NumberDuplicateEntries` that accepts a `map<string, string>` and returns the number of duplicate *values* in the `map` (that is, the number of key/value pairs in the `map` with the same value). ♦
3. Write a function `InvertMap` that accepts as input a `multimap<string, string>` and returns a `multimap<string, string>` where each pair (key, value) in the source `map` is represented by (value, key) in the generated `map`. Can you think of any applications for a function like this? ♦
4. As mentioned earlier, you can use a combination of `lower_bound` and `upper_bound` to iterate over elements in the closed interval `[min, max]`. What combination of these two functions could you use to iterate over the interval `[min, max)`? What about `(min, max]` and `(min, max)`? ♦
5. Write a function `CountLetters` that accepts as input an `ifstream` and a `map<char, int>` by reference, then updates the `map` such that each character that appears at least once in the file is mapped to the number of times that the character appears in the file. You can assume that the `map` is initially empty. As a useful FYI, unlike the CS106B/X `Map`, if you access a nonexistent key in the STL `map` using the bracket operators, the value associated with the key defaults to zero. That is, if you have a `map<string, int>` named `myMap` that doesn't contain the key "STL", then `myMap["STL"]` will yield zero (and also add "STL" as a key). Also note that to read a single character from a file, it's preferable to use the `get()` member function from the stream rather than the stream extraction operator `>>` since `get()` does not ignore whitespace. ♦
6. (Challenge problem!) Write a function `PrintMatchingPrefixes` that accepts a `set<string>` and a `string` containing a prefix and prints out all of the entries of the `set` that begin with that prefix. Your function should only iterate over the entries it finally prints out. You can assume the prefix is nonempty, consists only of alphanumeric characters, and should treat prefixes case-sensitively. (*Hint: In a `set<string>`, strings are sorted lexicographically, so all strings that start with "abc" will come before all strings that start with "abd."*) ♦

Chapter 8: Extended Example: Finite Automata

Computer science is often equated with programming and software engineering. Many a computer science student has to deal with the occasional “Oh, you’re a computer science major! Can you make me a website?” or “Computer science, eh? Why isn’t my Internet working?” This is hardly the case and computer science is a much broader discipline that encompasses many fields, such as artificial intelligence, graphics, and biocomputation.

One particular subdiscipline of computer science is *computability theory*. Since computer science involves so much programming, a good question is exactly *what* we can command a computer to do. What sorts of problems can we solve? How efficiently? What problems *can’t* we solve and why not? Many of the most important results in computer science, such as the undecidability of the halting problem, arise from computability theory.

But how exactly can we determine what can be computed with a computer? Modern computers are phenomenally complex machines. For example, here is a high-level model of the chipset for a mobile Intel processor: [Intel]



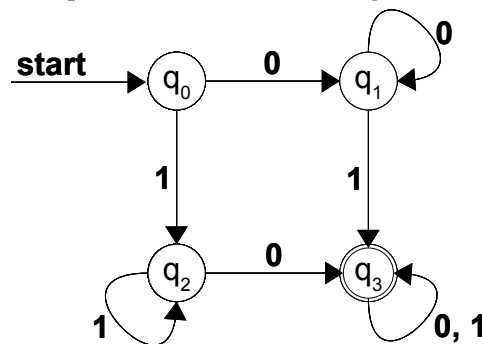
Modeling each of these components is exceptionally tricky, and trying to devise any sort of proof about the capabilities of such a machine would be all but impossible. Instead, one approach is to work with *automata*, abstract mathematical models of computing machines (the singular of *automata* is the plural of *automaton*). Some types of automata are realizable in the physical world (for example, deterministic and nondeterministic finite automata, as you’ll see below), while others are not. For example, the *Turing machine*, which computer scientists use as an overapproximation of modern computers, requires infinite storage space, as does the weaker *push-down automaton*.

Although much of automata theory is purely theoretical, many automata have direct applications to software engineering. For example, most production compilers simulate two particular types of automata (called *pushdown automata* and *nondeterministic finite automata*) to analyze and convert source code into a form readable by the compiler's semantic analyzer and code generator. Regular expression matchers, which search through text strings in search of patterned input, are also frequently implemented using an automaton called a *deterministic finite automaton*.

In this extended example, we will introduce two types of automata, *deterministic finite automata* and *nondeterministic finite automata*, then explore how to represent them in C++. We'll also explore how these automata can be used to simplify difficult string-matching problems.

Deterministic Finite Automata

Perhaps the simplest form of an automaton is a *deterministic finite automaton*, or DFA. At a high-level, a DFA is similar to a flowchart – it has a collection of *states* joined by various *transitions* that represent how the DFA should react to a given input. For example, consider the following DFA:



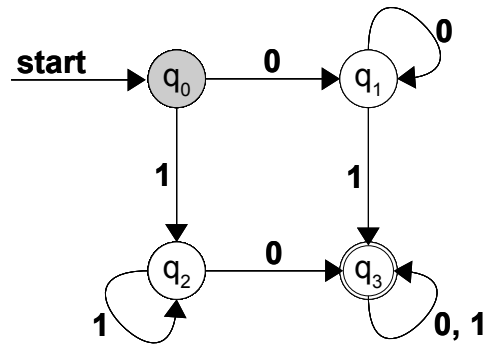
This DFA has four states, labeled q_0 , q_1 , q_2 , and q_3 , and a set of labeled transitions between those states. For example, the state q_0 has a transition labeled **0** to q_1 and a transition labeled **1** to q_2 . Some states have transitions to themselves; for example, q_2 transitions to itself on a **1**, while q_3 transitions to itself on either a **0** or **1**. Note that as shorthand, the transition labeled **0, 1** indicates two different transitions, one labeled with a **0** and one labeled with a **1**. The DFA has a designated state state, in this case q_0 , which is indicated by the arrow labeled **start**.

Notice that the state q_3 has two rings around it. This indicates that q_3 is an *accepting state*, which will have significance in a moment when we discuss how the DFA processes input.

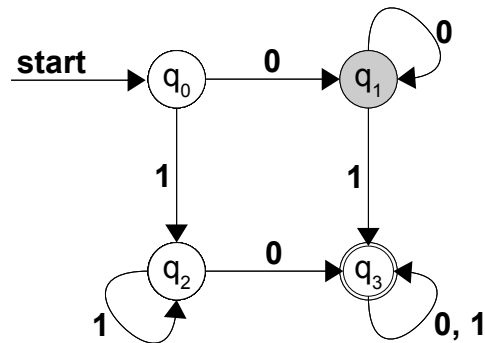
Since all of the transitions in this DFA are labeled either **0** or **1**, this DFA is said to have an *alphabet* of $\{0, 1\}$. A DFA can use any nonempty set of symbols as an alphabet; for example, the Latin or Greek alphabets are perfectly acceptable for use as alphabets in a DFA, as is the set of integers between 42 and 137. By definition, every state in a DFA is required to have a transition for each symbol in its alphabet. For example, in the above DFA, each state has exactly two transitions, one labeled with a **0** and the other with a **1**. Notice that state q_3 has only one transition explicitly drawn, but because the transition is labeled with two symbols we treat it as two different transitions.

The DFA is a simple computing machine that accepts as input a string of characters formed from its alphabet, processes the string, and then halts by either *accepting* the string or *rejecting* it. In essence, the DFA is a device for discriminating between two types of input – input for which some criterion is true and input for which it is false. The DFA starts in its designated start state, then processes its input character-by-character by transitioning from its current state to the state indicated by the transition. Once the DFA has finished consuming its input, it accepts the string if it ends in an accepting state; otherwise it rejects the input.

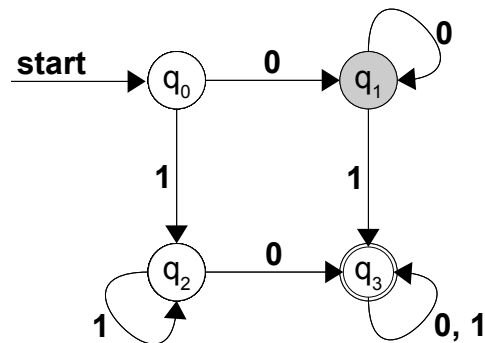
To see exactly how a DFA processes input, let us consider the above DFA simulated on the input **0011**. Initially, we begin in the start state, as shown here:



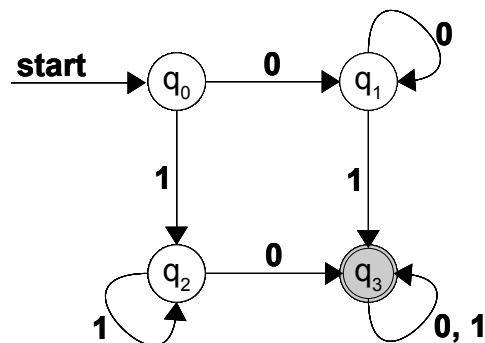
Since the first character of our string is a **0**, we follow the transition to state q_1 , as shown here:



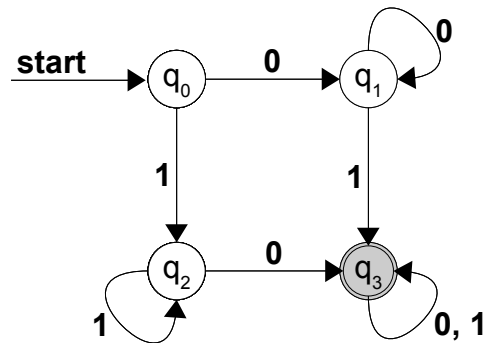
The second character of input is also a **0**, so we follow the transition labeled with a **0** and end up back in state q_1 , leaving us in this state:



Next, we consume the next input character, a **1**, which causes us to follow the transition labeled **1** to state q_3 :



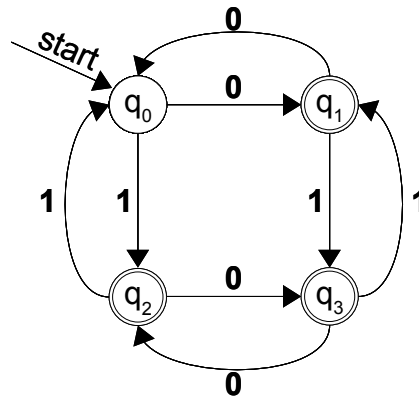
The final character of input is also a **1**, so we follow the transition labeled **1** and end back up in q_3 :



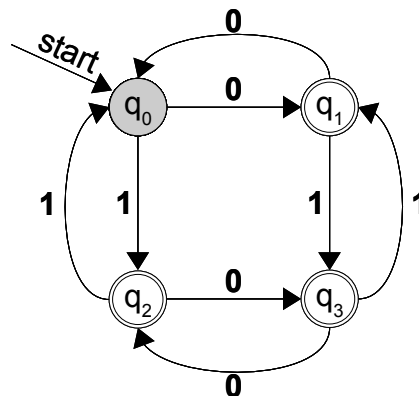
We are now done with our input, and since we have ended in an accepting state, the DFA accepts this input.

We can similarly consider the action of this DFA on the string **111**. Initially the machine will start in state q_0 , then transition to state q_2 on the first input. The next two inputs each cause the DFA to transition back to state q_2 , so once the input is exhausted the DFA ends in state q_2 , so the DFA rejects the input. We will not prove it here, but this DFA accepts all strings that have at least one **0** and at least one **1**.

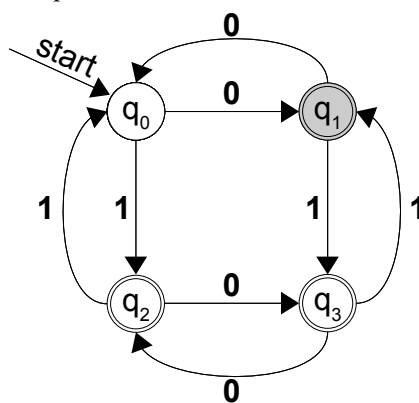
Two important details regarding DFAs deserve some mention. First, it is possible for a DFA to have multiple accepting states, as is the case in this DFA:



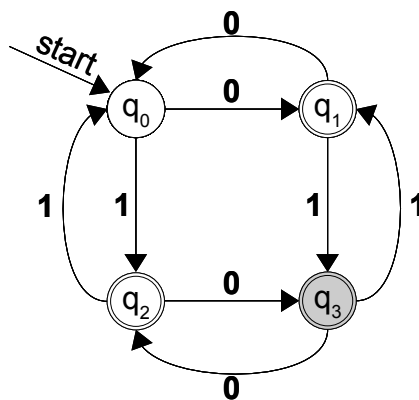
As with the previous DFA, this DFA has four states, but notice that three of them are marked as accepting. This leads into the second important detail regarding DFAs – the DFA only accepts its input if the DFA ends in an accepting state *when it runs out of input*. Simply transitioning into an accepting state does not cause the DFA to accept. For example, consider the effect of running this DFA on the input **0101**. We begin in the start state, as shown here:



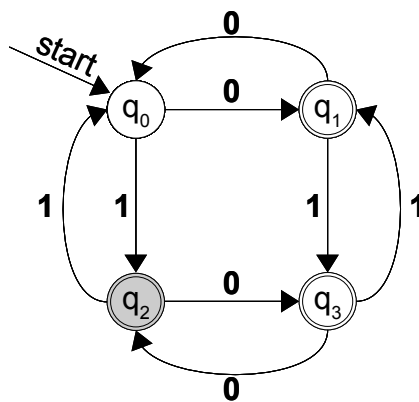
We first consume a **0**, sending us to state q_1 :



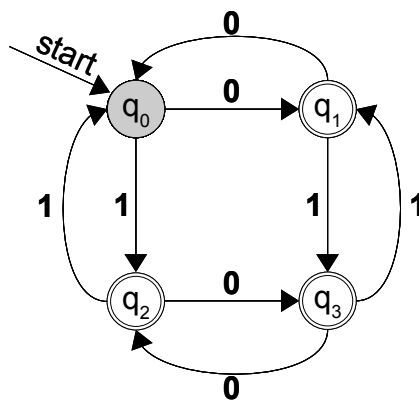
Next, we read a **1**, sending us to state q_3 :



The next input is a **0**, sending us to q_2 :



Finally, we read in a **1**, sending us back to q_0 :



Since we are out of input and are not in an accepting state, this DFA rejects its input, even though we transitioned through every single accepting state. If you want a fun challenge, convince yourself that this DFA accepts all strings that contain an odd number of 0s or an odd number of 1s (inclusive OR).

Representing a DFA

A DFA is a simple model of computation that can easily be implemented in software or hardware. For any DFA, we need to store five pieces of information:*

1. The set of states used by the DFA.
2. The DFA's alphabet.
3. The start state.
4. The state transitions.
5. The set of accepting states.

Of these five, the one that deserves the most attention is the fourth, the set of state transitions. Visually, we have displayed these transitions as arrows between circles in the graph. However, another way to treat state transitions is as a table with states along one axis and symbols of the alphabet along the other. For example, here is a transition table for the DFA described above:

State	0	1
q ₀	q ₁	q ₂
q ₁	q ₀	q ₃
q ₂	q ₃	q ₀
q ₃	q ₂	q ₁

To determine the state to transition to given a current state and an input symbol, we look up the row for the current state, then look at the state specified in the column for the current input.

If we want to implement a program that simulates a DFA, we can represent almost all of the necessary information simply by storing the transition table. The two axes encode all of the states and alphabet symbols, and the entries of the table represent the transitions. The information not stored in this table is the set of accepting states and the designated start state, so provided that we bundle this information with the table we have a full description of a DFA.

To concretely model a DFA using the STL, we must think of an optimal way to model the transition table. Since transitions are associated with pairs of states and symbols, one option would be to model the table as an STL map mapping a state-symbol pair to a new state. If we represent each symbol as a `char` and each state as an `int` (i.e. q₀ is 0, q₁ is 1, etc.), this leads to a state transition table stored as a `map<pair<int, char>, int>`. If we also track the set of accepting states as a `set<int>`, we can encode a DFA as follows:

```
struct DFA
{
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

* In formal literature, a DFA is often characterized as a quintuple (Q, Σ, q₀, δ, F) of the states, alphabet, start state, transition table, and set of accepting states, respectively. Take CS154 if you're interested in learning more about these wonderful automata, or CS143 if you're interested in their applications.

For the purposes of this example, assume that we have a function which fills this `DFA` struct with relevant data. Now, let's think about how we might go about simulating the DFA. To do this, we'll write a function `SimulateDFA` which accepts as input a `DFA` struct and a string representing the input, simulates the DFA when run on the given input, and then returns whether the input was accepted. We'll begin with the following:

```
bool SimulateDFA(DFA& d, string input)
{
    /* ... */
}
```

We need to maintain the state we're currently in, which we can do by storing it in an `int`. We'll initialize the current state to the starting state, as shown here:

```
bool SimulateDFA(DFA& d, string input)
{
    int currState = d.startState;
    /* ... */
}
```

Now, we need to iterate over the string, following transitions from state to state. Since the transition table is represented as a map from `pair<int, char>`s, we can look up the next state by using `make_pair` to construct a pair of the current state and the next input, then looking up its corresponding value in the map. As a simplifying assumption, we'll assume that the input string is composed only of characters from the DFA's alphabet.

This leads to the following code:

```
bool SimulateDFA(DFA& d, string input)
{
    int currState = d.startState;
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
        currState = d.transitions[make_pair(currState, *itr)];
    /* ... */
}
```

Once we've consumed all the input, we need to check whether we ended in an accepting state. We can do this by looking up whether the `currState` variable is contained in the `acceptingStates` set in the `DFA` struct, as shown here:

```
bool SimulateDFA(DFA& d, string input)
{
    int currState = d.startState;
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
        currState = d.transitions[make_pair(currState, *itr)];
    return d.acceptingStates.find(currState) != d.acceptingStates.end();
}
```

This function is remarkably simple but correctly simulates the DFA run on some input. As you'll see in the next section on applications of DFAs, the simplicity of this implementation lets us harness DFAs to solve a suite of problems surprisingly efficiently.

Applications of DFAs

The C++ `string` class exports a handful of searching functions (`find`, `find_first_of`, `find_last_not_of`, etc.) that are useful for locating specific strings or characters. However, it's surprisingly tricky to search strings for specific patterns of characters. The canonical example is searching for email addresses in a string of text. All email addresses have the same structure – a name field followed by an at sign (@)

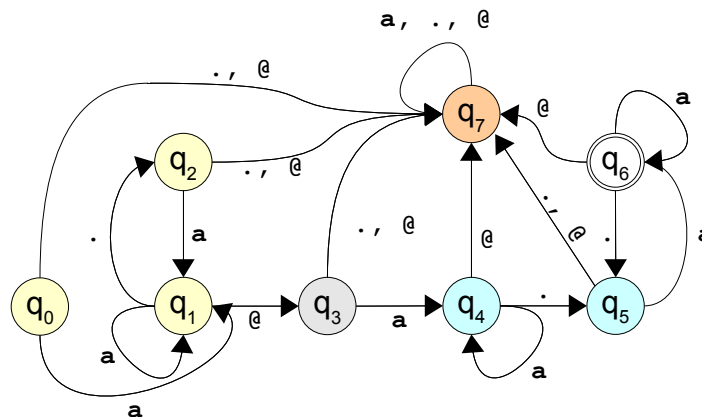
and a domain name. For example, `htiek@cs.stanford.edu` and `this.is.not.my.real.address@example.com` are valid email addresses. In general, we can specify the formatting of an email address as follows:*

- The name field, which consists of nonempty alphanumeric strings separated by periods. Periods can only occur between alphanumeric strings, never before or after. Thus `hello.world@example.com` and `cp-p.is.really.cool@example.com` are legal but `.oops@example.com`, `oops.@example.com`, and `oops..oops@example.com` are not.
- The host field, which is structured similarly to the above except that there must be at least two sequences separated by a dot.

Now, suppose that we want to determine whether a string is a valid email address. Using the searching functions exported by the `string` class this would be difficult, but the problem is easily solved using a DFA. In particular, we can design a DFA over a suitable alphabet that accepts a string if and only if the string has the above formatting.

The first question to consider is what alphabet this DFA should be over. While we could potentially have the DFA operate over the entire ASCII alphabet, it's easier if we instead group together related characters and use a simplified alphabet. For example, since email addresses don't distinguish between letters and numbers, we can have a single symbol in our alphabet that encodes any alphanumeric character. We would need to maintain the period and at-sign in the alphabet since they have semantic significance. Thus our alphabet will be $\{a, ., @\}$, where **a** represents alphanumeric characters, **.** is the period character, and **@** is an at-sign.

Given this alphabet, we can represent all email addresses using the following DFA:



This DFA is considerably trickier than the ones we've encountered previously, so let's take some time to go over what's happening here. The machine starts in state q_0 , which represents the beginning of input. Since all email addresses have to have a nonempty name field, this state represents the beginning of the first string in the name. The first character of an email address must be an alphanumeric character, which if read in state q_0 cause us to transition to state q_1 . States q_1 and q_2 check that the start of the input is something appropriately formed from alphanumeric characters separated by periods. Reading an alphanumeric character while in state q_1 keeps the machine there (this represents a continuation of the current word), and reading a dot transitions the machine to q_2 . In q_2 , reading anything other than an alphanumeric character puts the machine into the “trap state,” state q_7 , which represents that the input is invalid. Note that once the machine reaches state q_7 no input can get the machine out of that state and that q_7 isn't accepting. Thus any input that gets the machine into state q_7 will be rejected.

State q_3 represents the state of having read the at-sign in the email address. Here reading anything other than an alphanumeric character causes the machine to enter the trap state.

* This is a simplified version of the formatting of email addresses. For a full specification, refer to RFCs 5321 and 5322.

States q_4 and q_5 are designed to help catch the name of the destination server. Like q_1 , q_4 represents a state where we're reading a "word" of alphanumeric characters and q_5 is the state transitioned to on a dot. Finally, state q_6 represents the state where we've read at least one word followed by a dot, which is the accepting state. As an exercise, trace the action of this machine on the inputs `valid.address@email.com` and `invalid@not.com@ouch`.

Now, how can we use this DFA in code? Suppose that we have some way to populate a DFA struct with the information for this DFA. Then we could check if a string contains an email address by converting each character in the string into its appropriate character in the DFA alphabet, then simulating the DFA on the input. If the DFA rejects the input or the string contains an invalid character, we can signal that the string is invalid, but otherwise the string is a valid email address.

This can be implemented as follows:

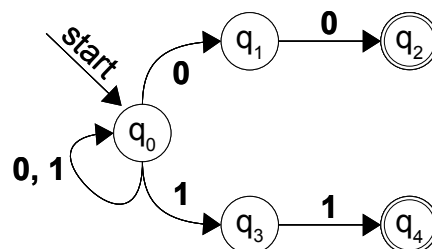
```
bool IsEmailAddress(string input)
{
    DFA emailChecker = LoadEmailDFA(); // Implemented elsewhere

    /* Transform the string one character at a time. */
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        /* isalnum is exported by <cctype> and checks if the input is an
         * alphanumeric character.
         */
        if(isalnum(*itr))
            *itr = 'a';
        /* If we don't have alphanumeric data, we have to be a dot or at-sign or
         * the input is invalid.
         */
        else if(*itr != '.' && *itr != '@')
            return false;
    }
    return SimulateDFA(emailChecker, input);
}
```

This code is remarkably concise, and provided that we have an implementation of `LoadEmailDFA` the function will work correctly. I've left out the implementation of `LoadEmailDFA` since it's somewhat tedious, but if you're determined to see that this actually works feel free to try your hand at implementing it.

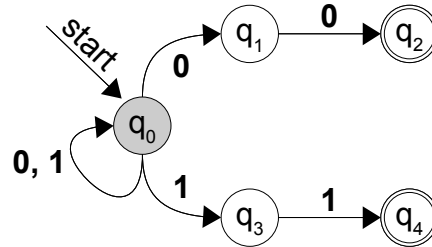
Nondeterministic Finite Automata

A generalization of the DFA is the *nondeterministic finite automaton*, or NFA. At a high level, DFAs and NFAs are quite similar – they both consist of a set of states connected by labeled transitions, of which some states are designated as accepting and others as rejecting. However, NFAs differ from DFAs in that a state in an NFA can have any number of transitions on a given input, including zero. For example, consider the following NFA:

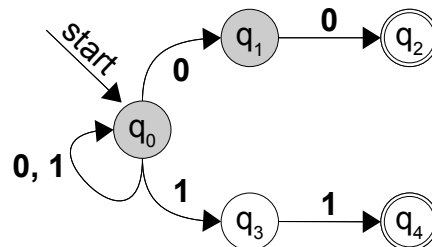


Here, the start state is q_0 and accepting states are q_2 and q_4 . Notice that the start state q_0 has two transitions on **0** – one to q_1 and one to itself – and two transitions on **1**. Also, note that q_3 has no defined transitions on **0**, and states q_2 and q_4 have no transitions at all.

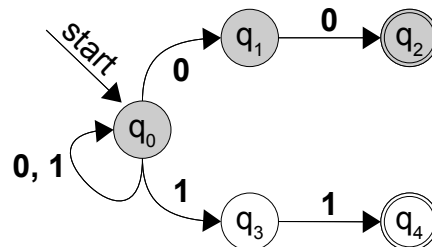
There are several ways to interpret a state having multiple transitions. The first is to view the automaton as choosing one of the paths nondeterministically (hence the name), then accepting the input if *some* set of choices results in the automaton ending in an accepting state. Another, more intuitive way for modeling multiple transitions is to view the NFA as being in several different states simultaneously, at each step following every transition with the appropriate label in each of its current states. To see this, let's consider what happens when we run the above NFA on the input **0011**. As with a DFA, we begin in the start state, as shown here:



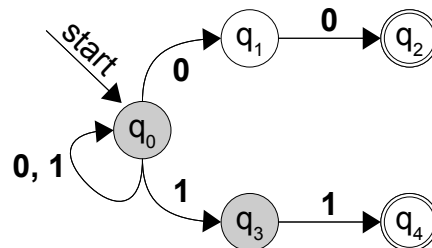
We now process the first character of input (**0**) and find that there are two transitions to follow – the first to q_0 and the second to q_1 . The NFA thus ends up in both of these states simultaneously, as shown here:



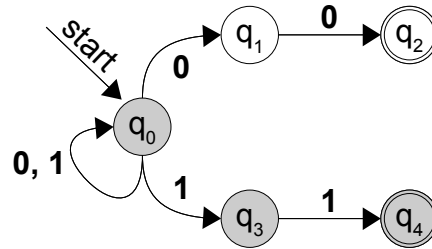
Next, we process the second character (**0**). From state q_0 , we transition into q_0 and q_1 , and from state q_1 we transition into q_2 . We thus end up in states q_0 , q_1 , and q_2 , as shown here:



We now process the third character of input, which is a **1**. From state q_0 we transition to states q_0 and q_3 . We are also currently in states q_1 and q_2 , but neither of these states has a transition on a **1**. When this happens, we simply drop the states from the set of current states. Consequently, we end up in states q_0 and q_3 , leaving us in the following configuration:



Finally, we process the last character, a **1**. State q_0 transitions to q_0 and q_1 , and state q_1 transitions to state q_4 . We thus end up in this final configuration:



Since the NFA ends in a configuration where at least one of the active states is an accepting state (q_4), the NFA accepts this input. Again as an exercise, you might want to convince yourself that this NFA accepts all and only the strings that end in either **00** or **11**.

Implementing an NFA

Recall from above the definition of the `DFA` struct:

```
struct DFA
{
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

Here, the transition table was encoded as a `map<pair<int, char>, int>` since for every combination of a state and an alphabet symbol there was exactly one transition. To generalize this to represent an NFA, we need to be able to associate an arbitrary number of possible transitions. This is an ideal spot for an STL `multimap`, which allows for duplicate key/value pairs. This leaves us with the following definition for an NFA type:

```
struct NFA
{
    multimap<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

How would we go about simulating this NFA? At any given time, we need to track the set of states that we are currently in, and on each input need to transition from the current set of states to some other set of states. A natural representation of the current set of states is (hopefully unsurprisingly) as a `set<int>`. Initially, we start with this set of states just containing the start state. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    /* ... */
}
```

Next, we need to iterate over the string we've received as input, following transitions where appropriate. This at least requires a simple `for` loop, which we'll write here:

```

bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        /* ... */
    }

    /* ... */
}

```

Now, for each character of input in the string, we need to compute the set of next states (if any) to which we should transition. To simplify the implementation of this function, we'll create a second `set<int>` corresponding to the next set of states the machine will be in. This eliminates problems caused by adding elements to our set of states as we're iterating over the set and updating it. We thus have

```

bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        set<int> nextStates;
        /* ... */
    }

    /* ... */
}

```

Now that we have space to put the next set of machine states, we need to figure out what states to transition to. Since we may be in multiple different states, we need to iterate over the set of current states, computing which states they transition into. This is shown here:

```

bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state)
        {
            /* ... */
        }
    }

    /* ... */
}

```

Given the state being iterated over by `state` and the current input character, we now want to transition to each state indicated by the `multimap` stored in the `NFA` struct. If you'll recall, the STL `multimap` exports a function called `equal_range` which returns a pair of iterators into the `multimap` that delineate the range of elements with the specified key. This function is exactly what we need to determine the set of new states we'll be entering for each given state – we simply query the `multimap` for all elements whose key is the pair of the specified state and the current input, then add all of the destination states to our next set of states. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state)
        {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second field
                 * is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    /* ... */
}
```

Finally, once we've consumed all input, we need to check whether the set of states contains any states that are also in the set of accepting states. We can do this by simply iterating over the set of current states, then checking if any of them are in the accepting set. This is shown here and completes the implementation of the function:

```

bool SimulateNFA(NFA& nfa, string input)
{
    /* Track our set of states. We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
    {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state)
        {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second field
                 * is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    for(set<int>::iterator itr = currStates.begin(); itr != currStates.end(); ++itr)
        if(nfa.acceptingStates.count(*itr)) return true;
    return false;
}

```

Compare this function to the implementation of the DFA simulation. There is substantially more code here, since we have to track multiple different states rather than just a single state. However, this extra complexity is counterbalanced by the simplicity of designing NFAs compared to DFAs. Building a DFA to match a given pattern can be much trickier than building an equivalent NFA because it's difficult to model “guessing” behavior with a DFA. However, both functions are a useful addition to your programming arsenal, so it's good to see how they're implemented.

More to Explore

This extended example introduced the DFA and NFA and demonstrated how to implement them, but didn't consider some of the more interesting algorithms that operate on them. If you're interested in an algorithmic and implementation challenge, consider looking into the following:

- **Subset Construction:** DFAs and NFAs are equivalently powerful, meaning that for every NFA that is an DFA that accepts and rejects the same inputs and vice-versa. The proof of this fact hinges on the *subset construction*, a way of transforming an NFA into an equivalent DFA. The construction is remarkably simple and elegant and would be a great way to play around with the STL.
- **DFA Minimization:** Multiple DFAs can accept and reject precisely the same sets of input. As an extreme case, any DFA where no state is an accepting state is equivalent to a one-state DFA that does not have any accepting states, since neither can possibly accept any inputs. Interestingly, for every DFA, there is a *unique* DFA with the fewest number of states that is equal to that DFA. There is a rather straightforward algorithm for eliminating redundant states in a DFA that is an excellent way to practice your STL skills; consult Wikipedia or a text on automata theory (*Introduction to Automata Theory, Languages, and Computation, Third Edition* by Hopcroft, Motwani, and Ullman is an excellent place to look).

Chapter 9: STL Algorithms

The STL algorithms are an amazing collection of template functions that work on ranges defined by iterators. They encompass a wide scope of functionality, enabling you to leverage heavily off preexisting code. While STL algorithms do not introduce any new functionality to your code, they nonetheless provide incredible speed bonuses during design and at runtime. However, STL algorithms can be complicated and require some time to adjusting to. This chapter discusses the basic syntax for STL algorithms, some of their programming conventions, and the more common algorithms you'll encounter in practice.

A Word on Compiler Errors

More so than with any other portion of the STL, when working with STL algorithms the compiler errors you will encounter can be completely incomprehensible. When you pass invalid parameters to an STL algorithm, the compiler may report errors in the code for the STL algorithms rather than in the code you wrote. If this happens, search the errors carefully and look for messages like “during template instantiation of” or “while instantiating.” These phrases identify the spot where the compiler choked on your code. Read over your code carefully and make sure that your parameters are valid. Did you provide an output iterator instead of an input iterator? Does your comparison function accept the right parameter types? Deciphering these types of errors is a skill you can only learn with time.

A Word on Header Files

STL algorithms are in either the `<algorithm>` or `<numeric>` header files. Algorithms from `<numeric>` tend to be based more on computational programming, while algorithms from `<algorithm>` tend to be more general-purpose. It's common to see both headers at the top of professional code, so don't be confused by the `<numeric>`. If you receive compiler errors about undefined functions, make sure you've included both these headers.

Basic STL Algorithms

The best way to understand STL algorithms is to see them in action. For example, the following code snippet prints the sum of all of the elements in a `set<int>`:

```
cout << accumulate(mySet.begin(), mySet.end(), 0) << endl;
```

In that tiny line of code, we're iterating over the entire `set` and summing the values together. This exemplifies STL algorithms – performing huge tasks in a tiny space.

The `accumulate` function, defined in the `<numeric>` header, takes three parameters – two iterators that define a range of elements and an initial value to use in the summation – and returns the sum of the elements in that range plus the base value.* Notice that nothing about this function requires that the elements be in a `set` – you could just as well pass in iterators from a `vector`, or even `istream_iterator<int>`s. Also note that there's no requirement that the iterators define the range of an entire container. For example, here's code to print the sum of all the elements in a `set<int>` between 42 and 137, inclusive:

```
cout << accumulate(mySet.lower_bound(42),  
                  mySet.upper_bound(137), 0) << endl;
```

* There is also a version of `accumulate` that accepts four parameters, as you'll see in the chapter on functors.

Behind the scenes, `accumulate` is implemented as a template function that accepts two iterators and simply uses a loop to sum together the values. Here's one possible implementation of `accumulate`:

```
template <typename InputIterator, typename Type> inline
Type accumulate(InputIterator start, InputIterator stop, Type initial)
{
    while(start != stop)
    {
        initial += *start;
        ++start;
    }
    return initial;
}
```

While some of the syntax specifics might be a bit confusing (notably the template header and the `inline` keyword), you can still see that the heart of the code is just a standard iterator loop that continues advancing the start iterator forward until it reaches the destination. There's nothing magic about `accumulate`, and the fact that the function call is a single line of code doesn't change the fact that it still uses a loop to sum all the values together.

If STL algorithms are just functions that use loops behind the scenes, why even bother with them? There are several reasons, the first of which is *simplicity*. With STL algorithms, you can leverage off of code that's already been written for you rather than reinventing the code from scratch. This can be a great time-saver and also leads into the second reason, *correctness*. If you had to rewrite all the algorithms from scratch every time you needed to use them, odds are that at some point you'd slip up and make a mistake. You might, for example, write a sorting routine that accidentally uses `<` when you meant `>` and consequently does not work at all. Not so with the STL algorithms – they've been thoroughly tested and will work correctly for any given input. The third reason to use algorithms is *speed*. In general, you can assume that if there's an STL algorithm that performs a task, it's going to be faster than most code you could write by hand. Through advanced techniques like template specialization and template metaprogramming, STL algorithms are transparently optimized to work as fast as possible. Finally, STL algorithms offer *clarity*. With algorithms, you can immediately tell that a call to `accumulate` adds up numbers in a range. With a for loop that sums up values, you'd have to read each line in the loop before you understood what the code did.

Algorithm Naming Conventions

There are over fifty STL algorithms and memorizing them all would be a chore, to say the least. Fortunately, many of them have common naming conventions so you can recognize algorithms even if you've never encountered them before.

The suffix `_if` on an algorithm (`replace_if`, `count_if`, etc.) means the algorithm will perform a task on elements only if they meet a certain criterion. Functions ending in `_if` require you to pass in a predicate function that accepts an element and returns a `bool` indicating whether the element matches the criterion. For example, the STL `count` algorithm accepts a range of iterators and a value, then returns the number of times that the value appears in that range. `count_if`, on the other hand, accepts a range of iterators and a predicate and returns the number of times the predicate evaluates to `true` in that range.

Algorithms containing the word `copy` (`remove_copy`, `partial_sort_copy`, etc.) will perform some task on a range of data and store the result in the location pointed at by an extra iterator parameter. With `copy` functions, you'll specify all the normal data for the algorithm plus an extra iterator specifying a destination for the result. We'll cover what this means from a practical standpoint later.

If an algorithm ends in `_n` (`fill_n`, `generate_n`, etc), then it will perform a certain operation `n` times. These functions are useful for cases where the number of times you perform an operation is meaningful, rather than the range over which you perform it.

Reordering Algorithms

There are a large assortment of STL algorithms at your disposal, so for this chapter it's useful to discuss the different algorithms in terms of their basic functionality. The first major grouping of algorithms we'll talk about are the *reordering algorithms*, algorithms that reorder but preserve the elements in a container.

Perhaps the most useful of the reordering algorithms is `sort`, which sorts elements in a range in ascending order. For example, the following code will sort a `vector<int>` from lowest to highest:

```
sort(myVector.begin(), myVector.end());
```

`sort` requires that the iterators you pass in be random-access iterators, so you cannot use `sort` to sort a `map` or `set`. However, since `map` and `set` are always stored in sorted order, this shouldn't be a problem.

By default, `sort` uses the `<` operator for whatever element types it's sorting, but you can specify a different comparison function if you wish. Whenever you write a comparison function for an STL algorithm, it should accept two parameters representing the elements to compare and return a `bool` indicating whether the first element is strictly less than the second element. In other words, your callback should mimic the `<` operator. Note that this is different than the comparison functions you've seen in CS106B/X, which return an `int` specifying the relation between the two elements.

For example, suppose we had a `vector<placeT>`, where `placeT` was defined as

```
struct placeT
{
    int x;
    int y;
};
```

Then we could sort the vector only if we wrote a comparison function for `placeTs`.^{*} For example:

```
bool ComparePlaces(placeT one, placeT two)
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

```
sort(myPlaceVector.begin(), myPlaceVector.end(), ComparePlaces);
```

You can also use custom comparison functions even if a default already exists. For example, here is some code that sorts a `vector<string>` by length, ignoring whether the strings are in alphabetical order:

```
bool CompareStringLength(string one, string two)
{
    return one.length() < two.length();
}
sort(myVector.begin(), myVector.end(), CompareStringLength);
```

^{*} When we cover operator overloading in the second half of this text, you'll see how to create functions that `sort` will use automatically.

One last note on comparison functions is that they should either accept the parameters by value or by “reference to const.” Since we haven’t covered `const` yet, for now your comparison functions should accept their parameters by value. Otherwise you can get some pretty ferocious compiler errors.

Another useful reordering function is `random_shuffle`, which randomly scrambles the elements of a container.* Because the scrambling is random, there’s no need to pass in a comparison function. Here’s some code that uses `random_shuffle` to scramble a `vector`’s elements:

```
random_shuffle(myVector.begin(), myVector.end());
```

As with `sort`, the iterators must be random-access iterators, so you can’t scramble a `set` or `map`. Then again, since they’re sorted containers, you shouldn’t want to do this in the first place.

The last major algorithm in this category is `rotate`, which cycles the elements in a container. For example, given the input container (0, 1, 2, 3, 4, 5), rotating the container around position 3 would result in the container (2, 3, 4, 5, 0, 1). The syntax for `rotate` is anomalous in that it accepts three iterators delineating the range and the new front, but in the order *begin*, *middle*, *end*. For example, to rotate a `vector` around its third position, we would write

```
rotate(v.begin(), v.begin() + 2, v.end());
```

Searching Algorithms

Commonly you’re interested in checking membership in a container. For example, given a `vector`, you might want to know whether or not it contains a specific element. While the `map` and `set` naturally support `find`, `vectors` and `deque`s lack this functionality. Fortunately, you can use STL algorithms to correct this problem.

To search for an element in a container, you can use the `find` function. `find` accepts two iterators delineating a range and a value, then returns an iterator to the first element in the range with that value. If nothing in the range matches, `find` returns the second iterator as a sentinel. For example:

```
if(find(myVector.begin(), myVector.end(), 137) != myVector.end())
    /* ... vector contains 137 ... */
```

Although you can legally pass `map` and `set` iterators as parameters to `find`, you should avoid doing so. If a container class has a member function with the same name as an STL algorithm, you should use the member function instead of the algorithm because member functions can use information about the container’s internal data representation to work much more quickly. Algorithms, however, must work for all iterators and thus can’t make any optimizations. As an example, with a `set` containing one million elements, the `set`’s `find` member function can locate elements in around twenty steps using binary search, while the STL `find` function could take up to one million steps to linearly iterate over the entire container. That’s a staggering difference and really should hit home how important it is to use member functions over STL algorithms.

Just as a sorted `map` and `set` can use binary search to outperform the linear STL `find` algorithm, if you have a sorted linear container (for example, a sorted `vector`), you can use the STL algorithm `binary_search` to perform the search in a fraction of the time. For example:

```
/* Assume myVector is sorted. */
if(binary_search(myVector.begin(), myVector.end(), 137))
    { /* ... Found 137 ... */ }
```

* Internally, `random_shuffle` calls the C library function `rand` to reorder the elements. Thus you should seed the randomizer using `srand` before calling `random_shuffle`.

Also, as with `sort`, if the container is sorted using a special comparison function, you can pass that function in as a parameter to `binary_search`. However, make sure you're consistent about what comparison function you use, because if you mix them up `binary_search` might not work correctly.

Note that `binary_search` doesn't return an iterator to the element – it simply checks to see if it's in the container. If you want to do a binary search in order to get an iterator to an element, you can use the `lower_bound` algorithm which, like the `map` and `set` `lower_bound` functions, returns an iterator to the first element greater than or equal to the specified value. Note that `lower_bound` might hand back an iterator to a different element than the one you searched for if the element isn't in the range, so be sure to check the return value before using it. As with `binary_search`, the container must be in sorted order for `lower_bound` algorithm to work correctly.

Set Algorithms

If you'll recall, the `set` container class does not support intersection, union, or subset. However, the STL algorithms provide a useful set of functions that can perform these operations. Unlike the CS106B/X `Set` functions, the STL algorithms will work on any sorted container type, not just a `set`, so you can intersect two `vectors`, a `set` and a `deque`, or even two `maps`.

Set algorithms need a place to store the result of the set operations, which you specify by providing an iterator to the start of the range that will hold the new values. Be careful, though, because this destination must have enough space to store the result of the operation or the algorithm will write out of bounds. Unfortunately, though, this forms a sort of circular dependency – you need to have enough space available to store the result of the algorithm before invoking the algorithm, but you can't know how much space you need until you've run the algorithm! To break this cycle, you can use iterator adapters in conjunction with the set algorithms. If you want to take the union of two sets, rather than preallocating enough space for the result, just specify an `insert_iterator` as the destination iterator and the container storing the result of the operation will automatically grow to contain all of the resulting elements.

There are four major set algorithms: `set_union`, which computes the union of two sets; `set_intersection`, which computes the intersection; `set_difference`, which creates a set of all elements in the first container except for those in the second container, and `set_symmetric_difference`, the set of all elements in either the first or second container but not both. Below is some code showcasing `set_union`, although any of the above four functions can be substituted in:

```
set<int> result;
set_union(setOne.begin(), setOne.end(),           // All of the elements in setOne
          setTwo.begin(), setTwo.end(),           // All of the elements in setTwo
          inserter(result, result.begin())); // Store in result.
```

Two other useful algorithms are `includes` and `equal`. `includes` accepts two sorted iterator ranges and returns whether all elements in the second range are contained in the first range; this is equivalent to testing whether one range is a subset of another. The `equal` algorithm takes in three iterators – one range of iterators and an iterator to the beginning of a different range – and returns whether the ranges are equal. Be careful when using `equal` – the algorithm expects the ranges to have the same number of elements and will read garbage data if there isn't enough room in the second sequence!

Here's code demonstrating `includes` and `equals`:

```
if(includes(setOne.begin(), setOne.end(), setTwo.begin(), setTwo.end()))
    /* ... setTwo is a subset of setOne ... */

if(setOne.size() == setTwo.size() &&
   equal(setOne.begin(), setOne.end(), setTwo.begin()))
    /* ... setOne == setTwo ... */
```

Removal Algorithms

The STL provides several algorithms for removing elements from containers. However, removal algorithms have some idiosyncrasies that can take some time to adjust to.

Despite their name, removal algorithms **do not** actually remove elements from containers. This is somewhat counterintuitive but makes sense when you think about how algorithms work. Algorithms accept *iterators*, not *containers*, and thus do not know how to erase elements from containers. Removal functions work by shuffling down the contents of the container to overwrite all elements that need to be erased. Once finished, they return iterators to the first element not in the modified range. So for example, if you have a `vector` initialized to 0, 1, 2, 3, 3, 3, 4 and then `remove` all instances of the number 3, the resulting `vector` will contain 0, 1, 2, 4, 3, 3, 4 and the function will return an iterator to one spot past the first 4. If you'll notice, the elements in the iterator range starting at `begin` and ending with the element one past the four are the sequence 0, 1, 2, 4 – exactly the range we wanted.

To truly remove elements from a container with the removal algorithms, you can use the container class member function `erase` to erase the range of values that aren't in the result. For example, here's a code snippet that removes all copies of the number 137 from a `vector`:

```
myVector.erase(remove(myVector.begin(), myVector.end(), 137), myVector.end());
```

Note that we're erasing elements in the range `[*, end)`, where `*` is the value returned by the `remove` algorithm.

There is another useful removal function, `remove_if`, that removes all elements from a container that satisfy a condition specified as the final parameter. For example, using the `ispunct` function from the header file `<cctype>`, we can write a `StripPunctuation` function that returns a copy of a string with all the punctuation removed:*

```
string StripPunctuation(string input)
{
    input.erase(remove_if(input.begin(), input.end(), ispunct), input.end());
    return input;
}
```

(Isn't it amazing how much you can do with a single line of code? That's the real beauty of STL algorithms.)

If you're shaky about how to actually remove elements in a container using `remove`, you might want to consider the `remove_copy` and `remove_copy_if` algorithms. These algorithms act just like `remove` and `remove_if`, except that instead of modifying the original range of elements, they copy the elements that aren't removed into another container. While this can be a bit less memory efficient, in some cases it's exactly what you're looking for.

Miscellaneous Algorithms

There are a number of other useful algorithms, of which three, `copy`, `for_each` and `transform`, we'll cover in this section.

`copy` accepts three parameters – a range of iterators delineating an input and an iterator specifying a destination – then copies the elements from the input range into the destination. As with the set algorithms, the destination iterator must point to the start of a range capable of holding all of the specified elements. For example,

* On some compilers, this code will not compile as written. See the later section on compatibility issues for more information.

here's a code snippet showing how to overwrite the beginning of a `vector` with elements in a `set` that are greater than or equal to 137:

```
copy(mySet.lower_bound(137), mySet.end(), myVector.begin());
```

`for_each` accepts as input an input range and a callback function, then calls the function on each element in the range. Note that `for_each` doesn't modify the values of the elements in the range; rather it simply mimics an iterator loop that calls a function for each element in the range. Currently, `for_each` might not seem all that useful, especially since the function to call must take exactly one parameter, but when we cover functors in the second half of this book `for_each` will become an invaluable weapon in your C++ arsenal.

Here is some code that iterates over a `vector<int>` and prints out each element:

```
void PrintSingleInt(int value)
{
    cout << value << endl;
}
```

```
for_each(myVector.begin(), myVector.end(), PrintSingleInt);
```

Another useful function is `transform`, which applies a function to a range of elements and stores the result in the specified destination. `transform` accepts four parameters – two iterators delineating an input range, an output iterator specifying a destination, and a callback function, then stores in the output destination the result of applying the function to each element in the input range. As with the set algorithms, `transform` assumes that there is sufficient storage space in the range pointed at by the destination iterator, so make sure that you have sufficient space before transforming a range.

`transform` is particularly elegant when combined with functors, but even without them is useful for a whole range of tasks. For example, consider the `tolower` function, a C library function declared in the header `<cctype>` that accepts a `char` and returns the lowercase representation of that character. Combined with `transform`, this lets us write `ConvertToLowerCase` from `strutils.h` in two lines of code, one of which is a return statement.*

```
string ConvertToLowerCase(string text)
{
    transform(text.begin(), text.end(), text.begin(), tolower);
    return text;
}
```

Note that after specifying the range `text.begin(), text.end()` we have another call to `text.begin()`. This is because we need to provide an iterator that tells `transform` where to put its output. Since we want to overwrite the old contents of our container with the new values, we specify `text.begin()` another time to indicate that `transform` should start writing elements to the beginning of the string as it generates them.

There is no requirement that the function you pass to `transform` return elements of the same type as those stored in the container. It's legal to transform a set of strings into a set of doubles, for example.

The following table lists some of the more common STL algorithms. It's by no means an exhaustive list, and you should consult a reference to get a complete list of all the algorithms available to you.

* Again, see the later section on compatibility issues if you're getting compiler errors when writing this code.

Type accumulate(InputItr start, InputItr stop, Type value)	Returns the sum of the elements in the range [start, stop) plus the value of value.
bool binary_search(RandomItr start, RandomItr stop, const Type& value)	Performs binary search on the sorted range specified by [start, stop) and returns whether it finds the element value. If the elements are sorted using a special comparison function, you must specify the function as the final parameter.
OutItr copy(InputItr start, InputItr stop, OutItr outputStart)	Copies the elements in the range [start, stop) into the output range starting at outputStart. copy returns an iterator to one past the end of the range written to.
size_t count(InputItr start, InputItr end, const Type& value)	Returns the number of elements in the range [start, stop) equal to value.
size_t count_if(InputItr start, InputItr end, PredicateFunction fn)	Returns the number of elements in the range [start, stop) for which fn returns true. Useful for determining how many elements have a certain property.
bool equal(InputItr start1, InputItr stop1, InputItr start2)	Returns whether elements contained in the range defined by [start1, stop1) and the range beginning with start2 are equal. If you have a special comparison function to compare two elements, you can specify it as the final parameter.
pair<RandomItr, RandomItr> equal_range(RandomItr start, RandomItr stop, const Type& value)	Returns two iterators as a pair that defines the sub-range of elements in the sorted range [start, stop) that are equal to value. In other words, every element in the range defined by the returned iterators is equal to value. You can specify a special comparison function as a final parameter.
void fill(ForwardItr start, ForwardItr stop, const Type& value)	Sets every element in the range [start, stop) to value.
void fill_n(ForwardItr start, size_t num, const Type& value)	Sets the first num elements, starting at start, to value.
InputItr find(InputItr start, InputItr stop, const Type& value)	Returns an iterator to the first element in [start, stop) that is equal to value, or stop if the value isn't found. The range doesn't need to be sorted.
InputItr find_if(InputItr start, InputItr stop, PredicateFunc fn)	Returns an iterator to the first element in [start, stop) for which fn is true, or stop otherwise.
Function for_each(InputItr start, InputItr stop, Function fn)	Calls the function fn on each element in the range [start, stop).
void generate(ForwardItr start, ForwardItr stop, Generator fn);	Calls the zero-parameter function fn once for each element in the range [start, stop), storing the return values in the range.
void generate_n(OutputItr start, size_t n, Generator fn);	Calls the zero-parameter function fn n times, storing the results in the range beginning with start.
bool includes(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2)	Returns whether every element in the sorted range [start2, stop2) is also in [start1, stop1). If you need to use a special comparison function, you can specify it as the final parameter.

<pre>bool lexicographical_compare(InputItr s1, InputItr s2, InputItr t1, InputItr t2)</pre>	Returns whether the range of elements defined by [s1, s2) is lexicographically less than [t1, t2); that is, if the first range precedes the second in a “dictionary ordering.”
<pre>InputItr lower_bound(InputItr start, InputItr stop, const Type& elem)</pre>	Returns an iterator to the first element greater than or equal to the element elem in the sorted range [start, stop). If you need to use a special comparison function, you can specify it as the final parameter.
<pre>InputItr max_element(InputItr start, InputItr stop)</pre>	Returns an iterator to the largest value in the range [start, stop). If you need to use a special comparison function, you can specify it as the final parameter.
<pre>InputItr min_element(InputItr start, InputItr stop)</pre>	Returns an iterator to the smallest value in the range [start, stop). If you need to use a special comparison function, you can specify it as the final parameter.
<pre>bool next_permutation(BidirItr start, BidirItr stop)</pre>	Given a range of elements [start, stop), modifies the range to contain the next lexicographically higher permutation of those elements. The function then returns whether such a permutation could be found. It is common to use this algorithm in a do ... while loop to iterate over all permutations of a range of data, as shown here: <pre>sort(range.begin(), range.end()); do { /* ... process ... */ } while(next_permutation(range.begin(), range.end()));</pre>
<pre>bool prev_permutation(BidirItr start, BidirItr stop)</pre>	Given a range of elements [start, stop), modifies the range to contain the next lexicographically lower permutation of those elements. The function then returns whether such a permutation could be found.
<pre>void random_shuffle(RandomItr start, RandomItr stop)</pre>	Randomly reorders the elements in the range [start, stop).
<pre>ForwardItr remove(ForwardItr start, ForwardItr stop, const Type& value)</pre>	Removes all elements in the range [start, stop) that are equal to value. This function will not remove elements from a container. To shrink the container, use the container's erase function to erase all values in the range [retValue, end()), where retValue is the return value of remove.
<pre>ForwardItr remove_if(ForwardItr start, ForwardItr stop, PredicateFunc fn)</pre>	Removes all elements in the range [start, stop) for which fn returns true. See remove for information about how to actually remove elements from the container.
<pre>void replace(ForwardItr start, ForwardItr stop, const Type& toReplace, const Type& replaceWith)</pre>	Replaces all values in the range [start, stop) that are equal to toReplace with replaceWith.
<pre>void replace_if(ForwardItr start, ForwardItr stop, PredicateFunction fn, const Type& with)</pre>	Replaces all elements in the range [start, stop) for which fn returns true with the value with.
<pre>ForwardItr rotate(ForwardItr start, ForwardItr middle, ForwardItr stop)</pre>	Rotates the elements of the container such that the sequence [middle, stop) is at the front and the range [start, middle) goes from the new middle to the end. rotate returns an iterator to the new position of start.

ForwardItr search(ForwardItr start1, ForwardItr stop1, ForwardItr start2, ForwardItr stop2)	Returns whether the sequence [start2, stop2) is a subsequence of the range [start1, stop1). To compare elements by a special comparison function, specify it as a final parameter.
InputItr set_difference(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutputItr dest)	Stores all elements that are in the sorted range [start1, stop1) but not in the sorted range [start2, stop2) in the destination pointed to by dest. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
InputItr set_intersection(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutputItr dest)	Stores all elements that are in both the sorted range [start1, stop1) and the sorted range [start2, stop2) in the destination pointed to by dest. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
InputItr set_union(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutputItr dest)	Stores all elements that are in either the sorted range [start1, stop1) or in the sorted range [start2, stop2) in the destination pointed to by dest. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
InputItr set_symmetric_difference(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutputItr dest)	Stores all elements that are in the sorted range [start1, stop1) or in the sorted range [start2, stop2), but not both, in the destination pointed to by dest. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
void swap(Value& one, Value& two)	Swaps the values of one and two.
ForwardItr swap_ranges(ForwardItr start1, ForwardItr stop1, ForwardItr start2)	Swaps each element in the range [start1, stop1) with the corresponding elements in the range starting with start2.
OutputItr transform(InputItr start, InputItr stop, OutputItr dest, Function fn)	Applies the function fn to all of the elements in the range [start, stop) and stores the result in the range beginning with dest. The return value is an iterator one past the end of the last value written.
RandomItr upper_bound(RandomItr start, RandomItr stop, const Type& val)	Returns an iterator to the first element in the sorted range [start, stop) that is strictly greater than the value val. If you need to specify a special comparison function, you can do so as the final parameter.

A Word on Compatibility

The STL is ISO-standardized along with the rest of C++. Ideally, this would mean that all STL implementations are uniform and that C++ code that works on one compiler should work on any other compiler. Unfortunately, this is not the case. No compilers on the market fully adhere to the standard, and almost universally compiler writers will make minor changes to the standard that decrease portability.

Consider, for example, the `ConvertToLowerCase` function from earlier in the section:

```
string ConvertToLowerCase(string text)
{
    transform(text.begin(), text.end(), text.begin(), tolower);
    return text;
}
```


This code will compile in Microsoft Visual Studio, but not in Xcode or the popular Linux compiler g++. The reason is that there are *two* `tolower` functions – the original C `tolower` function exported by `<cctype>` and a more modern `tolower` function exported by the `<locale>` header. Unfortunately, Xcode and g++ cannot differentiate between the two functions, so the call to `transform` will result in a compiler error. To fix the problem, you must explicitly tell C++ which version of `tolower` you want to call as follows:

```
string ConvertToLowerCase(string text)
{
    transform(text.begin(), text.end(), text.begin(), ::tolower);
    return text;
}
```

Here, the strange-looking `::` syntax is the *scope-resolution operator* and tells C++ that the `tolower` function is the original C function rather than the one exported by the `<locale>` header. Thus, if you're using Xcode or g++ and want to use the functions from `<cctype>`, you'll need to add the `::`.

Another spot where compatibility issues can lead to trouble arises when using STL algorithms with the STL `set`. Consider the following code snippet, which uses `fill` to overwrite all of the elements in an STL `set` with the value 137:

```
fill(mySet.begin(), mySet.end(), 137);
```

This code will compile in Visual Studio, but will not under g++. Recall from the second chapter on STL containers that manipulating the contents of an STL `set` in-place can destroy the set's internal ordering. Visual Studio's implementation of `set` will nonetheless let you modify `set` contents, even in situations like the above where doing so is unsafe. g++, however, uses an STL implementation that treats all `set` iterators as read-only. Consequently, this code won't compile, and in fact will cause some particularly nasty compiler errors.

When porting C++ code from one compiler to another, you might end up with inexplicable compiler errors. If you find some interesting C++ code online that doesn't work on your compiler, it doesn't necessarily mean that the code is invalid; rather, you might have an overly strict compiler or the online code might use an overly lenient one.

More to Explore

While this chapter lists some of the more common algorithms, there are many others that are useful in a variety of contexts. Additionally, there are some useful C/C++ library functions that work well with algorithms. If you're interested in maximizing your algorithmic firepower, consider looking into some of these topics:

1. **<cctype>**: This chapter briefly mentioned the `<cctype>` header, the C runtime library's character type library. `<cctype>` include support for categorizing characters (for example, `isalpha` to return if a character is a letter and `isxdigit` to return if a character is a valid hexadecimal digit) and formatting conversions (`toupper` and `tolower`).
2. **<cmath>**: The C mathematics library has all sorts of nifty functions that perform arithmetic operations like `sin`, `sqrt`, and `exp`. Consider looking into these functions if you want to use `transform` on your containers.
3. **Boost Algorithms**: As with most of the C++ Standard Library, the Boost C++ Libraries have a whole host of useful STL algorithms ready for you to use. One of the more useful Boost algorithm sets is the string algorithms, which extend the functionality of the `find` and `replace` algorithms on strings from dealing with single characters to dealing with entire strings.

Practice Problems

Algorithms are ideally suited for solving a wide variety of problems in a small space. Most of the following programming problems have short solutions – see if you can whittle down the space and let the algorithms do the work for you!

1. Write a function `PrintVector` that accepts a `vector<int>` as a parameter and prints its contents separated by space characters. (*Hint: This can be accomplished in one line – use an `ostream_iterator` and `copy`. You will want to see the solution to this problem.*) ♦
2. Using `remove_if` and a custom callback function, write a function `RemoveShortWords` that accepts a `vector<string>` and removes all strings of length 3 or less from it.
3. In n -dimensional space, the distance from a point $(x_1, x_2, x_3, \dots, x_n)$ to the origin is $\sqrt{x_1^2 + x_2^2 + x_3^2 + \dots + x_n^2}$. Write a function `DistanceToOrigin` that accepts a `vector<double>` representing a point in space and returns the distance from that point to the origin. Do not use any loops – let the algorithms do the heavy lifting for you. (*Hint: Use the `transform` algorithm and a custom callback function to square all of the elements of the vector.*)
4. Write a function `BiasedSort` that accepts a `vector<string>` by reference and sorts the vector lexicographically, except that if the vector contains the string “Me First,” that string is always at the front of the sorted list. ♦
5. Write a function `CriticsPick` that accepts a `map<string, double>` of movies and their ratings (between 0.0 and 10.0) and returns a `set<string>` of the names of the top ten movies in the map. If there are fewer than ten elements in the map, then the resulting set should contain every string in the map. (*Hint: Remember that all elements in a `map<string, double>` are stored internally as `pair<string, double>`*)
6. Implement the count algorithm for `vector<int>`s. Your function should have the prototype `int count(vector<int>::iterator start, vector<int>::iterator stop, int element)` and should return the number of elements in the range `[start, stop)` that are equal to `element`. ♦
7. Using the `generate_n` algorithm, the `rand` function, and a `back_insert_iterator`, show how to populate a vector with a specified number of random values. Then use `accumulate` to compute the average of the range.
8. Show how to use a combination of `copy`, `istreambuf_iterator`, and `ostreambuf_iterator` to open a file and print its contents to `cout`.

9. A *monoalphabetic substitution cipher* is a simple form of encryption. We begin with the letters of the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

We then scramble these letters randomly, yielding a new ordering of the alphabet. One possibility is as follows:

K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

This new ordering thus defines a mapping from each letter in the alphabet to some other letter in the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

To encrypt a source string, we simply replace each character in the string with its corresponding encrypted character. For example, the string “The cookies are in the fridge” would be encoded as follows:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
G	Y	J	D	H	H	F	N	J	O	K	I	J	N	R	G	Y	J	W	I	N	Q	A	J

Monoalphabetic substitution ciphers are surprisingly easy to break – in fact, most daily newspapers include a daily puzzle that involves deciphering a monoalphabetic substitution cipher – but they are still useful for low-level encryption tasks such as posting spoilers to websites (where viewing the spoiler explicitly requires the reader to decrypt the text).

Using the `random_shuffle` algorithm, implement a function `MonoalphabeticSubstitutionEncrypt` that accepts a source string and encrypts it with a random monoalphabetic substitution cipher.

Chapter 10: Extended Example: Palindromes

A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a tag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a tat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal – Panama!

– Dan Hoey [Pic96]

It is fitting to conclude our whirlwind tour of the STL with an example showcasing exactly how concise and powerful well-written STL code can be. This example is shorter than the others in this book, but should nonetheless illustrate how the different library pieces all fit together. Once you've finished reading this chapter, you should have a solid understanding of how the STL and streams libraries can come together beautifully to elegantly solve a problem.

Palindromes

A *palindrome* is a word or phrase that is the same when read forwards or backwards, such as “racecar” or “Malayalam.” It is customary to ignore spaces, punctuation, and capitalization when reading palindromes, so the phrase “Mr. Owl ate my metal worm” would count as a palindrome, as would “Go hang a salami! I'm a lasagna hog.”

Suppose that we want to write a function `IsPalindrome` that accepts a `string` and returns whether or not the string is a palindrome. Initially, we'll assume that spaces, punctuation, and capitalization are all significant in the string, so “Party trap” would not be considered a palindrome, though “Part y traP” would. Don't worry – we'll loosen this restriction in a bit. Now, we want to verify that the string is the same when read forwards and backwards. There are many possible ways to do this. Prior to learning the STL, we might have written this function as follows:

```
bool IsPalindrome(string input)
{
    for(int k = 0; k < input.size() / 2; ++k)
        if(input[k] != input[input.length() - 1 - k])
            return false;
    return true;
}
```

That is, we simply iterate over the first half of the string checking to see if each character is equal to its respective character on the other half of the string. There's nothing wrong with the approach, but it feels too *mechanical*. The high-level operation we're modeling asks whether the first half of the string is the same forwards as the second half is backwards. The code we've written accomplishes this task, but has to explicitly walk over the characters from start to finish, manually checking each pair. Using the STL, we can accomplish the same result as above without explicitly spelling out the details of how to check each character.

There are several ways we can harness the STL to solve this problem. For example, we could use the STL `reverse` algorithm to create a copy of the string in reverse order, then check if the string is equal to its reverse. This is shown here:

```
bool IsPalindrome(string input)
{
    string reversed = input;
    reverse(input.begin(), input.end());
    return reversed == input;
}
```

This approach works, but requires us to create a copy of the string and is therefore less efficient than our original implementation. Can we somehow emulate the functionality of the initial `for` loop using iterators? The answer is yes, thanks to `reverse_iterators`. Recall that every STL container class exports a pair of functions called `rbegin` and `rend` returning `reverse_iterators`, iterators that start at the end of the container and end at the beginning. This lets us iterate over the string contents backwards. Also recall that the STL `equal` algorithm accepts three inputs – two iterators delineating a range and a third iterator indicating the start of a second range – then returns whether the two ranges are equal. Combined with `reverse_iterators`, this yields the following *one-line implementation* of `IsPalindrome`:

```
bool IsPalindrome(string input)
{
    return equal(input.begin(), input.begin() + input.size() / 2, input.rbegin());
}
```

This is a remarkably simple approach that is identical to what we've written earlier but much less verbose. Of course, it doesn't correctly handle capitalization, spaces, or punctuation, but we can take care of that with only a few more lines of code. Let's begin by stripping out everything from the string except for alphabetic characters. For this task, we can use the STL `remove_if` algorithm, which accepts as input a range of iterators and a predicate, then modifies the range by removing all elements for which the predicate returns true. Like its partner algorithm `remove`, `remove_if` doesn't actually remove the elements from the sequence (see the last chapter for more details), so we'll need to `erase` the remaining elements afterwards.

Because we want to eliminate all characters from the string that are not alphabetic, we need to create a predicate function that accepts a character and returns whether it is not a letter. The header file `<cctype>` exports a helpful function called `isalpha` that returns whether a character *is* a letter. This is the opposite what we want, so we'll create our own function which returns the negation of `isalpha`.*

* When we cover the `<functional>` library in the second half of this book, you'll see a simpler way to do this.

```
bool IsNotAlpha(char ch)
{
    return !isalpha(ch);
}
```

We can now strip out nonalphabetic characters from our input string as follows:

```
bool IsPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha), input.end());
    return equal(input.begin(), input.begin() + input.size() / 2, input.rbegin());
}
```

Finally, we need to make sure that the string is treated case-insensitively, so inputs like “RACEcar” are accepted as palindromes. Using the code developed in the chapter on algorithms, we can convert the string to uppercase after stripping out everything except characters, yielding this final version of `IsPalindrome`:

```
bool IsPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha), input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    return equal(input.begin(), input.begin() + input.size() / 2, input.rbegin());
}
```

This function is remarkable in its elegance and terseness. In *three lines of code* we've stripped out all of the characters in a string that aren't letters, converted what's left to upper case, and returned whether the string is the same forwards and backwards. This is the STL in action, and I hope that you're beginning to appreciate the power of the techniques you've learned over the past few chapters.

Before concluding this example, let's consider a variant on a palindrome where we check whether the *words* in a phrase are the same forwards and backwards. For example, “Did mom pop? Mom did!” is a palindrome both with respect to its letters and its words, while “This is this” is a phrase that is not a palindrome but is a word-palindrome. As with regular palindromes, we'll ignore spaces and punctuation, so “It's an its” counts as a word-palindrome even though it uses two different forms of the word *its/it's*. The machinery we've developed above works well for entire strings; can we modify it to work on a word-by-word basis?

In some aspects this new problem is similar to the original. We still to ignore spaces, punctuation, and capitalization, but now need to treat words rather than letters as meaningful units. There are many possible algorithms for checking this property, but one solution stands out as particularly good. The idea is as follows:

1. Clean up the input: strip out everything except letters *and spaces*, then convert the result to upper case.
2. Break up the input into a list of words.
3. Return whether the list is the same forwards and backwards.

In the first step, it's important that we preserve the spaces in the original input so that we don't lose track of word boundaries. For example, we would convert the string “Hello? Hello!? HELLO?” into “HELLO HELLO HELLO” instead of “HELLOHELLOHELLO” so that we can recover the individual words in the second step. Using a combination of the `isalpha` and `isspace` functions from `<cctype>` and the convert-to-upper-case code used above, we can preprocess the input as shown here:

```

bool IsNotAlphaOrSpace(char ch)
{
    return !isalpha(ch) && !isspace(ch);
}

bool IsWordPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    /* ... */
}

```

At this point the string `input` consists of whitespace-delimited strings of uniform capitalization. We now need to tokenize the input into individual words. This would be tricky were it not for `stringstream`. Recall that when reading a string out of a stream using the stream extraction operator (`>>`), the stream treats whitespace as a delimiter. Thus if we funnel our string into a `stringstream` and then read back individual strings, we'll end up with a tokenized version of the input. Since we'll be dealing with an arbitrarily-long list of strings, we'll store the resulting list in a `vector<string>`, as shown here:

```

bool IsWordPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    /* ... */
}

```

Now, what is the easiest way to read strings out of the stream until no strings remain? We could do this manually, as shown here:

```

bool IsWordPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    while(true)
    {
        string token;
        tokenizer >> token;

        if(tokenizer.fail()) break;
        tokens.push_back(token);
    }
}

```

This code is correct, but it's bulky and unsightly. The problem is that it's just too *mechanical*. We want to insert all of the tokens from the `stringstream` into the `vector`, but as written it's not clear that this is what's happening. Fortunately, there is a much, *much* easier way to solve this problem thanks to `istream_iterator`. Recall that `istream_iterator` is an iterator adapter that lets you iterate over an input stream as if it were a range of

data. Using `istream_iterator` to wrap the stream operations and the vector's `insert` function to insert a range of data, we can rewrite this entire loop in one line as follows:

```
bool IsWordPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                 input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(), // Insert at beginning...
                  istream_iterator<string>(tokenizer), // Reading from tokenizer...
                  istream_iterator<string>());          // To the end of the stream.
}
```

Recall that two `istream_iterator`s are necessary to define a range, and that an `istream_iterator` constructed with no arguments is a special “end of stream” iterator. This one line of code replaces the entire loop from the previous implementation, and provided that you have some familiarity with the STL this second version is also easier to read.

The last step in this process is to check if the sequence of strings is the same forwards and backwards. But we already know how to do this – we just use `equal` and a `reverse_iterator`. Even though the original implementation applied this technique to a `string`, we can use the same pattern here on a `vector<string>` because all the container classes are designed with a similar interface. Remarkable, isn't it?

The final version of `IsWordPalindrome` is shown here:

```
bool IsWordPalindrome(string input)
{
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                 input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(), // Insert at the beginning of the vector
                  istream_iterator<string>(tokenizer), // Reading from tokenizer
                  istream_iterator<string>());
    return equal(tokens.begin(), tokens.begin() + tokens.size() / 2,
                 tokens.rbegin());
}
```

Concluding Remarks

This quick interlude hopefully gives you a taste for what's possible with the STL. The skills you've developed over the past chapters will follow you through the rest of your C++ programming career and armed with only the knowledge you've acquired so far you can tackle a wide array of programming problems. But this is just the beginning of C++ and it's now time to change gears and see what else C++ can offer us. Don't worry – although the second half of this book focuses more on language features than libraries, we will periodically revisit the STL to give you a feel for what this library can do when backed by the full support of the C++ language.

Part Two

C++ Core Language Features

C++ is a general purpose programming language with a bias towards systems programming that

- *is a better C.*
- *supports data abstraction.*
- *supports object-oriented programming.*
- *supports generic programming*

– Bjarne Stroustrup, inventor of C++ [Str09.2]

C++ has a rich set of core language features that makes it among the most expressive programming languages in modern use. As a systems programming language, C++ gives you direct access to the computer's memory, allowing you to fine-tune your code to take maximum advantage of available hardware. As an object-oriented language, C++ supports a number of inheritance schemata and has a wide array of tools and annotations you can use to define classes. As a generic programming language, C++ affords an enormous degree of flexibility in generic class design and has one of the most powerful template systems of any programming language.

In this section, we will explore C++'s key language features with the goal of understanding how to use C++ both as a tool for solving problems and as a language for precisely and naturally expressing the pieces of that problem. Ideally, by the time you have finished reading over this section you will have an intuitive understanding for how C++ is put together and will be able to use your newfound language skills to make your programs efficient, robust, and maintainable.

Before we begin discussing the specifics of any one language features, let us take a few minutes to go over some pieces of C++ design philosophy that will inform our later discussion.

The Zero-Overhead Principle

One of C++'s major design goals is the *zero-overhead principle*, that C++ code should only pay for the language features it actually uses. In other words, if you compile any C++ program, the resulting executable should be as efficient as if the C++ language consists solely of language features used in your program.

The zero-overhead principle has important implications for C++ programs. First, because you only pay runtime costs for language features you actually use, you should feel free to program in the style that you feel most comfortable with. Provided you understand the costs of the language features you use, you can get a precise sense for how efficiently your programs will perform. You might not know what virtual private multiple inheritance is, but provided you don't use it you don't need to worry about it making your implementation of a maze-solving algorithm run less efficiently. Second, the zero-overhead principle means that it's perfectly acceptable to learn only a subset of C++'s language features. C++ is an enormous language and even veteran C++ programmers learn new tricks from time to time,* but because you don't pay for what you don't use you shouldn't lose any sleep over potential holes in your C++ knowledge base.

* One of the great joys of teaching CS106L and writing up this course reader has been discovering new possibilities within the C++ programming language. You're not the only one still learning C++!

Of course, the zero-overhead principle means that C++ can look strange or verbose. At times you'll need to explicitly indicate that you want to use a certain language feature whereas other languages would have this as the default. For example, member functions in C++ by default are not polymorphic (see the chapter on inheritance for more information) because polymorphic function calls are expensive, whereas in Java *all* member functions (methods) are polymorphic unless indicated otherwise. As you read through these next few chapters, keep the zero-overhead principle in mind and see how it influences C++'s design.

Compile-Time Cost vs. Runtime Cost

C++ places a premium on runtime efficiency and tries as much as possible to shift difficult computations from runtime to compile-time. C++ is designed such that complex operations such as field accesses, multiple inheritance, and `virtual` functions can be greatly optimized at compile-time, which in part is why C++ is one of the most efficient general-purpose programming languages. However, this philosophy at times makes C++ tricky to work with because the compiler has to be able to infer substantial information from the source code. As you'll see, this means that certain language features can seem at times pedantic or counterintuitive. Of course, this extra work means that your programs run much, much faster than equivalent programs written in other languages, but be prepared to offer sacrifices on the altar of compiler appeasement.

C Compatibility

C++ inherited much of its core syntax and semantics from the C programming language, a language designed for operating system development. Historically, C++ has tried to maximize compatibility with C. At times this can be extremely useful, since libraries written in pure C can almost always be directly integrated into C++ projects. Moreover, C programmers who want to use some of the simpler features of C++ can do so without spending too much time learning all of C++.

But C compatibility is also the source of much criticism of C++. C++ allows for many inherently unsafe conversions (for example, conversions between pointer and integer types) and gives programmers direct and at times dangerous access to low-level memory. Automatic copying of `structs` in C translates into potentially dangerous default copy behavior for C++ `classes` and `structs`, and C's representation of character strings is a persistent source of programmer error and security vulnerabilities.

As you read over the next few chapters, be aware that many of the quirks you'll notice about C++ stem from this sort of backwards compatibility.

Prerequisites

The first few chapters in this section (Pointers and References, C Strings, the Preprocessor) can be read with no prior background. The rest of this section, however, expects a familiarity with classes. In particular, you should know what the following are, when they should be used, and, where applicable, how to implement them:

- The `class` keyword.
- The `public` and `private` access specifiers.
- Constructors and destructors.
- Member functions.

If you are unfamiliar with any of these terms, refer to *Programming Abstractions in C++* by Eric Roberts and Julie Zelenski for an introduction.

Chapter 11: Pointers and References

C++ supports special primitive data types called *pointers* and *references* that alias other variables or memory locations. With pointers and references, you can access other variables indirectly or refer to blocks of memory generated at runtime.

Used correctly, pointers and references can perform wonders. As you'll see in CS106B/X, pointers are crucial in data structures like trees, linked lists, and graphs. Pointers are also used extensively in systems programming, since they provide a way to write values to specific memory regions. However, pointer power comes at a high price, and it would not be much of a stretch to claim that almost all program crashes are caused by pointer and reference errors.

This chapter acts as a quick introduction to pointers and references. You will probably want to refer to the CS106B/X course reader for more information.

What is a Reference?

As you have learned in CS106B/X, C++ has two parameter-passing mechanisms. The first, *pass-by-value*, passes parameters by initializing the parameter as a copy of the argument. For example, consider the following function:

```
void DoSomething(int x)
{
    x++; // x is passed by value, so no changes to outside world occur.
}
```

Here, when we call the `DoSomething` function, the parameter `x` is initialized as a copy of the argument to the function, so the line `x++` will not affect the value of any variables outside the function. That is, given the following code:

```
int x = 137;
cout << x << endl; // Prints 137
DoSomething(x);
cout << x << endl; // Prints 137
```

Both the first and second `cout` statements will print the value 137.

The second version of parameter-passing in C++ is *pass-by-reference*, where arguments to the function can be modified by the function. As an example, the following function takes its parameter by reference, so local changes propagate to the caller:

```
void MutateParameter(int& x)
{
    x++; // x will be updated in the calling function
}
```

Using `MutateParameter` in the following snippet will cause the first `cout` statement to print 137, but the second to print 138:

```
int x = 137;
cout << x << endl; // Prints 137
MutateParameter(x);
cout << x << endl; // Prints 138
```

Notice that the syntax for passing `x` by reference was to declare `x` as a `int& x` rather than simply `int x`. Interestingly, you can use this syntax in other places in your C++ programs to declare variables called *references* which, like parameters that are passed by reference, transparently modify other program variables.

To see how references work, consider the following code snippet:

```
int myVariable = 137;
int& myReference = myVariable; // myReference is a reference to myVariable
```

In this code snippet, we initially declare a variable of type `int` called `myVariable`, then initialize it to 137. In the next line, we create a variable `myReference` of type `int&` and initialize it to the variable `myVariable`. From this point on, `myReference` acts a reference to `myVariable` and like a reference parameter in a function, any changes to `myReference` will update the value of `myVariable`. For example, consider the following code:

```
cout << myVariable << endl; // Prints 137
cout << myReference << endl; // Prints 137, the value of myVariable
myReference = 42; // Indirectly updates myVariable
cout << myVariable << endl; // Prints 42
cout << myReference << endl; // Prints 42
```

Here, we print out the values of `myVariable` and `myReference`, which are the same because `myReference` is a reference to `myVariable`. Next, we assign `myReference` the value 42, and because `myReference` is a reference, it updates the value of `myVariable`, as shown with the final two couts.

It is perfectly legal to have several references to the same variable, as shown here:

```
int myInt = 42;
int& ref1 = myInt, &ref2 = myInt; // ref1, ref2 now reference myInt
```

Notice that when declaring several references on the same line, it is necessary to prefix each of them with an ampersand. This is a quirk that helps make the syntax more consistent with that of the C programming language. Forgetting this rule is the source of many a debugging nightmare.

Because references are aliases for other variables, there are several restrictions on references that are not true of other types. First, references must be initialized when they are declared. For example, the following code is illegal:

```
int& myRef; // Error: need to initialize the reference!
```

This restriction at times can be a bit vexing, but in the long run will save you a good deal of trouble. Because references always must be initialized, when working with references you can be more certain that the variable they're referring to actually exists.

Second, because references are aliases, you cannot initialize a reference using a constant value or with the return value of a function. For example, both of the following are illegal:

```
int& myRef1 = 137; // Error: Can't initialize a reference to a constant.
int& myRef2 = GetInteger(); // Error: Can't initialize to a return value.
```

The reason for these restrictions is simple. Suppose that we could legally initialize `myRef` to 137. What would happen if we wrote the following?

```
myRef = 42;
```

Since `myRef` is a reference, when we change the value of `myRef`, we are actually changing the value of some other variable. But if `myRef` is initialized to 137, then this code would be equivalent to writing

```
137 = 42;
```

Which is clearly nonsensical.* Similarly, we cannot initialize `myRef` to the return value of a function, since otherwise we could try to write code to the effect of

```
GetInteger() = 42;
```

Which is illegal.

Technically speaking, a reference can only be initialized with what is known as an *lvalue*, something that can legally be put on the left-hand side of an assignment statement. For example, any variable is an lvalue, since variables can be assigned to. Constant literals like 137, on the other hand, are not lvalues because we cannot put a constant on the left-hand side of an assignment statement.

One final point about references is that once a reference has been initialized, it is impossible to change where that reference refers to. That is, once a reference has been bound to a value, we cannot “rebind” the reference to some other object. For example, consider the following code:

```
int myInt = 137, myOtherInt = 137;
int& myRef = myInt, &myOtherRef = myOtherInt;
myRef = myOtherRef;
```

In the last line, we assign `myRef` the value of `myOtherRef`. This does *not* cause `myRef` to begin referring to the same variable as `myOtherRef`; instead, it takes the value of the variable referenced by `myOtherRef` and stores it in the variable referenced by `myRef`.

Pointers

Put simply, a *pointer* is a variable that stores a data type and a memory address. For example, a pointer might encode that an `int` is stored at memory address 0x47D38B30, or that there is a `double` at 0x00034280. At a higher level, you can think of a pointer as a more powerful version of a reference. Like references, pointers allow you to indirectly refer to another variable or memory location. Unlike a reference, however, it is possible to change what object a pointer aliases. This versatility is both one of the most useful and one of the most vexing aspects of pointers, since you will have to take care to distinguish between the pointer and its *pointee*, the object at which it points.

To declare a pointer, we use the syntax `Type* variableName`, where `Type` is the type of variable the pointer will point to and `variableName` is the name of the newly-created variable. For example, to create a pointer to an `int`, you could write

```
int* myPointer;
```

* Interestingly, however, some older programming languages (notably FORTRAN) used to allow assignments like this, which would lead to all sorts of debugging headaches.

The whitespace around the star in this declaration is unimportant – the declarations `int * myPointer` and `int *myPointer` are also valid. Feel free to use whichever of these you feel is most intuitive, but be sure that you're able to read all three because each arises in professional code. In this course reader we will use the syntax `Type* ptr`.

Pointers need not point solely to primitive types. Here's code that creates a variable called `myStrPtr` that points to a C++ `string`:

```
string* myStrPtr;
```

As with references, when declaring multiple pointers in a single declaration, you will need to repeat the star for each variable. For example, consider the following code snippet:

```
int* myPtr1, myPtr2; // Legal, but incorrect.
```

Here, `myPtr1` is declared as an `int *` pointer to integer, but the variable `myPtr2`, despite its name, is just a plain old `int`. The star indicating a pointer only sticks onto the first variable it finds, so if you declare multiple pointers, you need to preface each with a star, as in

```
int* myPtr1, *myPtr2; // Legal and correct.
```

Initializing a Pointer

When you create a pointer using the above steps, you are simply declaring a variable. The pointer does not point to anything. Like all other primitive types, the pointer will contain garbage data and could be pointing anywhere in memory. Thus, before you attempt to work with a pointer, you must be sure to initialize it. Notice how pointers differ in this case from references – it is illegal to leave a reference uninitialized, but it is legal (but unsafe) to leave pointers uninitialized.

Initializing a pointer simply means assigning it the *address* (memory location) of the object you want it to point to. When this happens, the pointer is said to *point* to the variable at the address, called the *pointee*. While there are many ways to set up pointees, perhaps the simplest is to have the pointer point to a local variable declared on the stack. For example, consider this code snippet:

```
int myInteger = 137;
int* myIntPtr;
```

We'd like to have `myIntPtr` point to the variable `myInteger`. Now, if `myIntPtr` were a reference, we could simply write `myIntPtr = myInteger`. However, because `myIntPtr` is a pointer and not a reference, this code would be illegal. When working with pointers, to cause a pointer to point to a location, you must explicitly assign the pointer the address of the object to point to. That is, if we want `myIntPtr` to point to `myInteger`, we need to somehow get the address of the `myInteger` variable. To do this, we can use C++'s *address-of operator*, the `&` operator. For example, here is the code we could use to initialize `myIntPtr` to point to `myInteger`:

```
myIntPtr = &myInteger; // myIntPtr now points to myInteger.
```

It is an unfortunate choice of syntax that C++ uses `&` both to define a variable as a reference type and to get the address of a variable. The difference has to do with context – if `&` is used to declare a variable, it makes that variable a reference. Otherwise, `&` is used to take the address of a variable.

Here we see one major difference between pointers and references. When working with references, to initialize a reference to refer to a variable, we simply used a straight assignment statement. However, when using pointers, we have to explicitly take the address of the object we're pointing at. Why the difference? The main reason

is that when working with pointers, there is an explicit difference between the pointer variable and the object being pointed at. References are in some sense not “true” variables because using a reference always refers to the object the reference aliases rather than the reference itself. That is, if we have the following code:

```
int myInt;  
int& myRef = myInt;  
myRef++;
```

In the final line, `myRef++`, we are not incrementing `myRef`, but rather the object that it refers to.

Pointers, on the other hand, are true variables that can be reassigned, modified, and transformed without changing the value of the object that they point at. For example, given the following code:

```
int myInt1, myInt2;  
int* myPtr;  
myPtr = &myInt1;  
myPtr = &myInt2;
```

In the last two lines we first make `myPtr` point to `myInt1`, then reassign it to point to `myInt2`. Notice, however, that by saying `myPtr = &myInt1` and `myPtr = &myInt2`, we did **not** change the value of the object that `myPtr` points at. Instead, we simply changed what object the `myPtr` variable was pointing at. If we wanted to change the value of the object pointed at by `myPtr`, we would have to *dereference* it, as explained in a later section.

As with references, pointers are statically-typed and without explicitly subverting the type system can only be made to point to objects of a certain type. For example, the following code is illegal because it tries to make a `int*` point to a `double`:

```
double myDouble = 2.71828;  
int* myIntPtr = &myDouble; // Error: Can't store a double* in an int*
```

Another way to initialize a pointer is to have the pointer begin pointing to the same location as another pointer. For example, if we have a pointer named `myPtr` pointing to some location and we want to create a second pointer `myOtherPtr`, we can legally set `myOtherPtr` to point to the same location as `myPtr`. This is known as *pointer assignment* and, fortunately, has clean syntax. For example, here's code that makes two pointers each point to the same integer by using pointer assignment:

```
int myInteger = 137;  
int* myPtr, *myOtherPtr;  
myPtr = &myInteger; // Assign myPtr the address of myInteger  
myOtherPtr = myPtr; // myOtherPtr now points to the same object as myPtr
```

Note that when setting up `myPtr` to point to `myInteger`, we assigned it the value `&myInteger`, the address of the `myInteger` variable. However, when setting up `myOtherPtr`, we simply assigned it the value of `myPtr`. Had we written `myOtherPtr = &myPtr`, we would have gotten a compilation error, since `&myPtr` is the location of the `myPtr` variable, not the location of its pointee. In general, use straight assignment when assigning pointers to each other, and use the address-of operator when assigning pointers the addresses of other variables.

Dereferencing a Pointer

We now can initialize pointers, but how can we access the object being pointed at? This requires a *pointer dereference*, which follows a pointer to its destination and yields the object being pointed at.

To dereference a pointer, you preface the name of the pointer with a star, as in `*myIntPtr`. The dereferenced pointer then acts identically to the variable it's pointing to. For example, consider the following code snippet:

```
int myInteger = 137;
int* myPtr = &myInteger;

*myPtr = 42; // Dereference myPtr to get myInteger, then store 42.
cout << myInteger << endl; // Prints 42
```

In the first two lines, we create a variable called `myInteger` and a pointer `myPtr` that points to it. In the third line, `*myPtr = 42`, we dereference the pointer `myPtr` to get a reference to pointee, in this case `myInteger`, and assign it the value 42. In the final line we print the value 42, since we indirectly overwrote the contents of `myInteger` in the previous line.

Admittedly, the star notation with pointers can get a bit confusing since it means either “declare a pointer” or “dereference a pointer.” As with the `&` operator, with a little practice you'll be able to differentiate between the two.

Before you dereference a pointer, you must make sure that you've set it up correctly. Consider the following example:

```
int myInteger = 137;
int* myIntPtr; // Note: myIntPtr wasn't initialized!
*myIntPtr = 42;
cout << myInteger << endl;
```

This code is identical to the above example, except that we've forgotten to initialize `myIntPtr`. As a result, the line `*myIntPtr` will result in *undefined behavior*. When `myIntPtr` is created, like any other primitive type, it initially holds a garbage value. As a result, when we write `*myIntPtr = 42`, the program will almost certainly crash because `myIntPtr` points to an arbitrary memory address.

Almost all bugs that cause programs to crash stem from dereferencing invalid pointers. You will invariably run into this problem when working with C++ code, so remembering to initialize your pointers will greatly reduce the potential for error.

The -> Operator

When working with pointers to objects (i.e. `string*`), the pointer dereference operator sometimes can become a source of confusion. For example, consider the following program, which tries to create a pointer to a `string` and then return the length of the `string` being pointed at:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << *myPtr.length() << endl; // Error: See below.
```

This code at first might seem valid – we initialize `myPtr` to point to `myString`, and then try to print out the length of the object being pointed at. Unfortunately, this code is invalid because the `*` operator does not bind tightly enough. The problem is that C++ interprets `*myPtr.length()` as `*(myPtr.length())`; that is, dereferencing the expression `myPtr.length()`. This is a problem because `myPtr.length()` is illegal – `myPtr` is a pointer to a `string`, not a `string` itself, and we cannot apply the dot operator to it.

The proper version of the above code looks like this:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << (*myPtr).length() << endl; // Correct, but there's a better option
```

Here, we've put parentheses around the expression `*myPtr` so that C++ treats `(*myPtr).length()` correctly. But this code is bulky and difficult to read – it overly stresses that `myPtr` is a pointer rather than the fact that we're accessing the `length` member function. Fortunately, C++ has a special operator called the *arrow operator* that can be useful in these circumstances. Just as you can use the dot operator to access properties of regular objects, you can use the arrow operator to access properties of an object being pointed at. Here is another version of the above code, rewritten using the arrow operator:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << myPtr->length() << endl; // Use arrow to select length.
```

Pointer Comparison

Sometimes it is useful to compare two pointers to see if they are equal or unequal. To do so, you can use the `==` and `!=` operators, as shown here:

```
int myInt1, myInt2;
int* myPtr1 = &myInt1, *myPtr2 = &myInt2;

if(myPtr1 == myPtr2) // Check if the pointers point to the same location
    cout << "Pointers are equal" << endl;
if(myPtr1 != myPtr2) // Check if the pointers point to different locations
    cout << "Pointers are unequal" << endl;
```

Note that the `==` and `!=` operators check to see if the *pointers* are equal, not the *pointees*. Thus if two pointers point to different objects with the same value, `==` will return false. To see if the pointers point to objects with the same value, compare the values of the dereferenced pointers rather than the pointers themselves. For example:

```
if(*myPtr1 == *myPtr2)
    /* ... Pointers point to objects with the same value ... */
```

Null Pointers

In some cases, you may want to indicate that a pointer does not actually point to anything. For situations like this, you can assign pointers the special value `NULL`. Any pointer can be assigned the value `NULL`, no matter what the type of that pointer is. For example:

```
int* myIntPtr = NULL;
double* myDoublePtr = NULL;
```

Because `NULL` indicates that a pointer does not actually point at anything, dereferencing a `NULL` pointer leads to undefined behavior. In most cases, it crashes the program. Thus, before you dereference a pointer, make sure that you know it is non-`NULL`. For example, here's code to print out the value a pointer points to if the pointer is non-`NULL`, and an error otherwise:

```
if(myPtr == NULL) // Check if myPtr points to NULL
    cerr << "Pointer is NULL!" << endl;
else
    cout << "Value is: " << *myPtr << endl;
```

Unlike Java, in C++ pointers **do not** default to storing `NULL`. This means that if you forget to initialize a pointer, it does not necessarily point to `NULL` and in fact points to a random memory location. As a rule of thumb, if you declare a pointer but do not immediately initialize it, for safety reasons you should assign it the value `NULL` so that you can use `NULL`-checks later in your program.

Unlike other languages like Java or Visual Basic, the value `NULL` is not a C++ keyword. To use `NULL`, you need to include the header file `<cstdlib>`. However, since virtually every C++ standard library header references `<cstdlib>`, in most cases you can ignore the header file.

new and delete

Up to this point, we've only used pointers to refer to other variables declared on the stack. However, there's another place to store variables called the *heap*, a region of memory where variables can be created at runtime. While right now it might not be apparent why you would choose to allocate memory in the heap, as you'll see later in CS106B/X and CS106L, heap storage is critical in almost all nontrivial programs.

Heap storage gives you the ability to create and destroy variables as needed during program execution. If you need an integer to hold a value, or want to create a temporary `string` object, you can use the C++ `new` operator to obtain a pointer to one. The `new` operator sets aside a small block of memory to hold the new variable, then returns a pointer to the memory. For example, here's some code that allocates space for a `double` and stores it in a local pointer:

```
double* dynamicDouble = new double;
*dynamicDouble = 2.71828; // Write the value 2.71828
```

Note that because `new` yields a pointer you do not need to use the address-of operator to assign a pointer to memory created with `new`. In fact, it is illegal to do so, so code to this effect:

```
double* dynamicDouble = &(new double); // Error: Illegal to use & here.
```

will not compile.

Unlike other languages like Java, C++ does not have automatic garbage collection. As a result, if you allocate any memory with `new`, you must use the C++ `delete` keyword to reclaim the memory. Here's some code that allocates some memory, uses it a bit, then deletes it:

```
int* myIntPtr = new int;
*myIntPtr = 137;
cout << *myIntPtr << endl;
delete myIntPtr;
```

Note that when you write `delete myIntPtr`, you are *not* destroying the `myIntPtr` variable. Instead, you're instructing C++ to clean up the memory pointed at by `myIntPtr`. After writing `delete myIntPtr`, you're free to reassign `myIntPtr` to other memory and continue using it.

There are several important points to consider when using `delete`. First, calling `delete` on the same dynamically-allocated memory twice results in undefined behavior and commonly corrupts memory your program needs to function correctly. Thus you must be very careful to balance each call to `new` with one and *exactly* one call to `delete`. This can be tricky. For example, consider the following code snippet:

```
int *myIntPtr1 = new int;
int *myIntPtr2 = myIntPtr1;
delete myIntPtr1;
delete myIntPtr2;
```

At first you might think that this code is correct, since both `myIntPtr1` and `myIntPtr2` refer to dynamically-allocated memory, but unfortunately this code double-deletes the object. Since `myIntPtr1` and `myIntPtr2` refer to the same object, calling `delete` on both variables will try to clean up the same memory twice.

Second, after you call `delete` to clean up memory, accessing the reclaimed memory results in undefined behavior. In other words, calling `delete` indicates that you are done using the memory and do not plan on ever accessing it again. While this might seem simple, it can be quite complicated. For example, consider this code snippet:

```
int* myIntPtr1 = new int;
int* myIntPtr2 = myIntPtr1;
delete myIntPtr2;
*myIntPtr1 = 137;
```

Since `myIntPtr1` and `myIntPtr2` both refer to the same region in memory, after the call to `delete myIntPtr2`, both `myIntPtr1` and `myIntPtr2` point to reclaimed memory. However, in the next line we wrote `*myIntPtr1 = 137`, which tries to write a value to the memory address. This will almost certainly cause a runtime crash.

Finally, you should *only* delete memory that you explicitly allocated with `new`. That is, you should not delete a pointer to an object on a stack, nor should you try to delete a local variable. `delete` should only be used to balance out a call to `new` and nothing more. Using `delete` to clean up memory not allocated by `new` leads to undefined behavior that almost certainly will take down your entire program, so make sure you understand where to use it!

A Word on Undefined Behavior

I've used the term *undefined behavior* several times in this discussion of pointers to indicate potential sources of runtime crashes. Be sure that you don't equate undefined behavior and "runtime error;" undefined behavior is far more insidious. It would be wonderful if all undefined behavior caused a crash. If a program had a pointer error, it would be easy to diagnose and fix by just waiting for the crash to occur and then tracing back to the source of the problem. But this just isn't the case. Undefined behavior commonly manifests nondeterministically, crashing on some program runs and not on others. In fact, one of the telltale signs of undefined behavior is that the program works correctly most of the time, but periodically crashes unexpectedly.

For an extreme example of undefined behavior, consider the original DOS version of the classic game *SimCity*. Joel Spolsky, a software blogger, relates this story:

... [O]ne of the developers of the hit game SimCity... told me that there was a critical bug in his application: it used memory right after freeing it, a major no-no that *happened* to work OK on DOS but would not work under Windows where memory that is freed is likely to be snatched up by another running application right away. The testers on the Windows team were going through various popular applications, testing them to make sure they worked OK, but SimCity kept crashing. They reported this to the Windows developers, who disassembled SimCity, stepped through it in a debugger, found the bug, and *added special code* that checked if SimCity was running, and if it did, *ran the memory allocator in a special mode in which you could still use memory after freeing it*. [Spo08]

SimCity was using memory after freeing it, resulting in undefined behavior. On DOS-based systems where only a few applications would execute at any one time, "undefined behavior" coincidentally happened to work correctly. However, when switching to a new operating system like Windows 95 where multiple different processes each need access to memory, "undefined behavior" crashed the program and the only way to fix the problem was to rewrite Windows to explicitly avoid this problem.

Be very careful that you don't step into the realm of undefined behavior. Doing so can lead to bugs that manifest periodically yet can take down your entire program.

new[] and delete[]

Commonly, when writing programs to manipulate data, you'll need to allocate enough space to store an indeterminate number of variables. For example, suppose you want to write a program to play a variant on the game of checkers where the board can have any dimensions the user wishes. Without using the `Grid` ADT provided in the CS106B/X libraries, you would have a lot of trouble getting this program working, since you wouldn't know how much space the board would take up.

To resolve this problem, C++ has two special operators, `new[]` and `delete[]`, which allocate and deallocate blocks of memory holding multiple elements. For example, you could allocate space for 400 integers by writing `new int[400]`, or a sequence of characters twelve elements long with `new char[12]`.

The memory allocated with `new []` stores all the elements in sequential order, so if you know the starting address of the first variable, you can locate any variable in the sequence you wish. For example, if you have fourteen `ints` starting at address 1000, since `ints` are four bytes each, you can find the second integer in the list at position 1004, the third at 1008, etc. Therefore, although `new[]` allocates space for many variables, it returns only the address of the first variable. Thus you can store the list using a simple pointer, as shown below:

```
int* myManyInts = new int[137];
```

Once you have a pointer to a dynamically-allocated array of elements, you can access individual elements using square brackets `[]`. For example, here's code to allocate 200 integers and set each one equal to its position in the array:

```
const int NUM_ELEMS = 200;
int* myManyInts = new int[NUM_ELEMS];
for(int i = 0; i < NUM_ELEMS; ++i)
    myManyInts[i] = i;
```

As with regular `new`, you should clean up any memory you allocate with `new[]` by balancing it with a call to `delete[]`. You should not put any numbers inside the brackets of `delete []`, since the C++ memory manager is clever enough to keep track of how many elements to delete. For example, here's code to clean up a list of `ints`:

```
int *myInts = new int[100];
delete [] myInts;
```

Note that `delete` and `delete []` are different operators and cannot be substituted for one another. That is, you must be extremely careful not to clean up an array of elements using `delete`, nor a single element using `delete []`. Doing so will have disastrous consequences for your program and will almost certainly cause a crash.

Dynamic Arrays vs. STL `vector`/`deque`

You now know three ways to store a linear sequence of elements – the STL `vector` and `deque` containers and dynamically-managed arrays. Each behaves slightly differently, so when is it appropriate to use each? The answer is simple: prefer the STL `vector` and `deque` to dynamically-managed arrays whenever possible. The STL containers are inherently safer – they know their own size, they can grow automatically if you need to store additional data, and they are guaranteed not to leak memory. Contrast this with dynamically-allocated arrays. Dynamically-allocated arrays don't know how many elements they contain, and if you want to keep track of how much memory you've allocated you must explicitly track it yourself. Dynamic arrays also have a fixed capacity,

and once you've used up all of their storage space you must manually allocate additional storage space. Plus, you must remember to `delete []` the memory you allocate, or you will end up with a memory leak.

We have included dynamic arrays in our discussion of pointers for two reasons. First, though dynamic arrays are less safe than STL containers, many industrial C++ projects nonetheless use exposed dynamic arrays instead of the STL, either for compatibility reasons or to avoid some of the subtleties associated with the STL. Second, dynamic arrays are a powerful tool for implementing custom container classes. If you need to design a custom `vector` class or other specialized container, you may need to use dynamic array allocation to implement the container. But that said, you should be wary of dynamic array allocation and should opt to use the STL containers instead.

More to Explore

This introduction to pointers has been rather brief and does not address several important pointer topics. While the next chapter on C strings will cover some additional points, you should consider reading into some of these additional topics for more information on pointers. Also, please be sure to consult your course reader for CS106B/X.

1. **Automatic arrays:** C and C++ let you allocate arrays of elements on the stack as well as in the heap. Arrays are relatively unsafe compared to container classes like `vector`, but are a bit faster and arise in legacy code. Arrays are strongly related to pointers, and you should consider looking into them if you plan to use C++ more seriously.
2. **Smart pointers:** Pointers are difficult to work with – you need to make sure to `delete` or `delete []` memory once and exactly once, and must be on guard not to access deallocated memory. As a result, C++ programmers have developed objects called *smart pointers* that mimic standard pointers but handle all memory allocation and deallocation behind the scenes. Smart pointers are easy to use, reduce program complexity, and eliminate all sorts of errors. However, they do add a bit of overhead to your code. If you're interested in seriously using C++, be sure to look into smart pointers.
3. **`nullptr`:** C++ is an constantly-growing language and a new revision of C++, called “C++0x,” is currently being developed. In the new version of the language, the value `NULL` will be superseded by a special keyword `nullptr`, which functions identically to `NULL`. Once this new language revision is released, be prepared to see `nullptr` in professional code.

Practice Problems

1. It is illegal in C++ to have a reference to a reference. Why might this be the case? (*Hint: is there a meaningful distinction between the reference and what it refers to?*)
2. However, it is legal C++ to have a pointer to a pointer. Why might this be the case? (*Hint: is there a meaningful distinction between the pointer and what it points to?*)
3. What's wrong with this code snippet?

```
int myInteger = 137;
int* myIntPtr = myInteger;
*myIntPtr = 42;
```

4. Is this code snippet legal? If so, what does it do? If not, why not?

```
int myInteger = 0;
*(&myInteger) = 1;
```

5. When using `new[]` to allocate memory, you can access individual elements in the sequence using the notation `pointer[index]`. Recall that `pointer[index]` instructs C++ to start at the memory pointed at by `pointer`, march forward `index` elements, and read a value. Given a pointer initialized by `int* myIntPtr = new int`, what will `myIntPtr[0] = 42` do? What about `myIntPtr[1] = 42`? ♦
6. Why is there a null pointer constant but no null reference?
7. The C programming language has pointers but no references. Therefore, pure C code has no concept of “pass-by-reference.” In order for a function to update the values of its parameters outside of the function, the function must accept pointers as arguments that point to the variables to change. For example, the `Swap` function in pure C might be written as follows:

```
void Swap(int* one, int* two)
{
    int temp = *one;
    *one = *two;
    *two = temp;
}
```

To call this function to swap two integers, you would write code as follows:

```
int one = 137, two = 42;
Swap(&one, &two);
```

Rewrite the following function to use pass-by-pointer instead of pass-by-reference:

```
/* Replaces all instances of a given character in a string with another
 * character. This code uses the at() member function of the string, which
 * returns a reference to the character stored at that position.
 */
void ReplaceAll(string& input, char what, char with)
{
    for(int index = 0; index < input.length(); ++index)
        if(input.at(index) == what)
            input.at(index) = with;
}
```

Also, explain one advantage of pass-by-reference over pass-by-pointer and one advantage of pass-by-pointer over pass-by-reference.

8. It is legal to print pointers to `cout`; doing so prints out the memory address pointed at by the pointer. For example, the following code is perfectly legal:

```
int myInt = 0;
cout << &myInt << endl; // Prints the address of myInt
```

Write a program that allocates several `int` local variables and prints out their addresses. What do you notice about the memory locations they occupy? Then repeat this using `new` to dynamically allocate several `ints`. Do the memory addresses returned by `new` follow the same pattern?

Chapter 12: C Strings

C++ was designed to maximize backward compatibility with the C programming language, and consequently absorbed C's representation of character strings. Unlike the C++ `string` type, C strings are very difficult to work with. *Very* difficult. In fact, they are so difficult to work with that C++ programmers invented their own `string` type so that they can avoid directly using C strings.

While C strings are significantly more challenging than C++ `strings` and far more dangerous, no C++ text would be truly complete without a discussion of C strings. This chapter enters the perilous waters of C strings and their associated helper functions.

What is a C String?

In C++, `string` is a class that expresses many common operations with simple operator syntax. You can make deep copies with the `=` operator, concatenate with `+`, and check for equality with `==`. However, nearly every desirable feature of the C++ `string`, such as encapsulated memory management and logical operator syntax, uses language features specific to C++. C strings, on the other hand, are simply `char *` character pointers that store the starting addresses of specially-formatted character sequences. In other words, C++ `strings` exemplify abstraction and implementation hiding, while C strings among the lowest-level constructs you will routinely encounter in C++.

Because C strings operate at a low level, they present numerous programming challenges. When working with C strings you must manually allocate, resize, and delete string storage space. Also, because C strings are represented as blocks of memory, the syntax for accessing character ranges requires a sophisticated understanding of pointer manipulation. Compounding the problem, C string manipulation functions are cryptic and complicated.

However, because C strings are so low-level, they have several benefits over the C++ `string`. Since C strings are contiguous regions of memory, library code for manipulating C strings can be written in lightning-fast assembly code that can outperform even the most tightly-written C or C++ loops. Indeed, C strings will almost consistently outperform C++ `strings`.

Why study C strings?

In the early days of C++, the standard `string` type did not exist and C strings were the norm. Over time, C strings have been declining in usage. Most modern C++ programs either use the standard `string` type or a similarly-designed custom string type. So why should we dedicate any time to studying C strings? There are several reasons:

- **Legacy code.** C++ code often needs to interoperate with pure C code or older C++ code that predates the `string` type. In these cases you are likely to bump into C strings or classes layered on top of them, and without an understanding of C strings you are likely to get confused or stuck.
- **Edge cases.** It is legal C++ code to add an integer to a string literal, but doing so almost always results in a crash. Unless you understand how C strings and pointer arithmetic work, the root cause of this behavior will remain a mystery.
- **Library implementation.** You may be called upon to implement a string class that supports functionality beyond that of the standard `string`. Knowing how to manipulate C strings can greatly simplify this task.

We will encounter each of these cases in the remainder of this course reader.

Memory representations of C strings

A C string is represented in memory as a consecutive sequence of characters that ends with a “terminating null,” a special character with numeric value 0. Just as you can use the escape sequences `'\n'` for a newline and `'\t'` for a horizontal tab, you can use the `'\0'` (slash zero) escape sequence to represent a terminating null. Fortunately, whenever you write a string literal in C or C++, the compiler will automatically append a terminating null for you, so only rarely will you need to explicitly write the null character. For example, the string literal “Pirate” is actually seven characters long in C++ – six for “Pirate” plus one extra for the terminating null. Most library functions automatically insert terminating nulls for you, but you should always be sure to read the function documentation to verify this. Without a terminating null, C and C++ won't know when to stop reading characters, either returning garbage strings or causing crashes.*

The string “Pirate” might look something like this in memory:

Address 1000	P
1001	i
1002	r
1003	a
1004	t
1005	e
1006	\0

Note that while the end of the string is delineated by the terminating null, there is no indication here of where the string begins. Looking solely at the memory, it's unclear whether the string is “Pirate,” “irate,” “rate,” or “ate.” The only reason we “know” that the string is “Pirate” is because we know that its starting address is 1000.

This has important implications for working with C strings. Given a starting memory address, it is possible to entirely determine a string by reading characters until we reach a terminating null. In fact, provided the memory is laid out as shown above, it's possible to reference a string by means of a single `char *` variable that holds the starting address of the character block, in this case 1000. If you encounter a function that accepts a parameter of type `char*`, chances are that it accepts a C string rather than a pointer to a single character.

Memory Segments

Before we begin working with C strings, we need to quickly cover memory segments. When you run a C++ program, the operating system usually allocates memory for your program in “segments,” special regions dedicated to different tasks. You are most familiar with the stack segment, where local variables are stored and preserved between function calls. Also, there is a heap segment that stores memory dynamically allocated with the `new` and `delete` operators. There are two more segments, the code (or text) segment and the data segment, of which we must speak briefly.

When you write C or C++ code like the code shown below:

```
int main()
{
    char* myCString = "This is a C string!";
    return 0;
}
```

* Two C strings walk into a bar. One C string says “Hello, my name is John#30g4nvu342t7643t5k...”, so the second C string turns to the bartender and says “Please excuse my friend... he's not null-terminated.”

The text “This is a C string!” must be stored somewhere in memory when your program begins running. On many systems, this text is stored in either the read-only code segment or in a read-only portion of the data segment. This enables the compiler to reduce the overall memory footprint of the program, since if multiple parts of the code each use the same string literal they can all point to the same string. But this optimization is not without its costs. If you modify the contents of a read-only segment, you will crash your program with a *segmentation fault* (sometimes also called an *access violation* or “seg fault”).

Because your program cannot write to read-only memory segments, if you plan on manipulating the contents of a C string, you will need to first create a copy of that string somewhere where your program has write permission, usually in the heap. Thus, for the remainder of this chapter, any code that modifies strings will assume that the string resides either in the heap or on the stack (usually the former). Forgetting to duplicate the string and store its contents in a new buffer can cause many a debugging nightmare, so make sure that you have writing access before you try to manipulate C strings.

Allocating Space for Strings

Before you can manipulate a C string, you need to first allocate memory to store it. While traditionally this is done using older C library functions (briefly described in the “More to Explore” section), because we are working in C++, we will instead use the `new[]` and `delete[]` operators for memory management.

When allocating space for C strings, you must make sure to allocate enough space to store the entire string, including the terminating null character. If you do not allocate enough space, when you try to copy the string from its current location to your new buffer, you will write past the end of the buffer into memory you do not necessarily own. This is known as a *buffer overrun* and can crash your program or pose major security problems (take CS155 if you're curious about why this poses a security risk).

The best way to allocate space for a string is to make a new buffer with size equal to the length of the string you will be storing in the buffer. To get the length of a C string, you can use the handy `strlen` function, declared in the header file `<cstring>`.^{*} `strlen` returns the length of a string, *not* including the terminating null character. For example:

```
cout << strlen("String!") << endl; // Value is 7
char* myStr = "01234";
cout << strlen(myStr) << endl; // Value is 5
```

Thus, if you want to make a copy of a string, to allocate enough space you can use the following code:

```
/* Assume char *text points to a C string. */
char* myCopy = new char[strlen(text) + 1]; // Remember +1 for null
```

As always, remember to deallocate any memory you allocate with `new[]` with `delete[]`.

Basic String Operations

When working with C++ strings, you can make copies of a `string` using the `=` operator, as shown here:

```
string myString = "C++ strings are easy!";
string myOtherString = myString; // Copy myString
```

This is not the case for C strings, however. For example, consider the following code snippet:

^{*} `<cstring>` is the Standard C++ header file for the C string library. For programs written in pure C, you'll need to instead include the header file `<string.h>`.

```
char* myString = "C strings are hard!";
char* myOtherString = myString;
```

Here, the second line is a pointer assignment, *not* a string copy. As a result, when the second line finishes executing, `myString` and `myOtherString` both point to the same memory region, so changes to one string will affect the other string. This can be tricky to debug, since if you write the following code:

```
cout << myString << endl;
cout << myOtherString << endl;
```

You will get back two copies of the string `C strings are hard!`, which can trick you into thinking that you actually have made a deep copy of the string. To make a full copy of a C string, you can use the `strcpy` function, as shown below:

```
/* Assume char* source is initialized to a C string. */
char* destination = new char[strlen(source) + 1];
strcpy(destination, source);
```

Here, the line `strcpy(destination, source)` copies the data from `source` into `destination`. As with most C string operations, you must manually ensure that there is enough space in `destination` to hold a copy of `source`. Otherwise, `strcpy` will copy the data from `source` past the end of the buffer, which will wreck havoc on your program.

Note that the following code does *not* copy a C string from one location to another:

```
char* destination = new char[strlen(source) + 1];
*destination = *source; // Legal but incorrect
```

This code only copies the first character from `source` to `destination`, since `*source` is a single character, not the entire C string. The fact that `char*s` are used to point to C strings does not mean that C++ interprets them that way. As far as C++ is concerned, a `char*` is a pointer to a character and you will have to explicitly treat it like a C string using the library functions to make C++ think otherwise.

Another common string operation is concatenation. To append one C string onto the end of another, use the `strcat` function. Unlike in C++, when you concatenate two C strings, you must manually ensure there is enough allocated space to hold both strings. Here is some code to concatenate two C strings. Note the amount of space reserved by the `new[]` call only allocates space for one terminating null.

```
/* Assume char* firstPart, * secondPart are initialized C strings. */
char* result = new char[strlen(firstPart) + strlen(secondPart) + 1];
strcpy(result, firstPart); // Copy the first part.
strcat(result, secondPart); // Append the second part.
```

Because of C++'s array-pointer interchangeability, we can access individual characters using array syntax. For example:

```
/* Assume myCString is initialized to "C String Fun!"
 * This code might crash if myCString points to memory in a read-only
 * segment, so we'll assume you copied it by following the above steps.
 */
cout << myCString << endl;    // Output: C String Fun!
cout << myCString[0] << endl; // Output: C
myCString[10] = 'a';
cout << myCString << endl;    // Output: C String Fan!
```

Comparing Strings

When dealing with C strings, you **cannot** use the built-in relational operators (<, ==, etc.) to check for equality. This will only check to see if the two pointers point to the same object, not whether the strings are equal. Thus, you must use the `strcmp` function to compare two strings. `strcmp` compares two strings `str1` and `str2`, returning a negative number if `str1` precedes `str2` alphabetically, a positive number if `str1` comes after `str2`, and zero if `str1` and `str2` are equal. Thus, you can use the following code to check for string equality:

```
if(strcmp(str1, str2) == 0)
    /* ... strings are equal ... */
```

That `strcmp` returns zero if the two strings are equal is a common source of programming errors. For example, consider the following code:

```
char* one = "This is a string!";
char* two = "This is an entirely different string!";
if(strcmp(one, two)) // Watch out... this is incorrect!
    cout << "one and two are equal!" << endl;
else
    cout << "one and two are not equal!" << endl;
```

Here, we use the line `if(strcmp(one, two))` to check if `one` and `two` are equal. However, this check is completely wrong. In C++, any nonzero value is treated as “true” inside an `if` statement and any zero value is treated as “false.” However, `strcmp` returns 0 if the two strings are equal and a nonzero value otherwise, meaning the statement `if(strcmp(one, two))` will be true if the two strings are *different* and false if they're equivalent. When working with `strcmp`, make sure that you don't accidentally make this mistake.

Pointer Arithmetic

Because C strings are low-level constructs, string functions assume a familiarity with *pointer arithmetic* – the manipulation of pointers via arithmetic operators. This next section is tricky, but is necessary to be able to fully understand how to work with C strings. Fortunately, if you think back to our discussion of STL iterators, this material should be considerably less intimidating.

In C and C++, pointers are implemented as integral data types that store memory addresses of the values they point to. Thus, it is possible to change where a pointer points by adding and subtracting values from it.

Let's begin with an example using C strings. Suppose you have the string “Hello!” and a pointer to it laid out in memory as shown below:

Address 1000	H
1001	e
1002	l
1003	l
1004	o
1005	!
1006	\0

```
char* myString 1000
```

Currently, because `myString` stores memory address 1000, it points to the string “Hello!” What happens if we write a line of code like the one shown below?

```
myString = myString + 1;
```

In C and C++, adding one to a pointer returns a new pointer that points to the item one past the current pointer's location. In our current example, this is memory address 1001, the start of the string “ello!” Here is a drawing of the state of memory after performing the pointer arithmetic:

Address 1000	H
1001	e
1002	l
1003	l
1004	o
1005	!
1006	\0

`char* myString` 1001

In general, adding `n` to a pointer returns a pointer that points `n` items further than the original pointer. Thus, given the above state of memory, if we write `myString++`, we increment `myString` to point to memory location 1002, the string “llo!” Similarly, if afterwards we were to subtract two from `myString` by writing `myString -= 2`, `myString` would once again contain the value 1000 and would point to the string “Hello!”

Be careful when incrementing string pointers – it is easy to increment them beyond the ends of the buffers they point to. What if we were to write the code `myString += 1000`? The string “Hello!” is less than 1000 characters long, and pointer `myString` would point to a value far beyond the end of the string and into random memory. Trying to read or write from this pointer would therefore have undefined behavior and would probably result in a crash.

Let us consider one final type of pointer arithmetic, subtracting one pointer from another. Suppose we have the following C or C++ code:

```
char* ptr1 = "This is my string!";
char* ptr2 = ptr1 + 4;
cout << (ptr2 - ptr1) << endl;
```

What will the output be? Logically, we'd expect that since we set the value of `ptr2` to be four greater than `ptr1`, the result of the subtraction would be four. In general, subtracting two pointers yields the number of elements between them. Another way to interpret the result of pointer subtraction is as an array index. Assuming that `ptr1` points to the beginning of a C string and that `ptr2` points to an element somewhere in that string, `ptr2 - ptr1` will return the numeric index of `ptr2` in the string. This latter interpretation will be important in the upcoming section.

More String Functions

Armed with a understanding of pointer arithmetic, we can consider some more powerful string manipulation functions. Let us first consider the `strstr` function, which returns a pointer to the first occurrence of a given substring inside the specified string. If the substring isn't found, `strstr` returns `NULL` to signify an error.

`strstr` is demonstrated here:

```
char* myString = "C strings are difficult.";
char* found = strstr(myString, "if");
if(found == NULL)
    cout << "Substring not found." << endl;
else
    cout << "Substring occurs at index " << (found - myString) << endl;
```

You can also use the `strchr` function in a similar way to determine the first instance of a given character in a string.

One of the more useful string functions is the `strncpy` function, which copies a specified number of characters from the source string to the destination. However, `strncpy` is perhaps one of the most complicated library functions ever introduced.* Unlike the functions we've seen until this point, `strncpy` is not guaranteed to append a terminating null to a string. When you call `strncpy`, you specify a destination string, a source string, and a character count. If the end of the source string is reached before the specified number of characters have been copied, then `strncpy` will fill the remainder of the buffer with null characters. Otherwise, you must manually append a terminating null.

Although `strncpy` is complicated, it can be quite useful. For example, the following code demonstrates how to use `strncpy` in conjunction with pointer arithmetic to extract a substring from a source string:

```
char* GetSubstring(char* str, int start, int length)
{
    char* result = new char[length + 1]; // Include space for \0
    strncpy(result, str + start, length);
    result[length] = '\0'; // Manually append terminating null.
    return result;
}
```

The following table summarizes some of the more useful C string functions. As usual, we have not covered the `const` keyword yet, but it's safe to ignore it for now.

<code>size_t strlen (const char* str)</code>	<pre>int length = strlen("String!");</pre> Returns the length of the C string <code>str</code>, excluding the terminating null character. This function is useful for determining how much space is required to hold a copy of a string.
<code>char* strcpy (char* dest, const char* src)</code>	<pre>strcpy(myBuffer, "C strings rule!");</pre> Copies the contents of the C string <code>src</code> into the buffer pointed to by <code>dest</code>. <code>strcpy</code> will not perform any bounds checking, so you must make sure that the destination buffer has enough space to hold the source string. <code>strcpy</code> returns <code>dest</code>.
<code>char* strcat (char* dest, const char* src)</code>	<pre>strcat(myString, " plus more chars.");</pre> Appends the C string specified by <code>src</code> to the C string <code>dest</code>. Like <code>strcpy</code>, <code>strcat</code> will not bounds-check, so make sure you have enough room for both strings. <code>strcat</code> returns <code>dest</code>.

* The CS department's Nick Parlante calls `strncpy` the "Worst API design ever."

(C string functions, contd.)

<pre>int strcmp(const char* one, const char* two)</pre>	<pre>if(strcmp(myStr1, myStr2) == 0) // equal</pre> Compares two strings lexicographically and returns an integer representing how the strings relate to one another. If <code>one</code> precedes <code>two</code> alphabetically, <code>strcmp</code> returns a negative number. If the two are equal, <code>strcmp</code> returns zero. Otherwise, it returns a positive number.
<pre>int strncmp(const char* one, const char* two, size_t numChars)</pre>	<pre>if(strncmp(myStr1, myStr2, 4) == 0) // First 4 chars equal</pre> Identical to <code>strcmp</code>, except that <code>strncmp</code> accepts a third parameter indicating the maximum number of characters to compare.
<pre>const char* strstr(const char* src, const char* key) char* strstr(char* src, const char* key)</pre>	<pre>if(strstr(myStr, "iffy") != NULL) // found</pre> Searches for the substring <code>key</code> in the string <code>source</code> and returns a pointer to the first instance of the substring. If <code>key</code> is not found, <code>strstr</code> returns <code>NULL</code>.
<pre>char* strchr(char* src, int key) const char* strchr(const char* src, int key)</pre>	<pre>if(strchr(myStr, 'a') != NULL) // found</pre> Searches for the character <code>key</code> in the string <code>source</code> and returns a pointer to the first instance of the character. If <code>key</code> is not found, <code>strchr</code> returns <code>NULL</code>. Despite the fact that <code>key</code> is of type <code>int</code>, <code>key</code> will be treated as a <code>char</code> inside of <code>strchr</code>.
<pre>char* strrchr(char* src, int key) const char* strrchr(const char* src, int key)</pre>	<pre>if(strrchr(myStr, 'a') != NULL) // found</pre> Searches for the character <code>key</code> in the <code>source</code> and returns a pointer to the <i>last</i> instance of the character. If <code>key</code> is not found, <code>strchr</code> returns <code>NULL</code>.
<pre>char* strncpy (char* dest, const char* src, size_t count)</pre>	<pre>strncpy(myBuffer, "Theta", 3);</pre> Copies up to <code>count</code> characters from the string <code>src</code> into the buffer pointed to by <code>dest</code>. If the end of <code>src</code> is reached before <code>count</code> characters are written, <code>strncpy</code> appends null characters to <code>dest</code> until <code>count</code> characters have been written. Otherwise, <code>strncpy</code> does not append a terminating null. <code>strncpy</code> returns <code>dest</code>.
<pre>char* strncat (char* dest, const char* src, size_t count)</pre>	<pre>strncat(myBuffer, "Theta", 3); // Appends "The" to myBuffer</pre> Appends up to <code>count</code> characters from the string <code>src</code> to the buffer pointed to by <code>dest</code>. Unlike <code>strncpy</code>, <code>strncat</code> will always append a terminating null to the string.
<pre>size_t strcspn(const char* source, const char* chars)</pre>	<pre>int firstInt = strcspn(myStr, "0123456789");</pre> Returns the index of the first character in <code>source</code> that matches any of the characters specified in the <code>chars</code> string. If the entire string is made of characters not specified in <code>chars</code>, <code>strcspn</code> returns the length of the string. This function is similar to the <code>find_first_of</code> function of the C++ string.
<pre>char* strpbrk(char* source, const char* chars) const char* strpbrk(const char* source, const char* chars)</pre>	<pre>if(strpbrk(myStr, "0123456789") == NULL) // No ints found</pre> Returns a pointer to the first character in the source string that is contained in the second string. If no matches are found, <code>strpbrk</code> returns <code>NULL</code>. This function is functionally quite similar to <code>strcspn</code>.
<pre>size_t strspn (const char* source, const char* chars);</pre>	<pre>int numInts = strspn(myStr, "0123456789");</pre> Returns the index of the first character in <code>source</code> that is <i>not</i> one of the characters specified in the <code>chars</code> string. If the entire string is made of characters specified in <code>chars</code>, <code>strspn</code> returns the length of the string. This function is similar to the <code>find_first_not_of</code> function of the C++ string.

More to Explore

While this chapter has tried to demystify the beast that is the C string, there are several important topics we did not touch on. If you're interested in learning more about C strings, consider looking into the following topics:

1. **Command-line parameters:** Have you ever wondered why `main` returns a value? It's because it's possible to pass parameters to the `main` function by invoking your program at the command line. To write a `main` function that accepts parameters, change its declaration from `int main()` to

```
int main(int argc, char* argv[])
```

Here, `argc` is the number of parameters passed to `main` (the number of C strings in the array `argv`), and `argv` is an array of C strings containing the parameters. This is especially useful for those of you interested in writing command-line utilities. You will most certainly see this version of `main` if you continue writing more serious C++ code.

2. **`malloc`, `realloc`, and `free`:** These three functions are older C memory management functions that allocate, deallocate, and resize blocks of memory. These functions can be unsafe when mixed with C++ code (see Appendix 0: Moving from C to C++), but are nonetheless frequently used. Consider reading up on these functions if you're interested in programming in pure C.
3. **`sprintf` and `scanf`:** The C++ `stringstream` class allows you to easily read and write formatted data to and from C++ strings. The `sprintf` and `scanf` functions let you perform similar functions on C strings.
4. **C memory manipulation routines:** The C header file `<cstring>` contains a set of functions that let you move, fill, and copy blocks of memory. Although this is an advanced topic, some of these functions are useful in conjunction with C strings. For example, the `memmove` function can be used to shift characters forward and backward in a string to make room for insertion of a new substring. Similarly, you can use the `memset` function to create a string that's several repeated copies of a character, or to fill a buffer with terminating nulls before writing onto it.

Practice Problems

1. Explain why the code `string myString = "String" + '!'` will not work as intended. What is it actually doing? (*Hint: chars can implicitly be converted to ints.*) ♦
2. Write a function `Exaggerate` that accepts a C string and increases the value of each non-nine digit in it by one. For example, given the input string “I worked 90 hours and drove 24 miles” the function would change it to read “I worked 91 hours and drove 35 miles.”
3. Explain why the following code generates an “array bounds overflow” error during compilation:

```
char myString[6] = "Hello!";
```

4. When working with C++ strings, the `erase` function can be used as `myString.erase(n)` to erase all characters in the string starting at position `n`, where `n` is assumed to be within the bounds of the string. Write a function `TruncateString` that accepts a `char *` C-style string and an index in the string, then modifies the string by removing all characters in the string starting at that position. You can assume that the index is in bounds. (*Hint: Do you actually need to remove the characters at that position, or can you trick C++ into thinking that they're not there?*)

5. There are several ways to iterate over the contents of a C string. For example, we can iterate over the string using bracket syntax using the following construct:

```
int length = strlen(myStr); // Compute once and store for efficiency.
for(int k = 0; k < length; ++k)
    myStr[k] = /* ... */
```

Another means for iterating over the C string uses pointer arithmetic and explicitly checks for the terminating null character. This is shown below:

```
for(char* currLoc = myStr; *currLoc != '\0'; ++currLoc)
    *currLoc = /* ... */
```

Write a function `CountFrequency` which accepts as input a C-style string and a character, then returns the number of times that the character appears in the string. You should write this function two ways – once using the first style of for loop, and once using the second.

6. It is legal to provide raw C++ pointers to STL algorithms instead of STL iterators. Rewrite `CountFrequency` using the STL `count` algorithm.
7. Compiler vendors often add their own nonstandard library functions to the standard header files to entice customers. One common addition to the C string library is a function `strcasecmp`, which returns how two strings compare to one another case-insensitively. For example, `strcasecmp("HeLlO!", "hello!")` would return zero, while `strcasecmp("Hello", "Goodbye")` would return a negative value because `Goodbye` alphabetically precedes `Hello`.

`strcasecmp` is not available with all C++ compilers, but it's still a useful function to have at your disposal. While implementing a completely-correct version of `strcasecmp` is a bit tricky (mainly when deciding how to compare letters and punctuation symbols), it is not particularly difficult to write a similar function called `StrCaseEqual` which returns if two strings, compared case-insensitively, are equal to one another.

Without using `strcasecmp`, implement the `StrCaseEqual` function. It should accept as input two C-style strings and return whether they are exactly identical when compared case-insensitively. The `toupper`, `tolower`, or `isalpha` functions from `<cctype>` might be useful here; consult a C++ reference for more details. To give you more practice directly manipulating C strings, your solution should not use any of the standard library functions other than the ones exported by `<cctype>`.

8. Another amusing nonstandard C string manipulation function is the `strfry` function, available in GNU-compliant compilers. As described in [GNU]:

(`strfry`) addresses the perennial programming quandary: “How do I take good data in string form and painlessly turn it into garbage?” This is actually a fairly simple task for C programmers who do not use the GNU C library string functions, but for programs based on the GNU C library, the `strfry` function is the preferred method for destroying string data.

`strfry` accepts as input a C-style string, then randomly permutes its contents. For example, calling `strfry` on a string containing the text “Unscramble,” `strfry` might permute it to contain “barmsc-lUne.” Write an implementation of `strfry`. (Hint: Isn't there an algorithm that scrambles ranges for you?)

9. Suppose that we want to allocate a C string buffer large enough to hold a copy of an existing C string. A common mistake is to write the following:

```
char* buffer = new char[strlen(kSourceString + 1)]; // <-- Error here!
```

What does this code do? What was it intended to do? Why doesn't it work correctly?

Chapter 13: The Preprocessor

One of the most exciting parts of writing a C++ program is pressing the “compile” button and watching as your code transforms from static text into dynamic software. But what exactly goes on behind the scenes that makes this transition possible? There are several steps involved in compilation, among the first of which is *preprocessing*, where a special program called the *preprocessor* reads in commands called *directives* and modifies your code before handing it off to the compiler for further analysis. You have already seen one of the more common preprocessor directives, `#include`, which imports additional code into your program. However, the preprocessor has far more functionality and is capable of working absolute wonders on your code. But while the preprocessor is powerful, it is difficult to use correctly and can lead to subtle and complex bugs. This chapter introduces the preprocessor, highlights potential sources of error, and concludes with advanced preprocessor techniques.

A Word of Warning

The preprocessor was developed in the early days of the C programming language, before many of the more modern constructs of C and C++ had been developed. Since then, both C and C++ have introduced new language features that have obsoleted or superseded much of the preprocessor's functionality and consequently you should attempt to minimize your use of the preprocessor. This is not to say, of course, that you should never use the preprocessor – there are times when it's an excellent tool for the job, as you'll see later in the chapter – but do consider other options before adding a hastily-crafted directive.

`#include` Explained

In both CS106B/X and CS106L, every program you've encountered has begun with several lines using the `#include` directive; for example, `#include <iostream>` or `#include "genlib.h"`. Intuitively, these directives tell the preprocessor to import library code into your programs. Literally, `#include` instructs the preprocessor to locate the specified file and to insert its contents in place of the directive itself. Thus, when you write `#include "genlib.h"` at the top of your CS106B/X assignments, it is as if you had copied and pasted the contents of `genlib.h` into your source file. These header files usually contain prototypes or implementations of the functions and classes they export, which is why the directive is necessary to access other libraries.

You may have noticed that when `#include`-ing CS106B/X-specific libraries, you've surrounded the name of the file in double quotes (e.g. `"genlib.h"`), but when referencing C++ standard library components, you surround the header in angle brackets (e.g. `<iostream>`). These two different forms of `#include` instruct the preprocessor where to look for the specified file. If a filename is surrounded in angle brackets, the preprocessor searches for it a compiler-specific directory containing C++ standard library files. When filenames are in quotes, the preprocessor will look in the current directory.

`#include` is a preprocessor directive, not a C++ statement, and is subject to a different set of syntax restrictions than normal C++ code. For example, to use `#include` (or any preprocessor directive, for that matter), the directive must be the first non-whitespace text on its line. For example, the following is illegal:

```
cout << #include <iostream> << endl; // Error: #include must be at start of line.
```

Second, because `#include` is a preprocessor directive, not a C++ statement, it must not end with a semicolon. That is, `#include <iostream>;` will probably cause a compiler error or warning. Finally, the entire `#include` directive must appear on a single line, so the following code will not compile:

```
#include
<iostream> // Error: Multi-line preprocessor directives are illegal.
```

The #define Directive

One of the most commonly used (and abused) preprocessor directives is `#define`, the equivalent of a “search and replace” operation on your C++ source files. While `#include` splices new text into an existing C++ source file, `#define` replaces certain text strings in your C++ file with other values. The syntax for `#define` is

```
#define phrase replacement
```

After encountering a `#define` directive, whenever the preprocessor finds **phrase** in your source code, it will replace it with **replacement**. For example, consider the following program:

```
#define MY_CONSTANT 137

int main()
{
    int x = MY_CONSTANT - 3;
    return 0;
}
```

The first line of this program tells the preprocessor to replace all instances of `MY_CONSTANT` with the phrase `137`. Consequently, when the preprocessor encounters the line

```
int x = MY_CONSTANT - 3;
```

It will transform it to read

```
int x = 137 - 3;
```

So `x` will take the value 134.

Because `#define` is a preprocessor directive and not a C++ statement, its syntax can be confusing. For example, `#define` determines the stop of the **phrase** portion of the statement and the start of the **replacement** portion by the position of the first whitespace character. Thus, if you write

```
#define TWO WORDS 137
```

The preprocessor will interpret this as a directive to replace the phrase `TWO` with `WORDS 137`, which is probably not what you intended. The **replacement** portion of the `#define` directive consists of all text after **phrase** that precedes the newline character. Consequently, it is legal to write statements of the form `#define phrase` without defining a replacement. In that case, when the preprocessor encounters the specified phrase in your code, it will replace it with nothingness, effectively removing it.

Note that the preprocessor treats C++ source code as sequences of strings, rather than representations of higher-level C++ constructs. For example, the preprocessor treats `int x = 137` as the strings “`int`,” “`x`,” “`=`,” and “`137`” rather than a statement creating a variable `x` with value 137.* It may help to think of the preprocessor as a scanner that can read strings and recognize characters but which has no understanding whatsoever of their meanings, much in the same way a native English speaker might be able to split Czech text into individual words without comprehending the source material.

* Technically speaking, the preprocessor operates on “preprocessor tokens,” which are slightly different from the whitespace-differentiated pieces of your code. For example, the preprocessor treats string literals containing whitespace as a single object rather than as a collection of smaller pieces.

That the preprocessor works with text strings rather than language concepts is a source of potential problems. For example, consider the following `#define` statements, which define margins on a page:

```
#define LEFT_MARGIN 100
#define RIGHT_MARGIN 100
#define SCALE .5

/* Total margin is the sum of the left and right margins, multiplied by some
 * scaling factor.
 */
#define TOTAL_MARGIN LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE
```

What happens if we write the following code?

```
int x = 2 * TOTAL_MARGIN;
```

Intuitively, this should set `x` to twice the value of `TOTAL_MARGIN`, but unfortunately this is not the case. Let's trace through how the preprocessor will expand out this expression. First, the preprocessor will expand `TOTAL_MARGIN` to `LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE`, as shown here:

```
int x = 2 * LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE;
```

Initially, this may seem correct, but look closely at the operator precedence. C++ interprets this statement as

```
int x = (2 * LEFT_MARGIN * SCALE) + RIGHT_MARGIN * SCALE;
```

Rather the expected

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

And the computation will be incorrect. The problem is that the preprocessor treats the replacement for `TOTAL_MARGIN` as a string, not a mathematic expression, and has no concept of operator precedence. This sort of error – where a `#defined` constant does not interact properly with arithmetic expressions – is a common mistake. Fortunately, we can easily correct this error by adding additional parentheses to our `#define`. Let's re-write the `#define` statement as

```
#define TOTAL_MARGIN (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE)
```

We've surrounded the replacement phrase with parentheses, meaning that any arithmetic operators applied to the expression will treat the replacement string as a single mathematical value. Now, if we write

```
int x = 2 * TOTAL_MARGIN;
```

It expands out to

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

Which is the computation we want. In general, if you `#define` a constant in terms of an expression applied to other `#defined` constants, make sure to surround the resulting expression in parentheses.

Although this expression is certainly more correct than the previous one, it too has its problems. What if we re-define `LEFT_MARGIN` as shown below?

```
#define LEFT_MARGIN 200 - 100
```

Now, if we write

```
int x = 2 * TOTAL_MARGIN
```

It will expand out to

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

Which in turn expands to

```
int x = 2 * (200 - 100 * .5 + 100 * .5)
```

Which yields the incorrect result because $(200 - 100 * .5 + 100 * .5)$ is interpreted as

```
(200 - (100 * .5) + 100 * .5)
```

Rather than the expected

```
((200 - 100) * .5 + 100 * .5)
```

The problem is that the `#define` statement itself has an operator precedence error. As with last time, to fix this, we'll add some additional parentheses to the expression to yield

```
#define TOTAL_MARGIN ((LEFT_MARGIN) * (SCALE) + (RIGHT_MARGIN) * (SCALE))
```

This corrects the problem by ensuring that each `#define` subexpression is treated as a complete entity when used in arithmetic expressions. When writing a `#define` expression in terms of other `#defines`, make sure that you take this into account, or chances are that your constant will not have the correct value.

Another potential source of error with `#define` concerns the use of semicolons. If you terminate a `#define` statement with a semicolon, the preprocessor will treat the semicolon as part of the replacement phrase, rather than as an “end of statement” declaration. In some cases, this may be what you want, but most of the time it just leads to frustrating debugging errors. For example, consider the following code snippet:

```
#define MY_CONSTANT 137; // Oops-- unwanted semicolon!
```

```
int x = MY_CONSTANT * 3;
```

During preprocessing, the preprocessor will convert the line `int x = MY_CONSTANT * 3` to read

```
int x = 137; * 3;
```

This is not legal C++ code and will cause a compile-time error. However, because the problem is in the preprocessed code, rather than the original C++ code, it may be difficult to track down the source of the error. Almost all C++ compilers will give you an error about the statement `* 3` rather than a malformed `#define`.

As you can tell, using `#define` to define constants can lead to subtle and difficult-to-track bugs. Consequently, it's strongly preferred that you define constants using the `const` keyword. For example, consider the following `const` declarations:

```
const int LEFT_MARGIN = 200 - 100;
const int RIGHT_MARGIN = 100;
const int SCALE = .5;
const int TOTAL_MARGIN = LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE;
int x = 2 * TOTAL_MARGIN;
```

Even though we've used mathematical expressions inside the `const` declarations, this code will work as expected because it is interpreted by the C++ compiler rather than the preprocessor. Since the compiler understands the *meaning* of the symbols `200 - 100`, rather than just the characters themselves, you will not need to worry about strange operator precedence bugs.

Compile-time Conditional Expressions

Suppose we make the following header file, `myfile.h`, which defines a `struct` called `MyStruct`:

MyFile.h

```
struct MyStruct
{
    int x;
    double y;
    char z;
};
```

What happens when we try to compile the following program?

```
#include "myfile.h"
#include "myfile.h" // #include the same file twice

int main()
{
    return 0;
}
```

This code looks innocuous, but produces a compile-time error complaining about a redefinition of `struct MyStruct`. The reason is simple – when the preprocessor encounters each `#include` statement, it copies the contents of `myfile.h` into the program without checking whether or not it has already included the file. Consequently, it will copy the contents of `myfile.h` into the code twice, and the resulting code looks like this:

```
struct MyStruct
{
    int x;
    double y;
    char z;
};
struct MyStruct // <-- Error occurs here
{
    int x;
    double y;
    char z;
};

int main()
{
    return 0;
}
```

The indicated line is the source of our compiler error – we've doubly-defined `struct MyStruct`. To solve this problem, you might think that we should simply have a policy of not `#include`-ing the same file twice. In principle this may seem easy, but in a large project where several files each `#include` each other, it may be possible for a file to indirectly `#include` the same file twice. Somehow, we need to prevent this problem from happening.

The problem we're running into stems from the fact that the preprocessor has no memory about what it has done in the past. Somehow, we need to give the preprocessor instructions of the form "if you haven't already done so, #include the contents of this file." For situations like these, the preprocessor supports conditional expressions. Just as a C++ program can use `if ... else if ... else` to change program flow based on variables, the preprocessor can use a set of preprocessor directives to conditionally include a section of code based on #defined values.

There are several conditional structures built into the preprocessor, the most versatile of which are `#if`, `#elif`, `#else`, and `#endif`. As you might expect, you use these directives according to the pattern

```
#if statement
...
#elif another-statement
...
#elif yet-another-statement
...
#else
...
#endif
```

There are two details we need to consider here. First, what sorts of expressions can these preprocessor directives evaluate? Because the preprocessor operates before the rest of the code has been compiled, preprocessor directives can only refer to #defined constants, integer values, and arithmetic and logical expressions of those values. Here are some examples, supposing that some constant `MY_CONSTANT` is defined to 42:

```
#if MY_CONSTANT > 137           // Legal
#if MY_CONSTANT * 42 == MY_CONSTANT // Legal
#if sqrt(MY_CONSTANT) < 4       // Illegal, cannot call function sqrt
#if MY_CONSTANT == 3.14         // Illegal, can only use integral values
```

In addition to the above expressions, you can use the `defined` predicate, which takes as a parameter the name of a value that may have previously been #defined. If the constant has been #defined, `defined` evaluates to 1; otherwise it evaluates to 0. For example, if `MY_CONSTANT` has been previously #defined and `OTHER_CONSTANT` has not, then the following expressions are all legal:

```
#if defined(MY_CONSTANT)      // Evaluates to true.
#if defined(OTHER_CONSTANT)  // Evaluates to false.
#if !defined(MY_CONSTANT)    // Evaluates to false.
```

Now that we've seen what sorts of expressions we can use in preprocessor conditional expressions, what is the *effect* of these constructs? Unlike regular `if` statements, which change control flow at execution, preprocessor conditional expressions determine whether pieces of code are included in the resulting source file. For example, consider the following code:

```
#if defined(A)
    cout << "A is defined." << endl;
#elif defined(B)
    cout << "B is defined." << endl;
#elif defined(C)
    cout << "C is defined." << endl;
#else
    cout << "None of A, B, or C is defined." << endl;
#endif
```


Here, when the preprocessor encounters these directives, whichever of the conditional expressions evaluates to true will have its corresponding code block included in the final program, and the rest will be ignored. For example, if `A` is defined, this entire code block will reduce down to

```
cout << "A is defined." << endl;
```

And the rest of the code will be ignored.

One interesting use of the `#if ... #endif` construct is to comment out blocks of code. Since C++ interprets all nonzero values as true and zero as false, surrounding a block of code in a `#if 0 ... #endif` block causes the preprocessor to eliminate that block. Moreover, unlike the traditional `/* ... */` comment type, preprocessor directives can be nested, so removing a block of code using `#if 0 ... #endif` doesn't run into the same problems as commenting the code out with `/* ... */`.

In addition to the above conditional directives, C++ provides two shorthand directives, `#ifdef` and `#ifndef`. `#ifdef` (**if defined**) is a directive that takes as an argument a symbol and evaluates to true if the symbol has been `#defined`. Thus the directive `#ifdef symbol` is completely equivalent to `#if defined(symbol)`. C++ also provides `#ifndef` (**if not defined**), which acts as the opposite of `#ifdef`; `#ifndef symbol` is equivalent to `#if !defined(symbol)`. Although these directives are strictly weaker than the more generic `#if`, it is far more common in practice to see `#ifdef` and `#ifndef` rather than `#if defined` and `#if !defined`, primarily because they are more concise.

Using the conditional preprocessor directives, we can solve the problem of double-including header files. Let's return to our example with `#include "myfile.h"` appearing twice in one file. Here is a slightly modified version of the `myfile.h` file that introduces some conditional directives:

MyFile.h, version 2

```
#ifndef MyFile_included
#define MyFile_included

struct MyStruct
{
    int x;
    double y;
    char z;
};

#endif
```

Here, we've surrounded the entire file in a block `#ifndef MyFile_included ... #endif`. The specific name `MyFile_included` is not particularly important, other than the fact that it is unique to the `myfile.h` file. We could have just as easily chosen something like `#ifndef sdf39527dkb2` or another unique name, but the custom is to choose a name determined by the file name. Immediately after this `#ifndef` statement, we `#define` the constant we are considering inside the `#ifndef`. To see exactly what effect this has on the code, let's return to our original source file, reprinted below:

```
#include "myfile.h"
#include "myfile.h" // #include the same file twice

int main()
{
    return 0;
}
```

With the modified version of `myfile.h`, this code expands out to

```
#ifndef MyFile_included
#define MyFile_included

struct MyStruct
{
    int x;
    double y;
    char z;
};

#endif
#ifndef MyFile_included
#define MyFile_included

struct MyStruct
{
    int x;
    double y;
    char z;
};

#endif

int main()
{
    return 0;
}
```

Now, as the preprocessor begins evaluating the `#ifndef` statements, the first `#ifndef ... #endif` block from the header file will be included since the constant `MyFile_included` hasn't been defined yet. The code then `#defines` `MyFile_included`, so when the program encounters the second `#ifndef` block, the code inside the `#ifndef ... #endif` block will not be included. Effectively, we've ensured that the contents of a file can only be `#included` once in a program. The net program thus looks like this:

```
struct MyStruct
{
    int x;
    double y;
    char z;
};

int main()
{
    return 0;
}
```

Which is exactly what we wanted. This technique, known as an *include guard*, is used throughout professional C++ code, and, in fact, the boilerplate `#ifndef / #define / #endif` structure is found in virtually every header file in use today. Whenever writing header files, be sure to surround them with the appropriate preprocessor directives.

Macros

One of the most common and complex uses of the preprocessor is to define *macros*, compile-time functions that accepts parameters and output code. Despite the surface similarity, however, preprocessor macros and C++

functions have little in common. C++ functions represent code that executes at runtime to manipulate data, while macros expand out into newly-generated C++ code during preprocessing.

To create macros, you use an alternative syntax for `#define` that specifies a parameter list in addition to the constant name and expansion. The syntax looks like this:

```
#define macroname(parameter1, parameter2, ... , parameterN) macro-body*
```

Now, when the preprocessor encounters a call to a function named **macroname**, it will replace it with the text in **macro-body**. For example, consider the following macro definition:

```
#define PLUS_ONE(x) ((x) + 1)
```

Now, if we write

```
int x = PLUS_ONE(137);
```

The preprocessor will expand this code out to

```
int x = ((137) + 1);
```

So `x` will have the value 138.

If you'll notice, unlike C++ functions, preprocessor macros do not have a return value. Macros expand out into C++ code, so the “return value” of a macro is the result of the expressions it creates. In the case of `PLUS_ONE`, this is the value of the parameter plus one because the replacement is interpreted as a mathematical expression. However, macros need not act like C++ functions. Consider, for example, the following macro:

```
#define MAKE_FUNCTION(fnName) void fnName()
```

Now, if we write the following C++ code:

```
MAKE_FUNCTION(MyFunction)
{
    cout << "This is a function!" << endl;
}
```

The `MAKE_FUNCTION` macro will convert it into the function definition

```
void MyFunction()
{
    cout << "This is a function!" << endl;
}
```

As you can see, this is entirely different than the `PLUS_ONE` macro demonstrated above. In general, a macro can be expanded out to any text and that text will be treated as though it were part of the original C++ source file. This is a mixed blessing. In many cases, as you'll see later in the chapter, it can be exceptionally useful. However, as with other uses of `#define`, macros can lead to incredibly subtle bugs that can be difficult to track down. Perhaps the most famous example of macros gone wrong is this `MAX` macro:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

* Note that when using `#define`, the opening parenthesis that starts the argument list must not be preceded by whitespace. Otherwise, the preprocessor will treat it as part of the replacement phrase for a `#defined` constant.

Here, the macro takes in two parameters and uses the `? :` operator to choose the larger of the two. If you're not familiar with the `? :` operator, the syntax is as follows:

```
expression ? result-if-true : result-if-false
```

In our case, `((a) > (b) ? (a) : (b))` evaluates the expression `(a) > (b)`. If the statement is true, the value of the expression is `(a)`; otherwise it is `(b)`.

At first, this macro might seem innocuous and in fact will work in almost every situation. For example:

```
int x = MAX(100, 200);
```

Expands out to

```
int x = ((100) > (200) ? (100) : (200));
```

Which assigns `x` the value 200. However, what happens if we write the following?

```
int x = MAX(MyFn1(), MyFn2());
```

This expands out to

```
int x = ((MyFn1()) > (MyFn2()) ? (MyFn1()) : (MyFn2()));
```

While this will assign `x` the larger of `MyFn1()` and `MyFn2()`, it will not evaluate the parameters only once, as you would expect of a regular C++ function. As you can see from the expansion of the `MAX` macro, the functions will be called once during the comparison and possibly twice in the second half of the statement. If `MyFn1()` or `MyFn2()` are slow, this is inefficient, and if either of the two have side effects (for example, writing to disk or changing a global variable), the code will be incorrect.

The above example with `MAX` illustrates an important point when working with the preprocessor – in general, C++ functions are safer, less error-prone, and more readable than preprocessor macros. If you ever find yourself wanting to write a macro, see if you can accomplish the task at hand with a regular C++ function. If you can, use the C++ function instead of the macro – you'll save yourself hours of debugging nightmares.

Inline Functions

One of the motivations behind macros in pure C was program efficiency from *inlining*. For example, consider the `MAX` macro from earlier, which was defined as

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

If we call this macro, then the code for selecting the maximum element is directly inserted at the spot where the macro is used. For example, the following code:

```
int myInt = MAX(one, two);
```

Expands out to

```
int myInt = ((one) > (two) ? (one) : (two));
```

When the compiler sees this code, it will generate machine code that directly performs the test. If we had instead written `MAX` as a regular function, the compiler would probably implement the call to `MAX` as follows:

1. Call the function called `MAX` (which actually performs the comparison)
2. Store the result in the variable `myInt`.

This is considerably less efficient than the macro because of the time required to set up the function call. In computer science jargon, the macro is *inlined* because the compiler places the contents of the “function” at the call site instead of inserting an indirect jump to the code for the function. Inlined functions can be considerably more efficient than their non-inline counterparts, and so for many years macros were the preferred means for writing utility routines.

Bjarne Stroustrup is particularly opposed to the preprocessor because of its idiosyncrasies and potential for errors, and to entice programmers to use safer language features developed the `inline` keyword, which can be applied to functions to suggest that the compiler automatically inline them. Inline functions are not treated like macros – they’re actual functions and none of the edge cases of macros apply to them – but the compiler will try to safely inline them if at all possible. For example, the following `Max` function is marked `inline`, so a reasonably good compiler should perform the same optimization on the `Max` function that it would on the `MAX` macro:

```
inline int Max(int one, int two)
{
    return one > two ? one : two;
}
```

The `inline` keyword is only a suggestion to the compiler and may be ignored if the compiler deems it either too difficult or too costly to inline the function. However, when writing short functions it sometimes helps to mark the function `inline` to improve performance.

A #define Cautionary Tale

`#define` is a powerful directive that enables you to completely transform C++. However, many C/C++ experts agree that you should not use `#define` unless it is absolutely necessary. Preprocessor macros and constants obfuscate code and make it harder to debug, and with a few cryptic `#defines` veteran C++ programmers will be at a loss to understand your programs. As an example, consider the following code, which references an external file `mydefines.h`:

```
#include "mydefines.h"
```

```
Once upon a time a little boy took a walk in a park
He (the child) found a small stone and threw it (the stone) in a pond
The end
```

Surprisingly, and worryingly, it is possible to make this code compile and run, provided that `mydefines.h` contains the proper `#defines`. For example, here's one possible `mydefines.h` file that makes the code compile:

File: mydefines.h

```
#ifndef mydefines_included
#define mydefines_included

#include <iostream>
using namespace std;

#define Once
#define upon
#define a
#define time upon
#define little
#define boy
#define took upon
#define walk
#define in walk
#define the
#define park a
#define He(n) n MyFunction(n x)
#define child int
#define found {
#define small return
#define stone x;
#define and in
#define threw }
#define it(n) int main() {
#define pond cout << MyFunction(137) << endl;
#define end return 0; }
#define The the

#endif
```

After preprocessing (and some whitespace formatting), this yields the program

```
#include <iostream>
using namespace std;

int MyFunction(int x)
{
    return x;
}

int main()
{
    cout << MyFunction(137) << endl;
    return 0;
}
```

While this example is admittedly a degenerate case, it should indicate exactly how disastrous it can be for your programs to misuse #defined symbols. Programmers expect certain structures when reading C++ code, and by obscuring those structures behind walls of #defines you will confuse people who have to read your code. Worse, if you step away from your code for a short time (say, a week or a month), you may very well return to it with absolutely no idea how your code operates. Consequently, when working with #define, always be sure to ask yourself whether or not you are improving the readability of your code.

Advanced Preprocessor Techniques

The previous section ended on a rather grim note, giving an example of preprocessor usage gone awry. But to entirely eschew the preprocessor in favor of other language features would also be an error. In several circumstances, the preprocessor can perform tasks that other C++ language features cannot accomplish. The remainder of this chapter explores where the preprocessor can be an invaluable tool for solving problems and points out several strengths and weaknesses of preprocessor-based approaches.

Special Preprocessor Values

The preprocessor has access to several special values that contain information about the state of the file currently being compiled. The values act like `#defined` constants in that they are replaced whenever encountered in a program. For example, the values `__DATE__` and `__TIME__` contain string representations of the date and time that the program was compiled. Thus, you can write an automatically-generated “about this program” function using syntax similar to this:

```
string GetAboutInformation()
{
    stringstream result;
    result << "This program was compiled on " << __DATE__;
    result << " at time " << __TIME__;
    return result.str();
}
```

Similarly, there are two other values, `__LINE__` and `__FILE__`, which contain the current line number and the name of the file being compiled. We'll see an example of where `__LINE__` and `__FILE__` can be useful later in this chapter.

String Manipulation Functions

While often dangerous, there are times where macros can be more powerful or more useful than regular C++ functions. Since macros work with source-level text strings instead of at the C++ language level, some pieces of information are available to macros that are not accessible using other C++ techniques. For example, let's return to the `MAX` macro we used in the previous chapter:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Here, the arguments `a` and `b` to `MAX` are passed by *string* – that is, the arguments are passed as the strings that compose them. For example, `MAX(10, 15)` passes in the value `10` not as a numeric value ten, but as the character `1` followed by the character `0`. The preprocessor provides two different operators for manipulating the strings passed in as parameters. First is the *stringizing operator*, represented by the `#` symbol, which returns a quoted, C string representation of the parameter. For example, consider the following macro:

```
#define PRINTOUT(n) cout << #n << " has value " << (n) << endl
```

Here, we take in a single parameter, `n`. We then use the stringizing operator to print out a string representation of `n`, followed by the value of the expression `n`. For example, given the following code snippet:

```
int x = 137;
PRINTOUT(x * 42);
```

After preprocessing, this yields the C++ code

```
int x = 137;
cout << "x * 42" << " has value " << (x * 42) << endl;
```

Note that after the above program has been compiled from C++ to machine code, any notions of the original variable `x` or the individual expressions making up the program will have been completely eliminated, since variables exist only at the C++ level. However, through the stringizing operator, it is possible to preserve a string version of portions of the C++ source code in the final program, as demonstrated above. This is useful when writing diagnostic functions, as you'll see later in this chapter.

The second preprocessor string manipulation operator is the *string concatenation* operator, also known as the *token-pasting* operator. This operator, represented by `##`, takes the string value of a parameter and concatenates it with another string. For example, consider the following macro:

```
#define DECLARE_MY_VAR(type) type my_##type
```

The purpose of this macro is to allow the user to specify a type (for example, `int`), and to automatically generate a variable declaration of that type whose name is `my_type`, where `type` is the parameter type. Here, we use the `##` operator to take the name of the type and concatenate it with the string `my_`. Thus, given the following macro call:

```
DECLARE_MY_VAR(int);
```

The preprocessor would replace it with the code

```
int my_int;
```

Note that when working with the token-pasting operator, if the result of the concatenation does not form a single C++ token (a valid operator or name), the behavior is undefined. For example, calling `DECLARE_MY_VAR(const int)` will have undefined behavior, since concatenating the strings `my_` and `const` does not yield a single string (the result is `const int my_const int`).

Practical Applications of the Preprocessor I: Diagnostic Debugging Functions

When writing a program, at times you may want to ensure that certain invariants about your program hold true – for example, that certain pointers cannot be `NULL`, that a value is always less than some constant, etc. While in many cases these conditions should be checked using a language feature called *exception handling*, in several cases it is acceptable to check them at runtime using a standard library macro called `assert`. `assert`, exported by the header `<cassert>`, is a macro that checks to see that some condition holds true. If so, the macro has no effect. Otherwise, it prints out the statement that did not evaluate to true, along with the file and line number in which it was written, then terminates the program. For example, consider the following code:

```
void MyFunction(int *myPtr)
{
    assert(myPtr != NULL);
    *myPtr = 137;
}
```

If a caller passes a null pointer into `MyFunction`, the `assert` statement will halt the program and print out a message that might look something like this:

```
Assertion Failed: 'myPtr != NULL': File: main.cpp, Line: 42
```

Because `assert` abruptly terminates the program without giving the rest of the application a chance to respond, you should not use `assert` as a general-purpose error-handling routine. In practical software development, as-

`assert` is usually used to express programmer assumptions about the state of execution. For example, assuming we have some enumerated type `Color`, suppose we want to write a function that returns whether a color is a primary color. Here's one possible implementation:

```
bool IsPrimaryColor(Color c)
{
    switch(c)
    {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Here, if the color is Red, Green, or Blue, we return that the color is indeed a primary color. Otherwise, we return that it is not a primary color. However, what happens if the parameter is not a valid `Color`, perhaps if the call is `IsPrimaryColor(Color(-1))`? In this function, since we assume that the parameter is indeed a color, we might want to indicate that to the program by explicitly putting in an `assert` test. Here's a modified version of the function, using `assert` and assuming the existence of a function `IsColor`:

```
bool IsPrimaryColor(Color c)
{
    assert(IsColor(c)); // We assume that this is really a color.
    switch(c)
    {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Now, if the caller passes in an invalid `Color`, the program will halt with an assertion error pointing us to the line that caused the problem. If we have a good debugger, we should be able to figure out which caller erroneously passed in an invalid `Color` and can better remedy the problem. Were we to ignore this case entirely, we might have considerably more trouble debugging the error, since we would have no indication of where the problem originated.

You should not, however, use `assert` to check that input from `GetLine` is correctly-formed, for example, since it makes far more sense to reprompt the user than to terminate the program.

While `assert` can be used to catch a good number of programmer errors during development, it has the unfortunate side-effect of slowing a program down at runtime because of the overhead of the extra checking involved. Consequently, most major compilers disable the `assert` macro in release or optimized builds. This may seem dangerous, since it eliminates checks for problematic input, but is actually not a problem because, in theory, you shouldn't be compiling a release build of your program if `assert` statements fail during execution.* Because `assert` is entirely disabled in optimized builds, you should use `assert` only to check that specific relations

* In practice, this isn't always the case. But it's still a nice theory!

hold true, never to check the return value of a function. If an `assert` contains a call to a function, when `assert` is disabled in release builds, the function won't be called, leading to different behavior in debug and release builds. This is a persistent source of debugging headaches.

Using the tools outlined in this chapter, it's possible for us to write our own version of the `assert` macro, which we'll call `CS106LAssert`, to see how to use the preprocessor to design such utilities. We'll split the work into two parts – a function called `DoCS106LAssert`, which actually performs the testing and error-printing, and the macro `CS106LAssert`, which will set up the parameters to this function appropriately. The `DoCS106LAssert` function will look like this:

```
#include <cstdlib> // for abort();

/* These parameters will be assumed to be correct. */
void DoCS106LAssert(bool invariant, string statement, string file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:      " << file << endl;
        cerr << "Line:      " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}
```

This function takes in the expression to evaluate, along with a string representation of that statement, the name of the file it is found in, and the line number of the initial expression. It then checks the invariant, and, if it fails, signals an error and quits the program by calling `abort()`. Since these parameters are rather bulky, we'll hide them behind the scenes by writing the `CS106LAssert` macro as follows:

```
#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)
```

This macro takes in a single parameter, an expression `expr`, and passes it in to the `DoCS106LAssert` function. To set up the second parameter to `DoCS106LAssert`, we get a string representation of the expression using the stringizing operator on `expr`. Finally, to get the file and line numbers, we use the special preprocessor symbols `__FILE__` and `__LINE__`. Note that since the macro is expanded at the call site, `__FILE__` and `__LINE__` refer to the file and line where the macro is used, not where it was declared.

To see `CS106LAssert` in action, suppose we make the following call to `CS106LAssert` in `myfile.cpp` at line 137. Given this code:

```
CS106LAssert(myPtr != NULL);
```

The macro expands out to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", __FILE__, __LINE__);
```

Which in turn expands to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", "myfile.cpp", 137);
```

Which is exactly what we want.

Now, suppose that we've used `CS106LAssert` throughout a C++ program and have successfully debugged many of its parts. In this case, we want to disable `CS106LAssert` for a release build, so that the final program

doesn't have the overhead of all the runtime checks. To allow the user to toggle whether `CS106LAssert` has any effect, we'll let them `#define` a constant, `NO_CS106L_ASSERT`, that disables the assertion. If the user does not define `NO_CS106L_ASSERT`, we'll use `#define` to have the `CS106LAssert` macro perform the runtime checks. Otherwise, we'll have the macro do nothing. This is easily accomplished using `#ifndef ... #else ... #endif` to determine the behavior of `CS106LAssert`. This smart version of `CS106LAssert` is shown below:

```
#ifndef NO_CS106L_ASSERT // Enable assertions

#include <cstdlib> // for abort();

/* Note that we define DoCS106LAssert inside this block, since if
 * the macro is disabled there's no reason to leave this function sitting
 * around.
 */
void DoCS106LAssert(bool invariant, string statement, string file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:  " << file << endl;
        cerr << "Line: " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}

#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)

#else // Disable assertions

/* Define CS106LAssert as a macro that expands to nothing.  Now, if we call
 * CS106LAssert in our code, it has absolutely no effect.
 */
#define CS106LAssert(expr) /* nothing */

#endif
```

Practical Applications of the Preprocessor II: The X Macro Trick

That macros give C++ programs access to their own source code can be used in other ways as well. One uncommon programming technique that uses the preprocessor is known as the *X Macro trick*, a way to specify data in one format but have it available in several formats.

Before exploring the X Macro trick, we need to cover how to redefine a macro after it has been declared. Just as you can define a macro by using `#define`, you can also undefine a macro using `#undef`. The `#undef` preprocessor directive takes in a symbol that has been previously `#defined` and causes the preprocessor to ignore the earlier definition. If the symbol was not already defined, the `#undef` directive has no effect but is not an error. For example, consider the following code snippet:

```
#define MY_INT 137
int x = MY_INT;    // MY_INT is replaced
#undef MY_INT;
int MY_INT = 42;   // MY_INT not replaced
```

The preprocessor will rewrite this code as

```
int x = 137;
int MY_INT = 42;
```

Although `MY_INT` was once a `#defined` constant, after encountering the `#undef` statement, the preprocessor stopped treating it as such. Thus, when encountering `int MY_INT = 42`, the preprocessor made no replacements and the code compiled as written.

To introduce the X Macro trick, let's consider a common programming problem and see how we should go about solving it. Suppose that we want to write a function that, given as an argument an enumerated type, returns the string representation of the enumerated value. For example, given the `enum`

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

We want to write a function called `ColorToString` that returns a string representation of the color. For example, passing in the constant `Red` should hand back the string `"Red"`, `Blue` should yield `"Blue"`, etc. Since the names of enumerated types are lost during compilation, we would normally implement this function using code similar to the following:

```
string ColorToString(Color c)
{
    switch(c)
    {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}
```

Now, suppose that we want to write a function that, given a color, returns the opposite color.* We'd need another function, like this one:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

These two functions will work correctly, and there's nothing functionally wrong with them as written. The problem, though, is that these functions are not *scalable*. If we want to introduce new colors, say, `White` and `Black`, we'd need to change both `ColorToString` and `GetOppositeColor` to incorporate these new colors. If we accidentally forget to change one of the functions, the compiler will give no warning that something is missing and we will only notice problems during debugging. The problem is that a color encapsulates more information than

* For the purposes of this example, we'll work with additive colors. Thus red is the opposite of cyan, yellow is the opposite of blue, etc.

can be expressed in an enumerated type. Colors also have names and opposites, but the C++ enum `Color` knows only a unique ID for each color and relies on correct implementations of `ColorToString` and `GetOppositeColor` for the other two. Somehow, we'd like to be able to group all of this information into one place. While we might be able to accomplish this using a set of C++ struct constants (e.g. defining a color struct and making const instances of these structs for each color), this approach can be bulky and tedious. Instead, we'll choose a different approach by using X Macros.

The idea behind X Macros is that we can store all of the information needed above inside of calls to preprocessor macros. In the case of a color, we need to store a color's name and opposite. Thus, let's suppose that we have some macro called `DEFINE_COLOR` that takes in two parameters corresponding to the name and opposite color. We next create a new file, which we'll call `color.h`, and fill it with calls to this `DEFINE_COLOR` macro that express all of the colors we know (let's ignore the fact that we haven't actually defined `DEFINE_COLOR` yet; we'll get there in a moment). This file looks like this:

File: color.h

```
DEFINE_COLOR(Red, Cyan)
DEFINE_COLOR(Cyan, Red)
DEFINE_COLOR(Green, Magenta)
DEFINE_COLOR(Magenta, Green)
DEFINE_COLOR(Blue, Yellow)
DEFINE_COLOR(Yellow, Blue)
```

Two things about this file should jump out at you. First, we haven't surrounded the file in the traditional `#ifndef ... #endif` boilerplate, so clients can `#include` this file multiple times. Second, we haven't provided an implementation for `DEFINE_COLOR`, so if a caller *does* include this file, it will cause a compile-time error. For now, don't worry about these problems – you'll see why we've structured the file this way in a moment.

Let's see how we can use the X Macro trick to rewrite `GetOppositeColor`, which for convenience is reprinted below:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

Here, each one of the `case` labels in this switch statement is written as something of the form

```
case color: return opposite;
```

Looking back at our `color.h` file, notice that each `DEFINE_COLOR` macro has the form `DEFINE_COLOR(color, opposite)`. This suggests that we could somehow convert each of these `DEFINE_COLOR` statements into case labels by crafting the proper `#define`. In our case, we'd want the `#define` to make the first parameter the argument of the `case` label and the second parameter the return value. We can thus write this `#define` as

```
#define DEFINE_COLOR(color, opposite) case color: return opposite;
```

Thus, we can rewrite `GetOppositeColor` using X Macros as

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        #include "color.h"
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

This is pretty cryptic, so let's walk through it one step at a time. First, let's simulate the preprocessor by replacing the line `#include "color.h"` with the full contents of `color.h`:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        DEFINE_COLOR(Red, Cyan)
        DEFINE_COLOR(Cyan, Red)
        DEFINE_COLOR(Green, Magenta)
        DEFINE_COLOR(Magenta, Green)
        DEFINE_COLOR(Blue, Yellow)
        DEFINE_COLOR(Yellow, Blue)
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Now, we replace each `DEFINE_COLOR` by instantiating the macro, which yields the following:

```
Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Finally, we `#undef` the `DEFINE_COLOR` macro, so that the next time we need to provide a definition for `DEFINE_COLOR`, we don't have to worry about conflicts with the existing declaration. Thus, the final code for `GetOppositeColor`, after expanding out the macros, yields

```

Color GetOppositeColor(Color c)
{
    switch(c)
    {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result.
    }
}

```

Which is exactly what we wanted.

The fundamental idea underlying the X Macros trick is that all of the information we can possibly need about a color is contained inside of the file `color.h`. To make that information available to the outside world, we embed all of this information into calls to some macro whose name and parameters are known. We do not, however, provide an implementation of this macro inside of `color.h` because we cannot anticipate every possible use of the information contained in this file. Instead, we expect that if another part of the code wants to use the information, it will provide its own implementation of the `DEFINE_COLOR` macro that extracts and formats the information. The basic idiom for accessing the information from these macros looks like this:

```

#define macroname(arguments) /* some use for the arguments */
#include "filename"
#undef macroname

```

Here, the first line defines the mechanism we will use to extract the data from the macros. The second includes the file containing the macros, which supplies the macro the data it needs to operate. The final step clears the macro so that the information is available to other callers. If you'll notice, the above technique for implementing `GetOppositeColor` follows this pattern precisely.

We can also use the above pattern to rewrite the `ColorToString` function. Note that inside of `ColorToString`, while we can ignore the second parameter to `DEFINE_COLOR`, the macro we define to extract the information still needs to have two parameters. To see how to implement `ColorToString`, let's first revisit our original implementation:

```

string ColorToString(Color c)
{
    switch(c)
    {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}

```

If you'll notice, each of the `case` labels is written as

```

case color: return "color";

```

Thus, using X Macros, we can write `ColorToString` as

```
string ColorToString(Color c)
{
    switch(c)
    {
        /* Convert something of the form DEFINE_COLOR(color, opposite)
         * into something of the form 'case color: return "color"';
         */
        #define DEFINE_COLOR(color, opposite) case color: return #color;
        #include "color.h"
        #undef DEFINE_COLOR

        default: return "<unknown>";
    }
}
```

In this particular implementation of `DEFINE_COLOR`, we use the stringizing operator to convert the `color` parameter into a string for the return value. We've used the preprocessor to generate both `GetOppositeColor` and `ColorToString`!

There is one final step we need to take, and that's to rewrite the initial `enum Color` using the X Macro trick. Otherwise, if we make any changes to `color.h`, perhaps renaming a color or introducing new colors, the `enum` will not reflect these changes and might result in compile-time errors. Let's revisit `enum Color`, which is reprinted below:

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

While in the previous examples of `ColorToString` and `GetOppositeColor` there was a reasonably obvious mapping between `DEFINE_COLOR` macros and `case` statements, it is less obvious how to generate this `enum` using the X Macro trick. However, if we rewrite this `enum` as follows:

```
enum Color {
Red,
Green,
Blue,
Cyan,
Magenta,
Yellow
};
```

It should be slightly easier to see how to write this `enum` in terms of X Macros. For each `DEFINE_COLOR` macro we provide, we'll simply extract the first parameter (the color name) and append a comma. In code, this looks like

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

This, in turn, expands out to


```
enum Color {
    #define DEFINE_COLOR(color, opposite) color,
    DEFINE_COLOR(Red, Cyan)
    DEFINE_COLOR(Cyan, Red)
    DEFINE_COLOR(Green, Magenta)
    DEFINE_COLOR(Magenta, Green)
    DEFINE_COLOR(Blue, Yellow)
    DEFINE_COLOR(Yellow, Blue)
    #undef DEFINE_COLOR
};
```

Which in turn becomes

```
enum Color {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
};
```

Which is exactly what we want. You may have noticed that there is a trailing comma at after the final color (`Yellow`), but this is not a problem – it turns out that it's totally legal C++ code.

Analysis of the X Macro Trick

The X Macro-generated functions have several advantages over the hand-written versions. First, the X macro trick makes the code considerably shorter. By relying on the preprocessor to perform the necessary expansions, we can express all of the necessary information for an object inside of an X Macro file and only need to write the syntax necessary to perform some task once. Second, and more importantly, this approach means that adding or removing `Color` values is simple. We simply need to add another `DEFINE_COLOR` definition to `color.h` and the changes will automatically appear in all of the relevant functions. Finally, if we need to incorporate more information into the `Color` object, we can store that information in one location and let any callers that need it access it without accidentally leaving one out.

That said, X Macros are not a perfect technique. The syntax is considerably trickier and denser than in the original implementation, and it's less clear to an outside reader how the code works. Remember that readable code is just as important as correct code, and make sure that you've considered all of your options before settling on X Macros. If you're ever working in a group and plan on using the X Macro trick, be sure that your other group members are up to speed on the technique and get their approval before using it.*

* The X Macro trick is a special case of a more general technique known as *preprocessor metaprogramming*. If you're interested in learning more about preprocessor metaprogramming, consider looking into the Boost Metaprogramming Library (MPL), a professional C++ package that simplifies common metaprogramming tasks.

More to Explore / Practice Problems

I've combined the “More to Explore” and “Practice Problems” sections because many of the topics we didn't cover in great detail in this chapter are best understood by playing around with the material. Here's a sampling of different preprocessor tricks and techniques, mixed in with some programming puzzles:

1. List three major differences between `#define` and the `const` keyword for defining named constants.
2. Give an example, besides preventing problems from `#include`-ing the same file twice, where `#ifdef` and `#ifndef` might be useful. (*Hint: What if you're working on a project that must run on Windows, Mac OS X, and Linux and want to use platform-specific features of each?*)
3. Write a regular C++ function called `Max` that returns the larger of two `int` values. Explain why it does not have the same problems as the macro `MAX` covered earlier in this chapter.
4. Give one advantage of the macro `MAX` over the function `Max` you wrote in the previous problem. (*Hint: What is the value of `Max(1.37, 1.24)`? What is the value of `MAX(1.37, 1.24)`?*)
5. The following C++ code is illegal because the `#if` directive cannot call functions:

```
bool IsPositive(int x)
{
    return x < 0;
}

#if IsPositive(MY_CONSTANT) // <-- Error occurs here
    #define result true
#else
    #define result false
#endif
```

Given your knowledge of how the preprocessor works, explain why this restriction exists. ♦

6. Compilers rarely inline recursive functions, even if they are explicitly marked `inline`. Why do you think this is?
7. Most of the STL algorithms are inlined. Considering the complexity of the implementation of `accumulate` from the chapter on STL algorithms, explain why this is.
8. A common but nonstandard variant of the `assert` macro is the `verify` macro which, like `assert`, checks a condition at runtime and prints an error and terminates if the condition is false. However, in optimized builds, `verify` is not disabled, but simply does not abort at runtime if the condition is false. This allows you to use `verify` to check the return value of functions directly (Do you see why?). Create a function called `CS106LVerify` that, unless the symbol `NO_CS106L_VERIFY` is defined, checks the parameter and aborts the program if it is false. Otherwise, if `NO_CS106L_VERIFY` is defined, check the condition, but do not terminate the program if it is false.
9. Another common debugging macro is a “not reached” macro which automatically terminates the program if executed. “Not reached” macros are useful inside constructs such as `switch` statements where the default label should never be encountered. Write a macro, `CS106LNotReached`, that takes in a string parameter and, if executed, prints out the string, file name, and line number, then calls `abort` to end the program. As with `CS106LAssert` and `CS106LVerify`, if the user `#defines` the symbol `NO_CS106L_NOTREACHED`, change the behavior of `CS106LNotReached` so that it has no effect. ♦
10. If you've done the two previous problems, you'll notice that we now have three constants, `NO_CS106L_ASSERT`, `NO_CS106L_VERIFY`, and `NO_CS106L_NOTREACHED`, that all must be `#defined` to disable them at runtime. This can be a hassle and could potentially be incorrect if we accidentally omit one of these symbols. Write a code snippet that checks to see if a symbol named `DISABLE_ALL_CS106L_DEBUG` is defined and, if so, disables all of the three aforementioned debugging tools. However, still give the user the option to selectively disable the individual functions.

11. Modify the earlier definition of `enum Color` such that after all of the colors defined in `color.h`, there is a special value, `NOT_A_COLOR`, that specifies a nonexistent color. (*Hint: Do you actually need to change `color.h` to do this?*) ♦
12. Using X Macros, write a function `StringToColor` which takes as a parameter a string and returns the `Color` object whose name *exactly* matches the input string. If there are no colors with that name, return `NOT_A_COLOR` as a sentinel. For example, calling `StringToColor("Green")` would return the value `Green`, but calling `StringToColor("green")` or `StringToColor("Olive")` should both return `NOT_A_COLOR`.
13. Suppose that you want to make sure that the enumerated values you've made for `Color` do not conflict with other enumerated types that might be introduced into your program. Modify the earlier definition of `DEFINE_COLOR` used to define `enum Color` so that all of the colors are prefaced with the identifier `eColor_`. For example, the old value `Red` should change to `eColor_Red`, the old `Blue` would be `eColor_Blue`, etc. Do not change the contents of `color.h`. (*Hint: Use one of the preprocessor string-manipulation operators*) ♦
14. The `#error` directive causes a compile-time error if the preprocessor encounters it. This may sound strange at first, but is an excellent way for detecting problems during preprocessing that might snowball into larger problems later in the code. For example, if code uses compiler-specific features (such as the OpenMP library), it might add a check to see that a compiler-specific `#define` is in place, using `#error` to report an error if it isn't. The syntax for `#error` is `#error message`, where **message** is a message to the user explaining the problem. Modify `color.h` so that if a caller `#includes` the file without first `#define`-ing the `DEFINE_COLOR` macro, the preprocessor reports an error containing a message about how to use the file.
15. Suppose that you are designing a control system for an autonomous vehicle in the spirit of the DARPA Grand Challenge. As part of its initialization process, the program needs to call a function named `InitCriticalInfrastructure()` to set up the car's sensor arrays. In order for the rest of the program to respond in the event that the startup fails, `InitCriticalInfrastructure()` returns a `bool` indicating whether or not it succeeded. To ensure that the function call succeeds, you check the return value of `InitCriticalInfrastructure()` as follows:

```
assert(InitCriticalInfrastructure());
```

During testing, your software behaves admirably and you manage to secure funding, fame, and prestige. You then compile your program in release mode, install it in a production car, and to your horror find that the car immediately drives off a cliff. Later analysis determines that the cause of the problem was that `InitCriticalInfrastructure` had not been called and that consequently the sensor array had failed to initialize.

Why did the release version of the program not call `InitCriticalInfrastructure`? How would you rewrite the code that checks for an error so that this problem doesn't occur?

16. If you're up for a challenge, consider the following problem. Below is a table summarizing various units of length:

Unit Name	#meters / unit	Suffix	System
Meter	1.0	m	Metric
Centimeter	0.01	cm	Metric
Kilometer	1000.0	km	Metric
Foot	0.3048	ft	English
Inch	0.0254	in	English
Mile	1609.344	mi	English
Astronomical Unit	1.496×10^{11}	AU	Astronomical
Light Year	9.461×10^{15}	ly	Astronomical
Cubit*	0.55	cubit	Archaic

- Create a file called `units.h` that uses the X macro trick to encode the above table as calls to a macro `DEFINE_UNIT`. For example, one entry might be `DEFINE_UNIT(Meter, 1.0, m, Metric)`.
- Create an enumerated type, `LengthUnit`, which uses the suffix of the unit, preceded by `eLengthUnit_`, as the name for the unit. For example, a cubit is `eLengthUnit_cubit`, while a mile would be `eLengthUnit_mi`. Also define an enumerated value `eLengthUnit_ERROR` that serves as a sentinel indicating that the value is invalid.
- Write a function called `SuffixStringToLengthUnit` that accepts a string representation of a suffix and returns the `LengthUnit` corresponding to that string. If the string does not match the suffix, return `eLengthUnit_ERROR`.
- Create a struct, `Length`, that stores a double and a `LengthUnit`. Write a function `ReadLength` that prompts the user for a double and a string representing an amount and a unit suffix and stores data in a `Length`. If the string does not correspond to a suffix, reprompt the user. You can modify the code for `GetInteger` from the chapter on streams to make an implementation of `GetReal`.
- Create a function, `GetUnitType`, that takes in a `Length` and returns the unit system in which it occurs (as a string)
- Create a function, `PrintLength`, that prints out a `Length` in the format ***amount suffix (amount unitnames)***. For example, if a `Length` stores 104.2 miles, it would print out `104.2mi (104.2 Miles)`
- Create a function, `ConvertToMeters`, which takes in a `Length` and converts it to an equivalent length in meters.

Surprisingly, this problem is not particularly long – the main challenge is the user input, not the unit management!

* There is no agreed-upon standard for this unit, so this is an approximate average of the various lengths.

Chapter 14: Introduction to Templates

Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures.

– Alex Stepanov, inventor of the STL. [Ste07]

One of the most important lessons an upstart computer scientist or software engineer can learn is *decomposition* or *factoring* – breaking problems down into smaller and smaller pieces and solving each subproblem separately. At the heart of decomposition is the concept of *generality* – code should avoid overspecializing on a single problem and should be robust enough to adapt to other situations. Take as an example the STL. Rather than specializing the STL container classes on a single type, the authors decided to parameterize the containers over the types they store. This means that the code written for the `vector` class can be used to store almost any type, and the `map` can use arbitrary types as key/value pairs. Similarly, the STL algorithms were designed to operate on all types of iterators rather than on specific container classes, making them flexible and adaptable.

The STL is an excellent example of how versatile, flexible, and powerful C++ templates can be. In C++ a *template* is just that – a code pattern that can be instantiated to produce a type or function that works on an arbitrary type. Up to this point you've primarily been a client of template code, and now it's time to gear up to write your own templates. In this chapter we'll cover the basics of templates and give a quick tour of how template classes and functions operate under the hood. We will make extensive use of templates later in this course and especially in the extended examples, and hopefully by the time you've finished this course reader you'll have an appreciation for just how versatile templates can be.

Defining a Template Class

Once you've decided that the class you're writing is best parameterized over some arbitrary type, you can indicate to C++ that you're defining a template class by using the `template` keyword and specifying what types the template should be parameterized over. For example, suppose that we want to define our own version of the `pair` struct used by the STL. If we want to call this struct `MyPair` and have it be parameterized over two types, we can write the following:

```
template <typename FirstType, typename SecondType> struct MyPair
{
    FirstType first;
    SecondType second;
};
```

Here, the syntax `template <typename FirstType, typename SecondType>` indicates to C++ that what follows is a template that is parameterized over two types, one called `FirstType` and one called `SecondType`. The actual names of the parameters are unimportant as far as clients are concerned, much in the same way that the actual names of parameters to functions are unimportant. For example, the above definition is functionally equivalent to this one below:

```
template <typename One, typename Two> struct MyPair
{
    One first;
    Two second;
};
```

Within the body of the template struct, we can use the names `One` and `Two` (or `FirstType` and `SecondType`) to refer to the types that the client specifies when she instantiates `MyPair`.

Notice that when specifying the template arguments we used the `typename` keyword to indicate that the parameter should be a type, such as `int` or `vector<double>`. In some template code you will see the keyword `class` substituted for `typename`, as in this example:

```
template <class FirstType, class SecondType> struct MyPair
{
    FirstType first;
    SecondType second;
};
```

In this instance, `typename` and `class` are completely equivalent to one another. However, I find the use of `class` misleading because it incorrectly implies that the parameter must be a class type. This is not the case – you can still instantiate templates that are parameterized using `class` with primitive types like `int` or `double`. Thus for the remainder of this course reader we will use `typename` instead of the `class`.*

To create an instance of `MyPair` specialized over any particular types, we use the (hopefully familiar) syntax below:

```
MyPair<int, string> one; // A pair of an int and a string.
one.first = 137;
one.second = "Templates are cool!";
```

Classes and structs are closely related to one another, so unsurprisingly the syntax for declaring a template class is similar to that for a template struct. Let's suppose that we want to convert our `MyPair` struct into a class with full encapsulation (i.e. with accessor methods and constructors instead of exposed data members). Then we would begin by declaring `MyPair` as

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    /* ... */

private:
    FirstType first;
    SecondType second;
};
```

Now, what sorts of functions should we define for our `MyPair` class? Ideally, we'd like to have some way of accessing the elements stored in the pair, so we'll define a pair of functions `getFirst` and `setFirst` along with an equivalent `getSecond` and `setSecond`. This is shown here:

* You can only substitute `class` for `typename` in this instance – it's illegal to declare a regular C++ class using the `typename` keyword.

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};
```

Notice that we're using the template arguments `FirstType` and `SecondType` to stand for whatever types the client parameterizes `MyPair` over. We don't need to indicate that `FirstType` and `SecondType` are at all different from other types like `int` or `string`, since the C++ compiler already knows that from the template declaration. In fact, with a few minor restrictions, once you've defined a template argument, you can use it anywhere that an actual type could be used and C++ will understand what you mean.

Now that we've declared these functions, we should go about implementing them in the intuitive way. If `MyPair` were not a template class, we could write the following:

```
/* Note: This is not legal code! */
FirstType MyPair::getFirst()
{
    return first;
}
```

The problem with this above code is that `MyPair` is a class *template*, not an actual class. If we don't tell C++ that we're trying to implement a member function for a template class, the compiler won't understand what we mean. Thus the proper way to implement this member function is

```
template <typename FirstType, typename SecondType>
FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}
```

Here, we've explicitly prefaced the implementation of `getFirst` with a template declaration and we've marked that the member function we're implementing is for `MyPair<FirstType, SecondType>`. The template declaration is necessary for C++ to figure out what `FirstType` and `SecondType` mean here, since without this information the compiler would think that `FirstType` and `SecondType` were actual types instead of placeholders for types. That we've mentioned this function is available inside `MyPair<FirstType, SecondType>` instead of just `MyPair` is also mandatory since there is no real `MyPair` class – after all, `MyPair` is a class *template*, not an actual class.

The other member functions can be implemented similarly. For example, here's an implementation of `setSecond`:

```
template <typename FirstType, typename SecondType>
void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
{
    second = newValue;
}
```

When implementing member functions for template classes, you do *not* need to repeat the template definition if you define the function inside the body of the template class. Thus the following code is perfectly legal:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst()
    {
        return first;
    }
    void setFirst(FirstType newValue)
    {
        first = newValue;
    }

    SecondType getSecond()
    {
        return second;
    }
    void setSecond(SecondType newValue)
    {
        second = newValue;
    }
private:
    FirstType first;
    SecondType second;
};
```

Now, let's suppose that we want to define a member function called `swap` which accepts as input a reference to another `MyPair` class, then swaps the elements in that `MyPair` with the elements in the receiver object. Then we can prototype the function like this:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst()
    {
        return first;
    }
    void setFirst(FirstType newValue)
    {
        first = newValue;
    }

    SecondType getSecond()
    {
        return second;
    }
    void setSecond(SecondType newValue)
    {
        second = newValue;
    }

    void swap(MyPair& other);
private:
    FirstType first;
    SecondType second;
};
```


Even though `MyPair` is a template class parameterized over two arguments, inside the body of the `MyPair` template class definition we can use the name `MyPair` without mentioning that it's a `MyPair<FirstType, SecondType>`. This is perfectly legal C++ and will come up more when we begin discussing copying behavior in a few chapters. The actual implementation of `swap` is left as an exercise.

.h and .cpp files for template classes

When writing a C++ class, you normally partition the class into two files: a `.h` file containing the declaration and a `.cpp` file containing the implementation. The C++ compiler can then compile the code contained in the `.cpp` file and then link it into the rest of the program when needed. When writing a template class, however, breaking up the definition like this will cause linker errors. The reason is that C++ templates are just that – they're *templates* for C++ code. Whenever you write code that instantiates a template class, C++ generates code for the particular instance of the class by replacing all references to the template parameters with the arguments to the template. For example, with the `MyPair` class defined above, if we create a `MyPair<int, string>`, the compiler will generate code internally that looks like this:

```
class MyPair<int, string>
{
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);
private:
    int first;
    string second;
}

int MyPair<int, string>::getFirst()
{
    return first;
}

void MyPair<int, string>::setFirst(int newValue)
{
    first = newValue;
}

string MyPair<int, string>::getSecond()
{
    return second;
}

void MyPair<int, string>::setSecond(string newValue)
{
    second = newValue;
}
```

At this point, compilation continues as usual.

But what would happen if the compiler didn't have access to the implementation of the `MyPair` class? That is, let's suppose that we've created a header file, `my-pair.h`, that contains only the class declaration for `MyPair`, as shown here:

File: *my-pair.h*

```
#ifndef MyPair_Included // Include guard prevents multiple inclusions
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#endif
```

Suppose that we have a file that `#includes` the `my-pair.h` file and then tries to use the `MyPair` class. Since all that the compiler has seen of `MyPair` is the above class definition, the compiler will only generate the following code for `MyPair`:

```
class MyPair<int, string>
{
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);
private:
    int first;
    string second;
}
```

Notice that while all the member functions of `MyPair<int, string>` have been *prototyped*, they haven't been *implemented* because the compiler didn't have access to the implementations of each of these member functions. In other words, if a template class is instantiated and the compiler hasn't seen implementations of its member functions, the resulting template class will have no code for its member functions. This means that the program won't link, and our template class is now useless.

When writing a template class for use in multiple files, the entire class definition, including implementations of member functions, must be visible in the header file. One way of doing this is to create a `.h` file for the template class that contains both the class definition and implementation without creating a matching `.cpp` file. This is the approach adopted by the C++ standard library; if you open up any of the headers for the STL, you'll find the complete (and cryptic) implementations of all of the functions and classes exported by those headers.

To give a concrete example of this approach, here's what the `my-pair.h` header file might look like if it contained both the class and its implementation:

File: *my-pair.h*

```

/* This method of packaging the .h/.cpp pair puts the entire class definition and
 * implementation into the .h file. There is no .cpp file for this header.
 */

#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}

template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
{
    second = newValue;
}

#endif

```

Putting the class and its definition inside the header file is a valid way to prevent linker errors, but it seems to violate the principle of separation of interface and implementation. After all, the reason we have both .h and .cpp files is to hide a class implementation in a file that clients never have to look at. Putting the entire implementation into the .h file thus seems misguided. Is there some way to maintain the traditional .h/.cpp split with template classes? Yes, though you might want to buckle up, since this next section is rather technical.

The trick to splitting a template class into a .h/.cpp pair is to write the two files, then to `#include` the .cpp file at the end of the .h file. Since `#include`-ing the .cpp file at the end of the .h file splices the contents of the .cpp file into the .h file, this approach is functionally equivalent to writing one large .h file. However, if you do split the file up into two parts, you should make sure to *not include the .cpp file in the list of files to compile*. The

reason is that `#include`-ing the `.cpp` file inside the `.h` file leads to a circular dependency. To see how this happens, suppose that we partition the code into a `.h` file and a `.cpp` file as shown here:

File: my-pair.h

```
/* This method of packaging the .h/.cpp pair puts the class definition into the .h
 * file and implementations into the .cpp file, then #includes the .cpp file into
 * the .h file.
 */

#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#include "my-pair.cpp" // Import the implementation into the header

#endif
```

File: my-pair.cpp

```
#include "my-pair.h" // This is problematic, see below.

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}

template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
{
    second = newValue;
}
```

Let's suppose that the compiler now tries to compile `my-pair.cpp`. The first thing it does is try to `#include my-pair.h` into the file, resulting in the following code (with the comments removed)

```
#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#include "my-pair.cpp"

#endif

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}

template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue)
{
    second = newValue;
}
```

Now, the compiler will try to `#include my-pair.cpp`, which splices the original contents of the file *back into itself*. This results in the following code:

```

#ifndef MyPair_Included
#define MyPair_Included
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#include "my-pair.h"
template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}
template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond( SecondType newValue)
{
    second = newValue;
}

#endif

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}
template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond( SecondType newValue)
{
    second = newValue;
}

```

Next, the preprocessor will `#include "my-pair.h"` again. Fortunately, we've surrounded `my-pair.h` in an include guard, so nothing happens. After applying the effects of the `#ifndef` and `#define` directives, we end up with the following code getting handed off to the rest of the compiler:

```
template <typename FirstType, typename SecondType> class MyPair
{
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue)
{
    first = newValue;
}
template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond()
{
    return second;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond( SecondType newValue)
{
    second = newValue;
}

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst() // Error: redefinition
{
    return first;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue) // Error
{
    first = newValue;
}
template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond() // Error
{
    return second;
}
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond( SecondType newValue) // Error
{
    second = newValue;
}
```

If you'll notice, the contents of our .cpp file appear here *twice* – once from the original .cpp file and once because the header file `#included` the source file back into itself. When the compiler looks at this code, it will generate a compile-time error because we've provided two definitions for each member function and the compiler can't tell which one to use.

That was quite an complex chain of events culminating in a compile-time error, and everything seemed to stem from the fact that the .h and .cpp files each `#include` each other. We know that the .h file has to `#include` the .cpp file, but does the .cpp file have to `#include` the .h? The answer is no, *unless the .cpp file ends up getting compiled*. If that happens and the .h file isn't `#included`, then the compiler will trip up over the definitions of member functions for a template class it hasn't seen yet. In short, if we split the class into a .h/.cpp pair, we have to be very careful that the .cpp file isn't compiled along with the rest of our source code. Otherwise, we'll get a nasty set of compiler errors. Fortunately, there's a cute preprocessor trick we can use to let the .cpp file be compiled along with other files in the project. Let's review our current situation. We want the contents of the .cpp file to be available to the .h file, since clients of the template class need to have the full template definition available, but we *don't* want those contents to be visible if we just try compiling the .cpp file by itself. The trick is to surround the .cpp file in a `#ifdef ... #endif` block checking whether a symbol particular to the .h file is defined. If we try compiling the .cpp file directly, this symbol won't be defined and the code won't be compiled. Inside the .h file, however, we can surround the directive to `#include` the source file with a `#define` and `#undef` pair to define this particular symbol, include the .cpp file, then undefine the symbol. This makes the symbol available to the .cpp file, and it will be spliced into the header correctly. This version of the code looks as follows:

File: my-pair.h

```
/* This method of packaging the .h/.cpp pair puts the class definition into the .h
 * file and implementations into the .cpp file, then #includes the .cpp file into
 * the .h file.
 */

#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair
{
    /* ... omitted for brevity ... */
};

#define UseMyPairCpp    // Make symbol available here...
#include "my-pair.cpp"
#undef  UseMyPairCpp    // ... but nowhere else

#endif
```

File: my-pair.cpp

```
#ifdef UseMyPairCpp // Only compile this file if it's being used by the .h file

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst()
{
    return first;
}

/* ... etc. ... */

#endif
```


We now have two ways of packaging a template class in a header file: put the entire template definition inside the header file, or partition it into a .h/.cpp pair with a bit of preprocessor machinery. Given the choice between these two options, I personally find the monolithic .h file approach much more appealing, though you're free to use whichever you prefer.

Template Functions

As you saw with the STL, templates come in two forms – class templates, as described above, and function templates. The function templates you're most familiar with right now are the STL algorithms, which can accept any form of iterator you provide as an argument, whether raw C++ pointers, `vector<int>::iterators`, or `ostream_iterators`. While function templates are in many ways similar to class templates, they differ significantly in a few aspects and thus we'll devote some time to their nuances.

Let's suppose that we want to write a template function called `MakeIntoVector` which accepts as input an element of any type and a length, then returns a `vector` consisting of the specified number of copies of that element. This is admittedly a simple function, but it helps illustrate some of the important points of writing a template function.

Since our `MakeIntoVector` function can accept an object of any type, we'll make `MakeIntoVector` a template function parameterized over the type of the parameter. One possible implementation is as follows:

```
template <typename ElemType>
vector<ElemType> MakeIntoVector(ElemType e, int numCopies)
{
    vector<ElemType> result(numCopies, e);
    return result;
}
```

Like a template class, the template function begins with a `template` declaration indicating what the arguments to the template are. We then write the function under the assumption that `ElemType` is a valid type, and end up with a very concise function. Had we wanted to parameterize the function over more types, we could have added them to the template argument list.

Now that we've written `MakeIntoVector`, we can call the function as follows:

```
cout << "Enter string: ";
string line = GetLine();
vector<string> v = MakeIntoVector(line, 137);
```

Or to be more concise:

```
cout << "Enter string: ";
vector<string> v = MakeIntoVector(GetLine(), 137);
```

Notice that although `MakeIntoVector` is a template function, we didn't need to specify that we were calling `MakeIntoVector<string>`. Instead, the C++ compiler was able to infer that, since the first argument to the function has type `string`, the template parameter must be a `string`. This is known as *type inference* and is one of the more powerful features of the C++ template system.

Why doesn't C++ have support for automatically inferring the type of a class template? That is, why can't we just write

```
vector v = MakeIntoVector(GetLine(), 137);
```

Instead of

```
vector<string> v = MakeIntoVector(GetLine(), 137);
```

The answer has to do with what information C++ has available to it at the time that it needs to instantiate the template. When working with template functions, C++ has full knowledge of the types of its arguments and thus can try to deduce precisely what the template arguments must be. When working with template classes, however, it is not always possible to determine the template arguments from context. For example, suppose we were to write

```
vector v; // Hypothetical only; not legal C++ code
```

What type should we parameterize `v` over? There are no clues from context, so we cannot determine its type here.

Because function templates have type inference while class templates do not, it is common to find template classes paired with template functions that produce instances of the template class. One example you've already seen is the `make_pair` template function which accepts two arguments and produces a `pair` of the proper type. A possible implementation of `make_pair` is shown below:

```
template <typename First, typename Second>
pair<First, Second> make_pair(const First& one, const Second& two)
{
    pair<First, Second> result(one, two);
    return result;
}
```

If we call this `make_pair` function, C++ can deduce what types `First` and `Second` must be and can construct a `pair` with those arguments. If we only need the `pair` as a parameter to a function (for example, the STL `map`'s `insert` member function), using `make_pair` can save us some typing by automatically computing the type of the arguments. We'll see this technique used later when we talk about the STL `<functional>` library.

In some cases when you're writing template functions C++ won't be able to figure out what types the template is parameterized over and you'll have to specify them manually. As an example, suppose that we want to write a template function that, given a string, converts the contents of that string to a particular type; that is, this function is a generalization of the `StringToInteger` and `StringToReal` functions from `strutils.h`. We'll call this function `lexical_cast` in the spirit of C++'s casting operators.* One possible prototype is as follows:

```
template <typename TargetType> TargetType lexical_cast (string toConvert)
{
    /* ... see practice problems ... */
}
```

Notice that this function is parameterized over a type called `TargetType` that appears nowhere in the function's argument list. Template argument type inference only applies to function arguments, and C++ will not infer the types of template parameters that appear only in return values. For example, this code will not compile:

```
int myInteger = lexical_cast("137"); // Error - what is TargetType?
```

Why is there a distinction between arguments and return values? After all, can't the compiler look at this code and realize that the return value should be an `int`? Unfortunately, the answer is no. Consider this related code:

* C++ has four casting operators – `static_cast`, `const_cast`, `reinterpret_cast`, and `dynamic_cast`. These will be covered later. The inspiration for the name `lexical_cast` comes from the Boost library's template function of the same name.

```
int myInteger = lexical_cast("2.71828"); // Still illegal
```

Here, we're using `lexical_cast` to try to convert a string representation of 2.71828 into an integer. If C++ uses `lexical_cast` specialized over `double`, this code will work correctly (albeit with truncation) and set `myInteger` to 2. But if C++ chooses to use `lexical_cast` specialized over `int`, the code might cause an error as `lexical_cast` realizes that 2.71828 isn't an `int`. The problem is that there are many possible types for the return value, and anything that can be implicitly converted to an `int` is a candidate. Rather than guessing, in situations like these the compiler requires you to explicitly indicate the template arguments to the function as you would in a template class. Thus the correct version of the above code is

```
int myInteger = lexical_cast<int>("137");
```

Implicit Interfaces

Suppose that we're interested in writing a function which returns the maximum of two `int` values. Then we could write a function like this:*

```
int MaxInt(int one, int two)
{
    return one < two ? two : one;
}
```

Similarly, we could write a function which returns the maximum of two `doubles` as

```
double MaxDouble(double one, double two)
{
    return one < two ? two : one;
}
```

We could also write a function that returns which of two `strings` alphabetically follows the other as

```
string MaxString(string one, string two)
{
    return one < two ? two : one;
}
```

Notice that every single one of these functions has the following form:

```
Type Max(Type one, Type two)
{
    return one < two ? two : one;
}
```

This suggests that there is some way to unify all of these functions together into a single template function. Indeed there is, and one possible implementation is as follows:

```
template <typename T> T Max(T one, T two)
{
    return one < two ? two : one;
}
```

Now, to get the maximum of two `doubles` we could call `Max(1.0, 3.0)`, and to get the maximum of two `ints` we could call `Max(137, 2718)`.

* In case you're not familiar with the `?:` operator, see the chapter on the preprocessor for more info.

An interesting observation is that the meaning of the `one < two` depends on what type the user passes into the `Max` function. The `<` might be an integer comparison, a floating-point comparison, or even a lexicographic comparison in the case of `strings`. Provided that it's meaningful to have two objects of the specified type compared by the `<` operator, we can instantiate `Max` over any arguments. But what if we try to invoke `Max` on something that cannot be compared with the less-than operator, such as this struct?

```
struct MyStruct
{
    int x;
    double y;
};
```

In this case, C++ will not somehow “figure out” how to compare two objects of the `MyStruct` type, nor will it compile the program into something that crashes or results in undefined behavior. Instead, the program will generate a compile-time error because C++ recognizes that there is no way to compare two `MyStructs` using the less-than operator. Unfortunately, this might not be the prettiest of error messages; when compiling this exact code on Visual Studio 2005 I received this error message:

```
c:\...\main.cpp(3) : error C2676: binary '<' : 'MyStruct' does not define this
operator or a conversion to a type acceptable to the
predefined operator
c:\...\main.cpp(15) : see reference to function template instantiation
'T Max<MyStruct>(T,T)' being compiled
with
[
    T=MyStruct
]
```

The compiler correctly reports that the error is that two `MyStructs` cannot be compared using the `<` operator, but does so in a roundabout, convoluted way. Notice the key phrase “function template instantiation being compiled” here – any time you encounter an error like this, be sure to look at what the template arguments are and where the actual template is. This is where you'll encounter the error.

If you'll notice, we've written a template function that at first glance can accept objects of any type but which can only be successfully instantiated over types that have a well-defined `<` operator. This is perfectly acceptable C++ programming and in fact templates are often used to define functions that operate over any type meeting some set of restrictions. These restrictions are sometimes called an *implicit interface* (as opposed to *explicit interfaces* defined by inheritance; see the chapter on inheritance for more information).

Although our `Max` function is guaranteed to be safe (i.e. it will never compile when the elements can't be compared using `<`), there is nothing in the function signature to explicitly indicate what restrictions apply to the template arguments. To communicate these requirements to other coders, C++ programmers commonly choose names for the template argument that make these requirements more clear. For example, since the template argument to `Max` must be comparable using less-than, we might rewrite the function like this:

```
template <typename LessThanComparable>
LessThanComparable Max(LessThanComparable one, LessThanComparable two)
{
    return one < two ? two : one;
}
```

Naming the template argument `LessThanComparable` might help communicate these restrictions to other programmers, but otherwise has no effect on the C++ program. In fact, it's just as effective as naming a parameter to a function `mustBeNonnegative` – it might help programmers figure out what your intentions are, but the compiler won't enforce it.

Template Member Functions

In addition to template classes and template functions, you can define template *member* functions inside a class. The class itself need not be a template class to contain a template member function, though it is often the case. For example, suppose that we're writing a class that acts as a wrapper around the streams library to simplify file reading and writing. We're interested in exporting a function called `mapAll` which iterates over the characters in the file and calls a user-specified callback on each of them. For example, given the following function:

```
void AppendToString(char ch, string& out)
{
    out += ch;
}
```

If the class we're designing is called `FileWrapper`, we'd hope to be able to do the following:

```
FileWrapper fw("my-file.txt");
string text;
fw.mapAll(AppendToString, text);
```

Here, the `mapAll` function accepts two parameters, the function to call and some user-supplied data to pass into the function, then calls the function once per character read passing in the user-supplied data.* Since the user could theoretically supply auxiliary data they chose, we should make `mapAll` a template function parameterized over the type of the user-supplied data. This is shown here:

```
class FileWrapper
{
public:
    /* ... miscellaneous functions ... */

    template <typename AuxType>
        void mapAll(void function(char, AuxType&), AuxType& auxData)
};
```

As with member functions of template classes, when implementing template member functions you must explicitly mark that the function is a template and should be sure to put it in its proper `.h` file instead of in the `.cpp` file. For example, here's how we'd begin implementing `mapAll`:

```
template <typename AuxType>
void FileWrapper::mapAll(void function(char, AuxType&), AuxType& auxData)
{
    /* ... */
}
```

Now, suppose that we want to implement a similar function, except inside of the `CS106B/X Vector`. Since the `Vector` itself is a template class, the syntax for implementing the function is slightly different. The function declaration itself is similar to the declaration in the `FileWrapper` class:

```
template <typename ElemType> class Vector
{
public:
    /* ... */
    template <typename AuxData>
        void mapAll(void function(ElemType, AuxData&), AuxData& auxData);
};
```

* When we talk about functors you'll learn a much better way to do this. For now, though, this approach is perfectly fine.

When implementing the `mapAll` function, however, we have to use *two* template declarations – one to indicate that the `Vector` class is a template and one to indicate that `mapAll` is a template. For example:

```
template <typename ElemType>
    template <typename AuxData>
        void Vector<ElemType>::void mapAll(void function(ElemType, AuxData&),
                                           AuxData& auxData)
{
    /* ... */
}
```

That's quite a mouthful, and admittedly the syntax is a bit clunky, but we're now set up to implement the template member function of a template class.

The Two Meanings of `typename`

There is an unfortunate quirk in the C++ language that shows up when trying to access types defined inside of template classes. Suppose that we want to write a function accepts as input an arbitrary STL container class containing ints (`vector<int>`, `set<int>`, etc.) and returns the number of elements in the container that are even. Since the function needs to accept any valid STL container, we write the function as a template, as shown here:

```
/* Watch out! This code doesn't compile! */
template <typename ContainerType> int CountEvenElements(ContainerType& c)
{
    int result = 0;

    /* Iterate over the container counting evens. */
    for(ContainerType::iterator itr = c.begin(); itr != c.end(); ++itr)
        if(*itr % 2 == 0) ++result;

    return result;
}
```

Initially this function looks pretty straightforward – we simply use an iterator loop to walk over the elements of the container, incrementing a counter as we go, and finally return the counter. Unfortunately, this isn't legal C++. Due to a rather obscure technical reason, when inside the body of the template, C++ can't necessarily tell whether `ContainerType::iterator` is a type called `iterator` nested inside the type `ContainerType` or a class constant of `ContainerType` called `iterator`. We therefore need to tell the C++ compiler that `ContainerType::iterator` is indeed the name of a nested type by using the `typename` keyword. The revised code is as follows:

```
template <typename ContainerType> int CountEvenElements(ContainerType& c)
{
    int result = 0;

    /* Iterate over the container counting evens. */
    for(typename ContainerType::iterator itr = c.begin(); itr != c.end(); ++itr)
        if(*itr % 2 == 0) ++result;

    return result;
}
```

Notice that we've put `typename` in front of `ContainerType::iterator` to tell C++ that `iterator` is a nested type. This is another use of the `typename` keyword that crops up surprisingly often when writing template code. If you ever need to access the name of a type nested inside a template class, make sure to preface the type with `typename` or your code will not compile.

Be aware that you only have to preface the name of a type with `typename` if you are writing a template and need to access an inner class of a type that depends on a template argument. Thus you don't need to (and can't legally) put `typename` in front of `vector<int>::iterator` since `vector<int>::iterator` is a complete type. You would, however, have to put `typename` in front of `vector<T>::iterator` if you were implementing a template parameterized over `T`. This is an arcane rule that complicates template code, but fortunately compilers these days are very good at diagnosing missing `typename`s. In fact, the Linux compiler `g++` will actually give an error message that explicitly tells you to insert the missing `typename`.

More to Explore

We've covered a great deal of template syntax and usage in this chapter, but we've barely scratched the surface when it comes to templates. Modern C++ programming relies extensively on clever and subtle uses of templates, and there are many advanced techniques involving templates that are far beyond the scope of this course reader. While we will periodically revisit templates, there simply isn't enough room to cover them in their entirety. If you're interested in learning more about templates, consider looking into the following:

1. **Non-type Template Arguments.** All of the template code you've seen so far is parameterized over variable types, such as a `vector<int>` or a `map<string, string>`. However, it's possible to parameterize C++ templates over other things as well, such as `ints`, function pointers, or even other templates. While non-type template arguments are uncommon, they are at times quite useful and there are many examples where they are an excellent solution to a difficult problem. Consult a reference for more information. We will see one example of non-type template arguments in a later chapter.
2. **Traits Classes.** When writing template functions or classes, sometimes you will need additional information about the structure of a particular template argument. For example, when writing an STL algorithm that works on an arbitrary iterator range, you might need temporary storage space to hold elements of the type being iterated over. Given the type of an iterator, how can you determine what type of value it dereferences to? Using a technique called *traits classes*, it's possible to inspect template arguments to gain additional information about them. In the case of STL iterators, the C++ standard library provides a template class called `iterator_traits` that, when instantiated over a particular type, exports information about that type, such as its iterator category, the type it iterates over, etc. Traits classes are becoming increasingly more relevant in generic programming, and if you plan on using templates in an industrial setting you will almost certainly encounter them.
3. **Template Specialization.** It is possible to define a specific instance of a template class to use instead of the default version in certain circumstances. For example, when implementing a class akin to the STL `set`, you might be able to improve memory usage when storing a `set<char>` by taking advantage of the fact that there are only 256 possible values for a `char`. Template specialization can be used to optimize code for a particular application, to improve the accuracy of traits classes, or for far more creative purposes (see the next item on this list). If you plan on pursuing template programming more seriously, be sure to look into template specialization.
4. **Template Metaprogramming.** Using a combination of the techniques listed above, it is possible to write C++ code containing *template metaprograms*. A template metaprogram is a collection of templates and template specializations such that when one of the templates is instantiated, it fires off chains of further instantiations whose end result produces new C++ code. For example, it's possible to write a template metaprogram to compute *at compile time* whether a number is prime, or even to automatically generate a sorting routine optimized for data of a particular size. It has been proven that template metaprogramming is a Turing-complete proper subset of C++ that executes at compile-time, meaning that template metaprograms have the same computational power as the rest of the C++ language except that they execute inside the compiler. Code using template metaprogramming is notoriously difficult to read, but the sheer firepower and performance gains offered by template metaprogramming cannot be understated. If you are up for a true challenge, investigate template metaprogramming in more detail. You will not be disappointed.

Practice Problems

1. The STL algorithms are implemented as template functions parameterized over the types of all of their arguments. This allows them to operate over any types of arguments, provided of course that the arguments make sense. For example, consider the `transform` algorithm, which accepts four parameters – two iterators defining an input range, and iterator defining the start of an output range, and a transformation function – then applies the transformation function to all elements in the source range and stores them in the output range. One possible implementation of `transform` is as follows:

```
template <typename InputIterator, typename OutputIterator, typename Function>
inline OutputIterator transform(InputIterator start,
                               InputIterator end,
                               OutputIterator where,
                               Function fn)
{
    while(start != end)
    {
        *where = fn(*start);
        ++start;
        ++where;
    }
    return where; // Return iterator one past the end of the written range
}
```

Here, the body of the function simply walks over the range from `start` to `end`, at each point applying the transformation function and storing the result in the output range the starts with `where`. By convention, the algorithm returns an iterator one past the end of the range written.

Using `transform` as a reference, implement an STL algorithm-like function `copy_if` which accepts as input three iterators – two delineating an input range and one delineating an output range – along with a predicate function, then copies the elements in the input range for which the predicate is true to the output. For example, if `v` contains 0, 1, 2, 3, 4, 5 and if `IsEven` returns whether an integer is even, then

```
copy_if(v.begin(), v.end(), ostream_iterator<int>(cout, " "), IsEven);
```

should print out 0 2 4. If you've templated the function correctly, the fact that one of the parameters is an `ostream_iterator<int>` shouldn't be a problem. ♦

2. One particularly useful STL algorithm is `equal`, which is defined as follows:

```
template <typename InputIterator1, typename InputIterator2>
inline bool equal(InputIterator1 start1,
                  InputIterator1 end1,
                  InputIterator2 start2)
{
    while(start1 != end1)
    {
        if(*start1 != *start2) return false;
        ++start1;
        ++start2;
    }
    return true;
}
```

The algorithm walks over the range defined by `[start1, end1)` and `[start1, start1 + (end1 - start1))` and returns whether the elements in the range are equal. Notice that the algorithm is templated over two different types of input iterators. Why is this?

3. One hassle with the `equal` algorithm is that it assumes that the input ranges are the same size. But what if we have two iterator ranges containing unknown numbers of elements that we'd like to compare? In that case, we'd have to manually compute the size of the ranges first, then call `equal` if they agreed. For random-access iterators this can be implemented quickly, but for simple input iterators can be inefficient. Modify the implementation of `equal` given above to create an algorithm called `ranges_are_equal` that accepts four inputs (two pairs of iterators) and returns true if the ranges are the same size and contain the same elements. As an incentive, if you've written `ranges_are_equal` correctly, you can check whether two files have the same contents in three lines of code:

```
ifstream input1("file1.txt");
ifstream input2("file2.txt");
return ranges_are_equal(istreambuf_iterator<char>(input1),
                        istreambuf_iterator<char>(),
                        istreambuf_iterator<char>(input2),
                        istreambuf_iterator<char>())
```

`istreambuf_iterator` is an iterator adapter that, like `istream_iterator`, iterates over a stream reading values. Unlike `istream_iterator`, however, `istreambuf_iterator` always reads raw values from the stream with `get` instead of using stream extraction, so it doesn't skip whitespace. The iterator is specialized over `char` since we're reading from an `ifstream`, which encodes characters as `chars`.

4. In the chapter on the preprocessor, we defined a `MAX` macro as follows:

```
#define MAX(a, b) ((a) < (b) ? (b) : (a))
```

Compare this macro to the `Max` template function we wrote in this chapter and explain why the template function is safer. You might want to review the chapter on the preprocessor before answering this question.

5. One of the motivations behind requiring C++ programmers to put the `typename` keyword before accessing a type nested inside a templated type was to prevent ambiguity in templates. For example, consider the following code, which assumes the existence of a template type called `MyTemplate`:

```
template <typename T> void MyFunction(const T& elem)
{
    int itr;
    while(true)
    {
        typename MyTemplate<T>::Inner * itr;
    }
}
```

Consider the line `typename MyTemplate<T>::Inner * itr`. Since we have the `typename` keyword here, the compiler understands that we're declaring a pointer of type `MyTemplate<T>::Inner*` called `itr`. If we don't put the `typename` keyword in here, C++ will think that `MyTemplate<T>::Inner` is a class constant rather than a type. What will the statement `MyTemplate<T>::Inner * itr` mean in this case? Does the difference between these two cases help explain why the `typename` keyword is necessary?

6. Using the code for `GetInteger` we wrote in the streams chapter as a base, write a template function `GetValue` which safely reads a value of any type from the console. The signature for this function should be

```
template <typename ValueType> ValueType GetValue(string type);
```

Here, the `type` argument is a string containing a human-readable description of the type that is displayed in any error messages that arise from invalid input. For example, to read in an `int`, you would call `GetValue<int>("an integer");` ♦

7. Modify the code for `GetValue` to implement the `lexical_cast` function. If an error occurs because the string is not formatted properly, you can assume the existence of an `Error` function in the style of CS106B/X. Later, when we cover exceptions, we'll see a better way to report an error.
8. Write a template function `Clamp` that accepts three values – a value by reference to “clamp,” a lower bound, and an upper bound – then updates the reference parameter so that it is between the lower and upper bounds. For example, if `x` has value 42, calling `Clamp(x, 100, 200)` would set `x` to 100 while calling `Clamp(x, 0, 137)` would leave `x` unchanged. What is the implicit interface on the type of the reference parameter?

Chapter 15: `const`

At its core, C++ is a language based on modifying program state. `ints` get incremented in `for` loops; `vectors` have innumerable calls to `clear`, `resize`, and `push_back`; and console I/O overwrites variables with values read directly from the user.

When designing or maintaining a large software system, it is important to track exactly where side effects can occur. A member function named `getWidth`, for example, probably should not change the width of the receiver object since class clients would not expect the function to do so. Similarly, when passing a variable by reference to a function, it is critical that you are able to check whether the function will destructively modify the variable or whether it plans on leaving it untouched.

To track and monitor side effects, C++ uses the `const` keyword. You have already seen `const` in the context of global constants, but the `const` keyword has many other uses. This chapter introduces the mechanics of `const` (for example, where `const` can be used and what it means in these contexts) and how to properly use it in C++ code.

A Word on Error Messages

The error messages you'll get if you accidentally break `constness` can be intimidating and confusing. Commonly you'll get unusual errors about conversions losing qualifiers or lvalues specifying `const` objects. These are the telltale signs that you've accidentally tried to modify a `const` variable. Also, the errors for `const` can be completely incomprehensible when working with template classes and the STL. As with all STL errors, understanding these error messages will only come with practice.

`const` Variables

So far, you've only seen `const` in the context of global constants. For example, given the following global declaration:

```
const int MyConstant = 137;
```

Whenever you refer to the value `MyConstant` in code, the compiler knows that you're talking about the value 137. If later in your program you were to write `MyConstant = 42`, you'd get a compile-time error since you would be modifying a `const` variable.

However, `const` is not limited to global constants. You can also declare local variables `const` to indicate that their values should never change. Consider the following code snippet:

```
int length = myVector.size();
for(int i = 0; i < length; ++i)
    /* ...Do something that doesn't modify the length of the vector... */
```

Here, we compute `length` only once since we know at compile-time that our operation isn't going to modify the contents of `myVector`. Since we don't have the overhead of a call to `myVector.size()` every iteration, on some compilers this loop can be about ten percent faster than if we had simply written the conventional `for` loop.

Because the length of the `vector` is a constant, we know at compile-time that the variable `length` should never change. We can therefore mark it `const` to have the compiler enforce that it *must* never change, as shown here:

```

const int length = myVector.size();
for(int i = 0; i < length; ++i)
    /* ...Do something that doesn't modify the length of the vector... */

```

This code works identically to the original, except that we've explicitly announced that we will not change `length`. If we accidentally try to modify `length`, we'll get a compile-time error directing us to the offending line, rather than a runtime bug. In essence, the compiler will help us debug our code by converting a potential runtime bug into an easily fixable compile-time error. For example, suppose we want to perform some special handling if the vector is exactly ten elements long. If we write `if(length == 10)` instead of `if(length == 10)`, if `length` isn't declared `const`, our code will always execute exactly ten times and we'll get strange behavior where an `if` statement executes more frequently than expected. Since assignments-in-`if` statements are perfectly legal C++ code, the C++ compiler won't warn us of this fact and we will have to debug the program by hand. However, if we mark `length` `const`, the compiler will generate an error when compiling the statement `if(length == 10)` because it contains an assignment to a `const` variable. No longer will we have to track down errant runtime behavior – for once it appears that the C++ compiler is helping us write good code!

const and Pointers

The `const` keyword is useful but has its share of quirks. Perhaps the most persistent source of confusion when working with `const` arises in the context of pointers. For example, suppose that you want to declare a C string as a global constant. Since to declare a global C++ string constant you use the syntax

```
const string kGlobalCppString = "This is a string!";
```

You might assume that to make a global C string constant, the syntax would be:

```
const char* kGlobalString = "This is a string!";
```

This syntax is partially correct. If you were ever to write `kGlobalString[0] = 'X'`, rather than getting segmentation faults at runtime (see the C strings chapter for more info), you'd get a compiler error that would direct you to the line where you tried to modify the global constant. But unfortunately this variable declaration isn't quite right. Suppose, for example, that you write the following code:

```
kGlobalString = "Reassigned!";
```

Here, you're reassigning `kGlobalString` to point to the string literal “Reassigned!” Note that you aren't modifying the contents of the character sequence `kGlobalString` points to – instead you're changing *what character sequence* `kGlobalString` points to. In other words, you're modifying the *pointer*, not the *pointee*, so the above line will compile correctly and other code that references `kGlobalString` will suddenly begin using the string “Reassigned!” instead of “This is a string!” as you would hope.

When working with `const`, C++ distinguishes between two similar-sounding entities: a *pointer-to-const* and a *const pointer*. A *pointer-to-const* is a pointer like `kGlobalString` that points to data that cannot be modified. While you're free to reassign pointers-to-`const`, you cannot change the value of the elements they point to. To declare a *pointer-to-const*, use the syntax **const Type* myPointer**, with the `const` on the left of the star. Alternatively, you can declare pointers-to-`const` by writing **Type const* myPointer**.

A *const pointer*, on the other hand, is a pointer that cannot be assigned to point to a different value. Thus with a `const` pointer, you can modify the *pointee* but not the *pointer*. To declare a *const* pointer, you use the syntax **Type* const myConstPointer**, with the `const` on the right side of the star. Here, `myConstPointer` can't be reassigned, but you are free to modify the value it points to.

Note that the syntax for a pointer-to-const is `const Type * ptr` while the syntax for a const pointer is `Type * const ptr`. The only difference is where the `const` is in relation to the star. One trick for remembering which is which is to read the variable declaration from right-to-left. For example, reading `const Type * ptr` backwards says that “`ptr` is a pointer to a `Type` that's `const`,” while `Type * const ptr` read backwards is “`ptr` is a `const` pointer to a `Type`.”

Returning to the C string example, to make `kGlobalString` behave as a true C string constant, we'd need to make the pointer both a `const` pointer and a pointer-to-`const`. This is totally legal in C++, and the result is a `const` pointer-to-`const`. The syntax looks like this:

```
const char * const kGlobalString = "This is a string!";
```

Note that there are *two* `const`s here – one before the star and one after it. Here, the first `const` indicates that you are declaring a pointer-to-`const`, while the second means that the pointer itself is `const`. Using the trick of reading the declaration backwards, here we have “`kGlobalString` is a `const` pointer to a `char` that's `const`.” This is the correct way to make the C string completely `const`, although it is admittedly a bit clunky.

The following table summarizes what types of pointers you can create with `const`:

Declaration Syntax	Name	Can reassign?	Can modify pointee?
<code>const Type* myPtr</code>	Pointer-to- <code>const</code>	Yes	No
<code>Type const* myPtr</code>	Pointer-to- <code>const</code>	Yes	No
<code>Type *const myPtr</code>	<code>const</code> pointer	No	Yes
<code>const Type* const myPtr</code>	<code>const</code> pointer-to- <code>const</code>	No	No
<code>Type const* const myPtr</code>	<code>const</code> pointer-to- <code>const</code>	No	No

A subtle point with pointers-to-`const` is that it is legal to point to non-`const` data with a pointer-to-`const`. For example, the following code is perfectly legal:

```
int myInt;
const int* myPtr = &myInt;
```

Here, we are free to modify the `myInt` variable as we see fit, but we promise not to change it indirectly through `myPtr`. By restricting what we are capable of doing to `myInt` through `myPtr`, we reduce the potential for errors.

const Objects

So far, all of the `const` cases we've dealt with have concerned primitive types. What happens when we mix `const` with objects?

Let us first consider a `const` string, a C++ string whose contents cannot be modified. We can declare a `const` string as we would any other `const` variable. For example:

```
const string myString = "This is a constant string!";
```

Note that, like all `const` variables, we are still allowed to assign the `string` an initial value.

Because the `string` is `const`, we're not allowed to modify its contents, but we can still perform some basic operations on it. For example, here's some code that prints out the contents of a `const` string:

```
const string myString = "This is a constant string!";
for(int i = 0; i < myString.length(); ++i)
    cout << myString[i] << endl;
```

To us humans, the above code seems completely fine and indeed it is legal C++ code. But how does the compiler know that the `length` function doesn't modify the contents of the `string`? This question generalizes to a larger question: given an arbitrary class, how can the compiler tell which member functions might modify the class and which ones don't? To answer this question, let's look at the prototype for the `string` member function `length`:

```
size_type length() const;
```

Note that there is a `const` after the member function declaration. This is another use of the `const` keyword that indicates that the member function does not modify any of the class's instance variables. That is, when calling a `const` member function, you're guaranteed that the object's contents will not change.*

When working with `const` objects, you are only allowed to call member functions on that object that have been explicitly marked `const`. That is, even if you have a function that doesn't modify the object, unless you tell the compiler that the member function is `const`, the compiler will treat it as a non-`const` function.

For example, let's consider a `Point` class that simply stores a point in two-dimensional space:

```
class Point
{
public:
    double getX();
    double getY();

    void setX(double newX);
    void setY(double newY);
private:
    double x, y;
};
```

Consider the following implementation of `getX`:

```
double Point::getX()
{
    return x;
}
```

Since this function doesn't modify the `Point` object in any way, we should change the prototype in the class definition to read

```
double getX() const
```

so we can call this function on `const Point` objects. Similarly, we need to add a `const` marker to the function definition, as shown here:

```
double Point::getX() const
{
    return x;
}
```

* In some cases it is possible for a `const` member function to modify its receiving object, as you'll see later in this chapter. However, it's perfectly reasonable to view `const` member functions this way.

Forgetting to add this `const` can be a source of much frustration because the C++ treats `getX()` and `getX() const` as two different functions.

In a `const` member function, all the class's instance variables are treated as `const`. You can read their values, but must not modify them. Additionally, inside a `const` member function, you cannot call other non-`const` member functions, since they might modify the receiver. Beyond these restrictions, `const` member functions can do anything that regular member functions can do. Consider, for example, the following implementation of a `distanceToOrigin` function for the `Point` class:

```
void Point::distanceToOrigin() const
{
    double dx = getX();    // Legal!   getX is const.
    double dy = y;         // Legal!   Reading an instance variable.
    dx *= dx;              // Legal!   We're modifying dx, which isn't an
                            //         instance variable.
    dy *= dy;              // Legal!
    return sqrt(dx + dy);  // Legal!   sqrt is a free function that can't
                            //         modify the current object.
}
```

const References

Throughout this course we've used pass-by-reference to avoid copying objects between function calls. However, pass-by-reference can lead to some ambiguity. For example, suppose you see the following function prototype:

```
void DoSomething(vector<int>& vec);
```

You know that this function accepts a `vector<int>` by reference, but it's not clear why. Does `DoSomething` modify the contents of the `vector<int>`, or is it just accepting by reference to avoid making a deep copy of the `vector`?

To remove this ambiguity, we can use `const` references. A `const` reference is like a normal reference except that the original object is treated as though it were `const`. For example, consider this modified function prototype:

```
void DoSomething(const vector<int> &vec);
```

Because the parameter is a `const` reference, the `DoSomething` function cannot modify the `vector`.

You are allowed to pass both `const` and non-`const` variables to functions accepting `const` references. Whether or not the original variable is `const`, inside the function call it is treated as though it were. Thus it's legal (and encouraged) to write code like this:

```
/* Since we're not changing vec, we marked it const in this function. */
void PrintVector(const vector<int>& vec)
{
    copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main()
{
    vector<int> myVector(NUM_INTS);
    PrintVector(myVector); // Becomes const once inside PrintVector
    myVector.push_back(137); // Legal here, myVector isn't const.
}
```

While it's legal to pass non-`const` objects to functions accepting `const` references, you cannot pass `const` objects into functions accepting non-`const` references. You can think of `const` as a universal acceptor and of non-`const` as the universal donor – you can convert both `const` and non-`const` data to `const` data, but you can't convert `const` data to non-`const` data.

`const` references differ from regular references in one additional aspect. Consider the following code:

```
void DoSomething(int& x);

int myInt;
double myDouble;
DoSomething(myInt);
DoSomething(137);      // Error!
DoSomething(myDouble); // Error!
DoSomething(2.71828);  // Error!
```

The last three calls to `DoSomething` are illegal because the function accepts its parameter by reference. In the first case, `DoSomething` cannot accept 137 as a parameter because 137 is not an lvalue; that is, it is illegal to have 137 on the left-hand side of an assignment. In the second, it is illegal to pass `myDouble` to `DoSomething` because `DoSomething` accepts an `int&` and while doubles can be converted to ints, `double&`s cannot be converted to `int&`s. The last call to `DoSomething` is illegal because 2.71828 is neither an integer nor an lvalue.

However, if we change the prototype of `DoSomething` to accept its parameter by `const` reference, all three of these calls are legal. Why the difference? The reason is that a value referenced by a `const` reference cannot be assigned a value through that reference. For example, if we prototype `DoSomething` to accept `x` by reference-to-`const`, it is illegal to write `x = 42` in the body of the function. Because the value is not assignable, it is safe to bind the reference to values that might not be lvalues or even integers, since C++ can initialize the reference to a temporary value.

This behavior of `const` references is not restricted to ints and in fact any function that accepts a parameter by reference-to-`const` can accept constants or values implicitly convertible to the specified type. Since functions accepting parameters by reference-to-`const` do not make deep copies of their arguments, you can think of reference-to-`const` as a smarter version of pass-by-value. We will address this more next chapter.

`const_iterator`

Suppose you have a function that accepts a `vector<string>` by reference-to-`const` and you'd like to print out its contents. You might want to write code that looks like this:

```
void PrintVector(const vector<string> &myVector)
{
    for(vector<string>::iterator itr = myVector.begin(); // ERROR!
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

Unfortunately, this code will generate a compile-time error. The problem is in the first part of the for loop where we declare an object of type `vector<string>::iterator`. Because the `vector` is `const`, somehow the compiler has to know that the iterator you're getting to the `vector` can't modify the `vector`'s contents. Otherwise, we might be able to do something like this:

```
vector<string>::iterator itr = myVector.begin(); // Assume myVector is const
*itr = 42; // Just modified a const object!
```


Your initial thought might be to declare the iterator `const` to indicate that you won't modify what the iterator's pointing to. This won't work either, though, since a `const` iterator is like a `const` pointer – while you can't modify which element the iterator is iterating over, you can change the value of the element referenced by the iterator.

To fix this problem, all STL containers define a type called a `const_iterator` that is capable of reading the values of a container but not modifying them. Thus the proper version of the above code is:

```
void PrintVector(const vector<string> &mySet)
{
    for(vector<string>::const_iterator itr = myVector.begin(); // Correct!
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

To maintain constness, you cannot use `const_iterators` in functions like `insert` or `erase` that modify containers. You can, however, define iterator ranges using `const_iterators` for algorithms like `binary_search` that don't modify the ranges they apply to.

One interesting point about the difference between the `iterator` and `const_iterator` is that all STL containers define two different `begin` and `end` functions – non-`const` versions that return `iterators` and `const` versions that return `const_iterators`. When two member functions have the same name but differ in their constness, C++ will call the version of the function that has the same constness as the receiver object. That is, a non-`const` vector will always call the non-`const` version of `begin`, while a `const` vector will always call the `const` version of `begin`. This is sometimes known as “const overloading.”

Limitations of `const`

Although `const` is a useful programming construct, it is imperfect. One common problem arises when using pointers in `const` member functions. Suppose you have the following class that encapsulates a C string:

```
class CString
{
public:
    /* ... other members ...*/

    void constFunction() const;
private:
    char* theString;
};
```

Consider the following legal implementation of `constFunction`:

```
void CString::constFunction() const
{
    strcpy(theString, "Oh no!");
}
```

Unfortunately, while this code modifies the value of the object pointed to by `theString`, it is totally legal since we didn't modify the value of `theString` – instead, we changed the value of the elements it pointed at. In effect, because the member function is declared `const`, `theString` acts as a `const` pointer instead of a pointer-to-`const`.

This raises the issue of the distinction between “bitwise constness” versus “semantic constness.” *Bitwise constness*, which is the type enforced by C++, means that `const` classes are prohibited from making any bit-

wise changes to themselves. In the above example, since the value of `theString` didn't change (because we didn't reassign it), C++ considers it `const`-correct. However, from the viewpoint of *semantic constness*, `const` classes should be prohibited from modifying anything that would make the object appear somehow different. With regards to the above scenario with `theString`, the class isn't semantically `const` because the object, while `const`, was able to modify its data.

When working with `const` it's important to remember that while C++ will enforce bitwise `constness`, you must take care to ensure that your program is semantically `const`. From your perspective as a programmer, if you call a function that's marked `const`, you would expect that it cannot modify whatever class it was working on. If the function isn't semantically `const`, however, you'll run into problems where code that shouldn't be modifying an object somehow leaves the object in a different state.

To demonstrate the difference between bitwise and semantically `const` code, let's consider another member function of the `CString` class that simply returns the internally stored string:

```
char* CString::getString() const
{
    return theString;
}
```

Initially, this code looks correct. Since returning the string doesn't modify the string's contents, we've marked the function `const`, and, indeed, the function is bitwise `const`. However, our code has a major flaw. Consider the following code:

```
const CString myStr = "This is a C string!";
strcpy(myStr.getString(), "Oh no!");
```

Here, we're using the pointer we obtained from `getString` as a parameter to `strcpy`. After the `strcpy` completes, `myStr`'s internal string will contain "Oh no!" instead of "This is a C string!" We've modified a `const` object using only `const` member functions, something that defeats the purpose of `const`.

Somehow we need to change the code to prevent this from happening. The problem is that the pointer returned by `getString` is not itself `const`, so it's legal to use it with functions like `strcpy`. To resolve this problem, we can simply change the return value of `getString` from `char *` to `const char *`. This approach solves the problem because it's now illegal to modify the string pointed at by the return value. In general, when returning pointers or references to an object's internal data, you should make sure to mark them `const` when appropriate.

The above example illustrates that making semantically-`const` code can be difficult. However, the benefits of semantically-`const` code are noticeable – your code will be more readable and less error-prone.

mutable

Because C++ enforces bitwise `constness` rather than semantic `constness`, you might find yourself in a situation where a member function changes an object's bitwise representation without modifying its semantic value. At first this might seem unusual – how could we possibly leave the object in the same logical state if we change its binary representation? – but such situations can arise in practice. For example, suppose that we want to write a class that represents a grocery list. The class definition is provided here:

```

class GroceryList
{
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const;
private:
    vector<string> data;
};

```

The `GroceryList` constructor takes in a filename representing a grocery list (with one element per line), then allows us to look up items in the list using the member function `getItemAt`. Initially, we might want to implement this class as follows:

```

GroceryList::GroceryList(const string& filename)
{
    /* Read in the entire contents of the file and store in the vector. */
    ifstream input(filename.c_str());
    data.insert(data.begin(), istream_iterator<string>(input),
                istream_iterator<string>()); // See note*
}

/* Returns the element at the position specified by index. */
string GroceryList::getItemAt(int index) const
{
    return data[index];
}

```

Here, the `GroceryList` constructor takes in the name of a file and reads the contents of that file into a `vector<string>` called `data`. The `getItemAt` member function then accepts an index and returns the corresponding element from the `vector`. While this implementation works correctly, in many cases it is needlessly inefficient. Consider the case where our grocery list is several million lines long (maybe if we're literally trying to find enough food to feed an army), but where we only need to look at the first few elements of the list. With the current implementation of `GroceryList`, the `GroceryList` constructor will read in the entire grocery list file, an operation which undoubtedly will take a long time to finish and dwarfs the small time necessary to retrieve the stored elements. How can we resolve this problem?

There are several strategies we could use to eliminate this inefficiency. Perhaps the easiest approach is to have the constructor open the file, and then to only read in data when it's explicitly requested in the `getItemAt` function. That way, we don't read any data unless it's absolutely necessary. Here is one possible implementation:

* If you're still a bit shaky on iterator adapters, the final two arguments to the `insert` function define a range spanning the entire source file `input`. See the chapter on STL iterators for more information.

```

class GroceryList
{
public:
    GroceryList(const string& filename);

    /* ... other member functions ... */

    string getItemAt(int index); // NOTE: not const
private:
    vector<string> data;
    ifstream sourceStream;
};

GroceryList::GroceryList(const string& filename)
{
    sourceStream.open(filename.c_str()); // Open the file.
}

string GroceryList::getItemAt(int index)
{
    /* Read in enough data to satisfy the request.  If we've already read it
     * in, this loop will not execute and we won't read any data.
     */
    while(index < data.length())
    {
        string line;
        getline(sourceStream, line);
        /* ... some sort of error-checking ... */
        data.push_back(line);
    }
    return data[index];
}

```

Unlike our previous implementation, the new `GroceryList` constructor opens the file without reading any data. The new `getItemAt` function is slightly more complicated. Because we no longer read all the data in the constructor, when asked for an element, one of two cases will be true. First, we might have already read in the data for that line, in which case we simply hand back the value stored in the `data` object. Second, we may need to read more data from the file. In this case, we loop reading data until there are enough elements in the `data` vector to satisfy the request, then return the appropriate string.

Although this new implementation is more efficient,* the `getItemAt` function can no longer be marked `const` because it modifies both the `data` and `sourceStream` data members. If you'll notice, though, despite the fact that the `getItemAt` function is not bitwise `const`, it is semantically `const`. `GroceryList` is supposed to encapsulate an immutable grocery list, and by shifting the file reading from the constructor to `getItemAt` we have only changed the implementation, not the guarantee that `getItemAt` will not modify the list. For situations such as these, where a function is semantically `const` but not bitwise `const`, C++ provides the `mutable` keyword. `mutable` is an attribute that can be applied to data members that indicates that those data members can be modified inside member functions that are marked `const`. We can thus rewrite the `GroceryList` class definition to look like this:

* The general technique of deferring computations until they are absolutely required is called *lazy evaluation* and is an excellent way to improve program efficiency. Consider taking CS242 or CS258 if you're interested in learning more about lazy evaluation.

```

class GroceryList
{
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const; // Now marked const
private:
    /* These data members now mutable. */
    mutable vector<string> data;
    mutable ifstream sourceStream;
};

```

Because `data` and `sourceStream` are both mutable, the new implementation of `getItemAt` can now be marked `const`, as shown above.

`mutable` is a special-purpose keyword that should be used sparingly and with caution. Mutable data members are exempt from the type-checking rules normally applied to `const` and consequently are prone to the same errors as non-`const` variables. Also, once data members have been marked `mutable`, *any* member functions can modify them, so be sure to double-check your code for correctness. Most importantly, though, do not use `mutable` to silence compiler warnings and errors unless you're absolutely certain that it's the right thing to do. If you do, you run the risk of having functions marked `const` that are neither bitwise nor semantically `const`, entirely defeating the purpose of the `const` keyword.

const-Correctness

I still sometimes come across programmers who think `const` isn't worth the trouble. "Aw, `const` is a pain to write everywhere," I've heard some complain. "If I use it in one place, I have to use it all the time. And anyway, other people skip it, and their programs work fine. Some of the libraries that I use aren't `const`-correct either. Is `const` worth it?"

We could imagine a similar scene, this time at a rifle range: "Aw, this gun's safety is a pain to set all the time. And anyway, some other people don't use it either, and some of them haven't shot their own feet off..."

Safety-incorrect riflemen are not long for this world. Nor are `const`-incorrect programmers, carpenters who don't have time for hard-hats, and electricians who don't have time to identify the live wire. There is no excuse for ignoring the safety mechanisms provided with a product, and there is particularly no excuse for programmers too lazy to write `const`-correct code.

– Herb Sutter, author of *Exceptional C++* and all-around C++ guru. [Sut98]

Now that you're familiar with the mechanics of `const`, we'll explore how to use `const` correctly in real-world C++ code. In the remainder of this chapter, we will explore *const-correctness*, a system for using `const` to indicate the effects of your functions (or lack thereof). From this point forward, *all* of the code in this reader will be `const`-correct and you should make a serious effort to `const`-correct your own code.

What is `const`-correctness?

At a high-level, `const`-correct code is code that clearly indicates which variables and functions cannot modify program state.

More concretely, *const*-correctness requires that *const* be applied consistently and pervasively. In particular, *const*-correct code tends to use *const* as follows:

1. *Objects are never passed by value.* Any object that would be passed by value is instead passed by reference-to-*const* or pointer-to-*const*.
2. *Member functions which do not change state are marked *const*.* Similarly, a function that is not marked *const* should mutate state somehow.
3. *Variables which are set but never modified are marked *const*.* Again, a variable not marked *const* should have its value changed at some point.

Let us take some time to explore the ramifications of each of these items individually.

Objects are never passed by value

C++ has three parameter-passing mechanisms – pass-by-value, pass-by-reference, and pass-by-pointer. The first of these requires C++ to make a full copy of the parameter being passed in, while the latter two initialize the parameter by copying a pointer to the object instead of the full object.* When passing primitive types (*int*, *double*, *char**, etc.) as parameters to a function, the cost of a deep copy is usually negligible, but passing a heavy object like a *string*, *vector*, or *map* can at times be as expensive as the body of the function using the copy. Moreover, when passing objects by value to a function, those objects also need to be cleaned up by their destructors once that function returns. The cost of passing an object by value is thus at least the cost of a call to the class's copy constructor (discussed in a later chapter) and a call to the destructor, whereas passing that same object by reference or by pointer simply costs a single pointer copy.

To avoid incurring the overhead of a full object deep-copy, you should avoid passing objects by value into functions and should instead opt to pass either by reference or by pointer. To be *const*-correct, moreover, you should consider passing the object by reference-to-*const* or by pointer-to-*const* if you don't plan on mutating the object inside the function. In fact, you can treat pass-by-reference-to-*const* or pass-by-pointer-to-*const* as the smarter, faster way of passing an object by value. With both pass-by-value and pass-by-reference-to-*const*, the caller is guaranteed that the object will not change value inside the function call.

There is one difference between pass-by-reference-to-*const* and pass-by-value, though, and that's when using pass-by-value the function gets a fresh object that it is free to destructively modify. When using pass-by-reference-to-*const*, the function cannot mutate the parameter. At times this might be a bit vexing. For example, consider the *ConvertToLowerCase* function we wrote in the earlier chapter on STL algorithms:

```
string ConvertToLowerCase(string toConvert)
{
    transform(toConvert.begin(), toConvert.end(), toConvert.begin(), ::tolower);
    return toConvert;
}
```

Here, if we simply change the parameter from being passed-by-value to being passed-by-reference-to-*const*, the code won't compile because we modify the *toConvert* variable. In situations like these, it is sometimes preferable to use pass-by-value, but alternatively we can rewrite the function as follows:

```
string ConvertToLowerCase(const string& toConvert)
{
    string result = toConvert;
    transform(result.begin(), result.end(), result.begin(), ::tolower);
    return result;
}
```

* References are commonly implemented behind-the-scenes in a manner similar to pointers, so passing an object by reference is at least as efficient as passing an object by pointer.

Here, we simply create a new variable called `result`, initialize it to the parameter `toConvert`, then proceed as in the above function.

Member functions which do not change state are `const`

If you'll recall from the previous chapter's discussion of `const` member functions, when working with `const` instances of a class, C++ only allows you to invoke member functions which are explicitly marked `const`. No matter how innocuous a function is, if it isn't explicitly marked `const`, you cannot invoke it on a `const` instance of an object. This means that when designing classes, you should take great care to mark `const` every member function that does not change the state of the object. Is this a lot of work? Absolutely! Does it pay off? Of course!

As an extreme example of why you should always mark nonmutating member functions `const`, suppose you try to pass a CS106B/X `Vector` to a function by reference-to-`const`. Since the `Vector` is marked as `const`, you can only call `Vector` member functions that themselves are `const`. Unfortunately, *none* of the `Vector`'s member functions are `const`, so you can't call *any* member functions of a `const Vector`. A `const CS106B/X Vector` is effectively a digital brick. As fun as bricks are, from a functional standpoint they're pretty much useless, so do make sure to `constify` your member functions.

If you take care to `const` correct all member functions that don't modify state, then your code will have an additional, stronger property: member functions which are *not* marked `const` are almost guaranteed to make some sort of change to their internal state. From an interface perspective this is wonderful – if you want to call a particular function that isn't marked `const`, you can almost guarantee that it's going to make some form of destructive modification to the receiver object. Thus when you're getting accustomed to a new code base, you can quickly determine what operations on an object modify that object and which just return some sort of internal state.

Variables which are set but never changed are `const`

Variables vary. That's why they're called variables. Constants, on the other hand, do not. Semantically, there is a huge difference between the sorts of operations you can perform on constants and the operations you can perform on variables, and using one where you meant to use the other can cause all sorts of debugging headaches. Using `const`, we can make explicit the distinction between constant values and true variables, which can make debugging and code maintenance much easier. If a variable is `const`, you cannot inadvertently pass it by reference or by pointer to a function which subtly modifies it, nor can you accidentally overwrite it with `=` when you meant to check for equality with `==`. Many years after you've marked a variable `const`, programmers trying to decipher your code will let out a sigh of relief as they realize that they don't need to watch out for subtle operations which overwrite or change its value.

Without getting carried away, you should try to mark as many local variables `const` as possible. The additional compile-time safety checks and readability will more than compensate for the extra time you spent typing those extra five characters.

Example: CS106B/X Map

As an example of what `const`-correctness looks like in practice, we'll consider how to take a variant of the CS106B/X `Map` class and modify it so that it is `const`-correct. The initial interface looks like this:

```
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size();
    bool isEmpty();

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key);

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key);
    ValueType& operator[](string key);

    void clear();

    void mapAll(void (fn)(string key, ValueType val));

    template <typename ClientDataType>
    void mapAll(void (fn)(string key, ValueType val, ClientDataType& data),
               ClientDataType &data);

    Iterator iterator();

private:
    /* ... Implementation specific ... */
};
```

The `operator[]` function shown here is what's called an *overloaded operator* and is the function that lets us write code to the effect of `myMap["Key"] = value` and `value = myMap["Key"]`. We will cover overloaded operators in a later chapter, but for now you can think of it simply as a function that is called whenever the `Map` has the element-selection brackets applied to it.

The first set of changes we should make to the `Map` is to mark all of the public member functions which don't modify state `const`. This results in the following interface:


```

/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key) const;
    ValueType& operator[](string key);

    void clear();

    void mapAll(void (fn)(string key, ValueType val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(string key, ValueType val, ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};

```

The `size`, `isEmpty`, and `containsKey` functions are all `const` because they simply query object properties without changing the `Map`. `get` is also `const` since accessing a key/value pair in the `Map` does not actually modify the underlying state, but `operator[]` should definitely *not* be marked `const` because it may update the container if the specified key does not exist.

The trickier functions to `const`-correct are `mapAll` and `iterator`. Unlike the STL iterators, CS106B/X iterators are read-only and can't modify the underlying container. Handing back an iterator to the `Map` contents therefore cannot change the `Map`'s contents, so we have marked `iterator` `const`. In addition, since `mapAll` passes its arguments to the callback function by value, there is no way for the callback function to modify the underlying container. It should therefore be marked `const`.

Now that the interface has its member functions `const`-ified, we should make a second pass over the `Map` and replace all instances of pass-by-value with pass-by-reference-to-`const`. In general, objects should never be passed by value and should always be passed either by pointer or reference with the appropriate `const`ness. This eliminates unnecessary copying and can make programs perform asymptotically better. The resulting class looks like this:

```

/* Note: Still more changes to make. Do not use this code as a reference! */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist. operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                        ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};

```

The parameters to `put`, `remove`, `containsKey`, `get`, and `operator[]` have all been updated to use pass-by-reference-to-`const` instead of pass-by-value. The trickier functions to modify are the `mapAll` functions. These functions themselves accept function pointers which initially took their values by value. We have updated them appropriately so that the function pointers accept their arguments by reference-to-`const`, since we assume that the class client will also be `const`-correct. Note that we did *not* mark the `ClientDataType&` parameter to `mapAll` `const`, since the `Map` client may actually want to modify that parameter.

There is one last change to make, and it concerns the `get` function, which currently returns a copy of the value associated with a given key. At a high-level, there is nothing intuitively wrong with returning a copy of the stored value, but from an efficiency standpoint we may end up paying a steep runtime cost by returning the object by value. After all, this requires a full object deep copy, plus a call to the object's destructor once the returned object goes out of scope. Instead, we'll modify the interface such that this function returns the object by reference-to-`const`. This allows the `Map` client to look at the value and, if they choose, copy it, but prevents clients from intrusively modifying the `Map` internals through a `const` function. The final, correct interface for `Map` looks like this:

```

/* const-corrected version of the CS106B/X Map. */
template <typename ValueType> class Map
{
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    const ValueType& get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                        ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};

```

As an interesting intellectual exercise, compare this code to the original version of the `Map`. The interface declaration is considerably longer than before because of the additional `const`s, but ultimately is more pleasing. Someone unfamiliar with the interface can understand, for example, that the `Map`'s `Iterator` type cannot modify the underlying container (since otherwise the `iterator()` function wouldn't be `const`), and can also note that `mapAll` allows only a read-only map operation over the `Map`. This makes the code more self-documenting, a great boon to programmers responsible for maintaining this code base in the long run.

Why be `const`-correct?

As you can see from the example with the CS106B/X `Map`, making code `const`-correct can be tricky and time-consuming. Indeed, typing out all the requisite `const`s and `&s` can become tedious after a while. So why should you want to be `const`-correct in the first place?

There are multiple reasons why code is better off `const`-correct than non-`const`-correct. Here are a few:

- *Code correctness.* If nothing else, marking code `const` whenever possible reduces the possibility for lurking bugs in your code. Because the compiler can check which regions of the code are and are not mutable, your code is less likely to contain logic errors stemming either from a misuse of an interface or from a buggy implementation of a member function.

- *Code documentation.* `const`-correct code is self-documenting and clearly indicates to other programmers what it is and is not capable of doing. If you are presented an interface for an entirely foreign class, you may still be able to figure out which methods are safe to call with important data by noting which member functions are `const` or accept parameters by reference-to-`const`.
- *Library integration.* The C++ standard libraries and most third-party libraries are fully `const`-correct and expect that any classes or functions that interface with them to be `const`-correct as well. Writing code that is not `const`-correct can prevent you from fully harnessing the full power of some of these libraries.

Practice Problems

Here are some open-ended problems that you can use to play around with `const` and `const`-correctness. I strongly encourage you to try them out as a way of getting a feel for what `const` feels like in the real world.

1. The following line of code declares a member function inside a class:

```
const char * const MyFunction(const string& input) const;
```

Explain what each `const` in this statement means.

2. Write a class with a member function `isConst` that returns whether the receiver object is `const`. (*Hint: Write two different functions named `isConst`*)
3. The STL `map`'s bracket operator accepts a key and returns a reference to the value associated with that key. If the key is not found, the `map` will insert a new key/value pair so that the returned reference is valid. Is this function bitwise `const`? Semantically `const`?
4. When working with pointers to pointers, `const` can become considerably trickier to read. For example, a `const int * const * const` is a `const` pointer to a `const` pointer to a `const int`, so neither the pointer, its pointee, or its pointee's pointee can be changed. What is an `int * const *`? How about an `int ** const **`?
5. In C++, it is legal to convert a pointer of type `T *` to a pointer of type `const T *`, since the `const T *`'s restrictions on what can be done with the pointee are a subset of the behaviors allowed by a variable of type `T *`. It is not legal, however, to convert a pointer of type `T **` to a pointer of type `const T **`, because doing so would open a hole in the type system that would allow you to modify a variable of type `const T`. Show why this is the case. ♦

6. The CS106B/X `Vector` has the following interface:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size();
    bool isEmpty();

    ElemType getAt(int index);
    void setAt(int index, ElemType value);

    ElemType& operator[](int index);

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(ElemType elem));
    template <typename ClientDataType>
        void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
                    ClientDataType & data);

    Iterator iterator();
};
```

Modify this interface so that it is `const`-correct. (*Hint: You may need to `const`-overload some of these functions*) ♦

7. Modify the Snake simulation code from the earlier extended example so that it is `const`-correct.

Chapter 16: Extended Example: UnionFind

We've covered a good many language features in the past few chapters, and to see how all of them work together we'll implement a specialized ADT called a *union-find data structure* (sometimes also called a *disjoint-set data structure*). Union-find structures are critical for several important algorithms, most notably *Kruskal's algorithm* for computing minimal spanning trees. As we design a union-find structure, we'll see what thought processes underlie the design of a C++ class and how the language features we've recently explored make for an expressive and flexible class that is surprisingly simple to use.

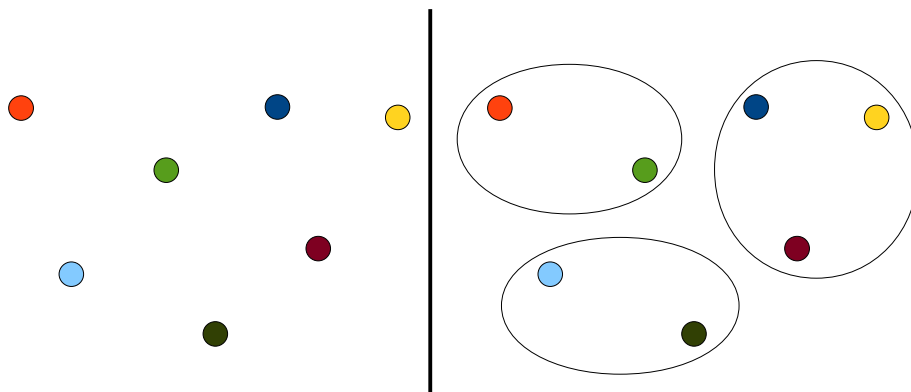
Much of the theoretical material described here about the union-find data structure is based on the discussion of disjoint-set forests in Chapter 21 of *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein. This is a most excellent algorithms text and if you're interested in expanding your computer science knowledge I highly recommend picking up a copy.

Tracking Disjoint Sets

Simply put, a set is an unordered collection of elements. For example, the set $\{1, 2, 3\}$ contains three elements and is equal to the set $\{2, 1, 3\}$ since ordering does not matter. However, it is not equal to the set $\{1, 2\}$ or the set $\{1, 2, 3, 4\}$ because the former does not contain the element three and the latter contains an extra element (4).

The *union* of two sets (denoted $A \cup B$) is the set formed by adding all of the elements of the first set to the second. For example, $\{1, 2, 3\} \cup \{3, 4, 5\}$ is $\{1, 2, 3, 4, 5\}$. Note that although 3 appears in both the first and second set, it only appears once in the union because sets do not allow for duplicate elements. Two sets are said to be *disjoint* if they contain no elements in common. For example, $\{1, 2, 3\}$ and $\{4, 5\}$ are disjoint, but $\{1, 2, 3\}$ and $\{3, 4, 5\}$ are not because both sets contain 3.

Suppose that we have some master set of elements S . Then a *partition* of S is a collection of sets such that all of the sets are disjoint and the union of the sets is the set S . In other words, every element of S belongs to one and exactly one set in the partition. As an example, given the set $\{1, 2, 3, 4, 5\}$, one partition is the sets $\{1, 2\}$, $\{3\}$, and $\{4, 5\}$ and another is $\{1, 2, 3, 4\}$, $\{5\}$. If we represent a set visually as a collection of points, then a partition is a way of grouping together those points such that no points are contained in two different groups. For example, here is a figure demonstrating a set S and a partition of S :

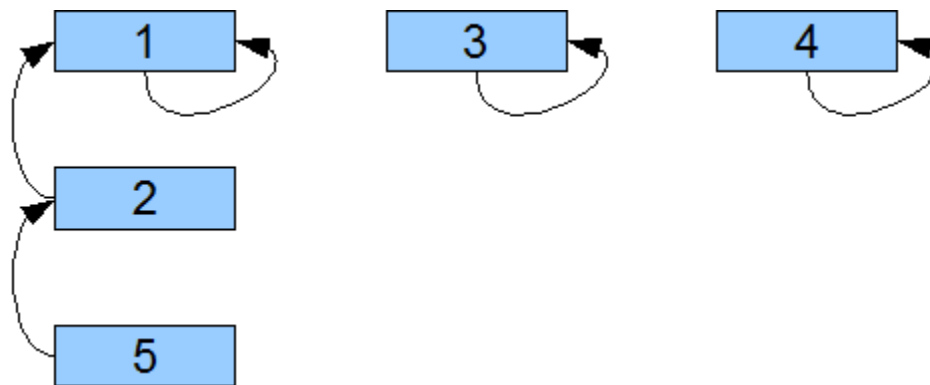


Given a set S that is partitioned into smaller sets $S_1, S_2, \dots, S_n$, we say that an element s_i is a *representative* of S_i if s_i is an element of S_i . For example, in the above example with the partition $\{1, 2\}, \{3\}, \{4, 5\}$ of the first five nonnegative integers, 1 is a representative of $\{1, 2\}$, 3 is a representative of $\{3\}$, and 5 is a representative of $\{4, 5\}$. Note that representatives are not unique – 2 is also a representative of $\{1, 2\}$ and 4 is a representative of $\{4, 5\}$.

If we have a partition of a set S , we can choose from every set in the partition a single, arbitrary element to form a *representative set* for S . Mathematically, this set can be useful if we have additional information about the underlying structure of the partition. From a computer science perspective, however, a representative set is useful if we want to test whether two elements are contained in the same partition. To check whether two elements are in the same partition, we first choose a representative set called the *designated representative set* and associate each set with its designated representative. To check whether a single set contains two elements, we can simply look up the designated representative of the set containing the first element and of the set containing the second element, then check whether the representatives are the same. If so, the two elements must belong to the same partition since each partition has a unique designated representative. If not, the elements must lie in different partitions.

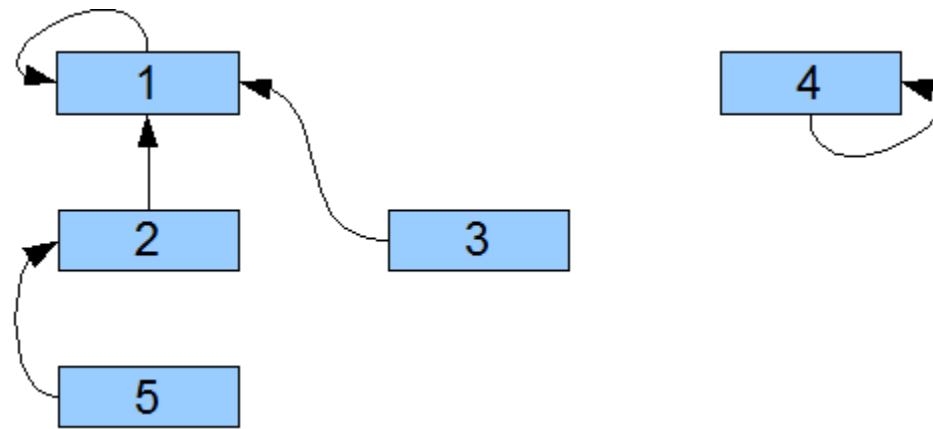
A *union-find data structure* is a data structure that manages a partition of a set S and a set of representatives for the sets in the partition. As its name implies, a union-find data structure supports two operations – *union*, which merges two sets, and *find*, which returns the representative of the set containing a given element. For example, suppose that $S = \{1, 2, 3, 4, 5\}$ as above. Initially, a union-find data structure stores the partition of S where every element is in its own set. In our case, this is the partition $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$ which has representatives 1, 2, 3, 4, 5. If we were to union together the sets containing 1 and 2, the new partition would be $\{1, 2\}, \{3\}, \{4\}, \{5\}$ and we would need to pick a new representative for $\{1, 2\}$; let's arbitrarily pick 1. Now our set of representatives is 1, 3, 4, 5. If at this point we wanted to take the union of the sets containing 2 and 5, the new partition would be $\{1, 2, 5\}, \{3\}, \{4\}$ and we would need to pick another representative for $\{1, 2, 5\}$; again we arbitrarily pick 1. This leaves us with a set of representatives 1, 3, 4.

There are many ways in which we can implement a union-find data structure, of which the fastest is called a *disjoint-set forest*. In a disjoint-set forest, every element in the master set S is represented as a node storing the value of the element and a link to the representative of its partition. For example, given the above partition and set of representatives (i.e. $\{1, 2, 5\}, \{3\}, \{4\}$ with representatives 1, 3, 4), one possibility for the disjoint-set forest looks like this:



Notice that the elements 3 and 4 point to themselves, since they are the representatives of their own equivalence class. Also, note that although 5 has 1 as the representative of its equivalence class, its link points to the element 2 which in turn points to 1. This is perfectly acceptable, and in fact the method we will use for computing the representative of a set is simply to follow the link pointers up to the first element that refers to itself.

Given this structure, it is simple to implement the union operation. Given two elements x and y whose sets we want to merge, we look up the representatives of their two sets, then arbitrarily change one of the two to refer to the other as a representative. For example, here is the effect of merging the sets $\{1, 2, 5\}$ and $\{3\}$:



Implementing the UnionFind Class

In the rest of this chapter, we will implement a class called `UnionFind` which encapsulates a disjoint-set forest. In summary, the `UnionFind` class should behave as follows:

1. When constructed, the `UnionFind` class stores a set partitioned such that every element is in its own partition.
2. `UnionFind` should allow us to query for the representative of the equivalence class of any element in the set. (*find*)
3. `UnionFind` should allow us to take the union of any two sets in the partition (*union*)

Notice that in this discussion of `UnionFind` we never explicitly mentioned what types of elements we plan on storing inside this `UnionFind` class. Our above discussion considered sets of integers, but we could have just as easily considered a set of strings or of doubles or `vector<set<int>>s`. Consequently, we will define `UnionFind` as a template class parameterized over the type of object we plan on storing. This leads to the following pseudo-definition of `UnionFind`:

```
template <typename ElemType> class UnionFind
{
    /* ... */
};
```

To implement the disjoint-set forest, we need to keep track of a collection of elements paired with a link to their parent. While there are several ways of implementing this, one simple solution is to store everything in a `map<ElemType, ElemType>` mapping every element to its parent. To save time, we'll use a `typedef` statement and define this type as `mapT`. This yields

```
template <typename ElemType> class UnionFind
{
    /* ... */
private:
    typedef map<ElemType, ElemType> mapT; // Shorthand for simplicity.
    mapT forest; // The actual disjoint-set forest.
};
```

As mentioned in the above description of `UnionFind`, the `UnionFind` constructor should accept as input a set of elements and initialize the partition such that each element is in its own set. But how should we accept the set of elements? As an STL `set`? A `vector`? A raw C++ array? Indeed, any of these choices seems reasonable. Rather than singling any of these forms out, instead we'll cater to the lowest common denominator by having the constructor accept as input a range of iterators delineating the input to use. Since the iterators can come from

any possible source, however, the constructor itself must be a template function parameterized over the type of the iterators. This makes `UnionFind` look as follows:

```
template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* ... */
private:
    typedef map<ElemType, ElemType> mapT; // Shorthand for simplicity.
    mapT forest; // The actual disjoint-set forest.
};
```

We can then implement the constructor as follows:

```
template <typename ElemType>
    template <typename IteratorType>
        UnionFind<ElemType>::UnionFind(IteratorType begin, IteratorType end)
{
    /* Iterate over the range, setting each element to be its representative. */
    for(; begin != end; ++begin)
        forest[*begin] = *begin;
}
```

Note that in this implementation we have two template headers – one indicating that the function is a member function of a class parameterized over `ElemType` and one indicating that the function itself is parameterized over `IteratorType`. The body of the function is not particularly tricky – we simply iterate over the range and set each element to refer to itself.

At this point we're ready to implement the `find` function, which simply walks up the links in the forest starting at the specified position until it reaches an element which has itself as a representative. Initially, we might want to consider prototyping `find` as follows:

```
ElemType find(const ElemType& toFind) const;
```

This is functionally correct, but it returns a deep copy of the stored object. If we are storing `ints` this isn't a problem, but if we are using a `UnionFind<string>` or a `UnionFind<vector<int>>` this operation could be costly. To circumvent this, we'll have `find` return a `const` reference to the element. This way, clients can still view the representative, but don't have to pay for a copy if they doesn't want to. This results in the following interface for `UnionFind`:

```

template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* Returns the representative of the equivalence class containing the
     * specified element.
     */
    const ElemType& find(const ElemType& toFind) const;

    /* ... */
private:
    typedef map<ElemType, ElemType> mapT; // Shorthand for simplicity.
    mapT forest; // The actual disjoint-set forest.
};

```

We can then implement `find` (recursively!) as follows:

```

template <typename ElemType>
const ElemType& UnionFind<ElemType>::find(const ElemType& toFind) const
{
    /* Get an iterator to the element/link pair, complaining if we can't. */
    mapT::const_iterator itr = forest.find(toFind);
    assert(itr != forest.end());

    /* If the iterator's second field is equal to toFind, the element is its
     * own representative and we can return it. Otherwise, recursively look
     * for the parent of this element.
     */
    return itr->second == toFind ? toFind : find(itr->second);
}

```

The final step in designing `UnionFind` is to implement the union operation. Unfortunately, `union` is a reserved keyword in C++,* so rather than naming this function `union` we will name it `link`. The implementation of `link` is surprisingly straightforward – we simply look up the representatives for the two sets to link, then arbitrarily set one to use the other as a representative. This results in the following interface and implementation:

* A union is similar to a `struct` or `class`, except that the data members all share a memory location. Consult a reference for more details.

```

template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* Returns the representative of the equivalence class containing the
     * specified element.
     */
    const ElemType& find(const ElemType& toFind) const;

    /* Unions the two sets containing the specified elements. */
    void link(const ElemType& one, const ElemType& two);
private:
    typedef map<ElemType, ElemType> mapT; // Shorthand for simplicity.
    mapT forest; // The actual disjoint-set forest.
};

template <typename ElemType>
void UnionFind<ElemType>::link(const ElemType& one, const ElemType& two)
{
    forest[find(one)] = forest[find(two)];
}

```

Implementing Path Compression

The implementation of a disjoint-set forest used in `UnionFind` performs all of the required operations of a union-find data structure, but can be made considerably more efficient via several small transformations.

One particular change that is easy to implement is *path compression*. Every time we look up the representative of a set, we may have to traverse a very long chain of nodes until we hit the representative. After we've computed the representative once, however, we can simply change the representative link so that it directly refers to the representative. This reduces the number of recursive calls necessary to look up the representative for a given class. Thus, after each call to `find`, we will update all of the link pointers for elements we accessed during the search.

An important point to note here is that we marked `find` `const` in the initial version of `UnionFind`, since looking up the representative for a set doesn't change the structure of the sets themselves. However, path compression potentially requires us to modify the `forest` in the `UnionFind` data structure on each call to `find`, meaning that `find` is no longer bitwise `const`. However, it *is* semantically `const`, so we will mark the `forest` field mutable so we can modify it in the `const` member function `find`.

The updated `UnionFind` interface and implementation of `find` are given below:

```

template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* Returns the representative of the equivalence class containing the
     * specified element.
     */
    const ElemType& find(const ElemType& toFind) const;

    /* Unions the two sets containing the specified elements. */
    void link(const ElemType& one, const ElemType& two);
private:
    typedef map<ElemType, ElemType> mapT; // Shorthand for simplicity.
    mutable mapT forest; // The actual disjoint-set forest.
};

template <typename ElemType>
const ElemType& UnionFind<ElemType>::find(const ElemType& val) const
{
    mapT::iterator itr = lookupTable.find(val);
    assert(itr != lookupTable.end());

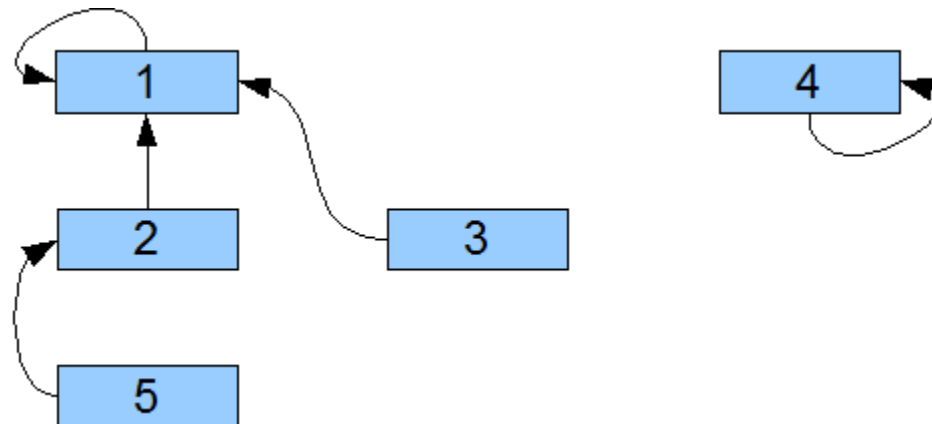
    /* If this element cycles to itself, return it. */
    if(itr->second == val) return val;

    /* Update the link to point directly to the representative, then
     * return it.
     */
    itr->second = find(itr->second);
    return itr->second;
}

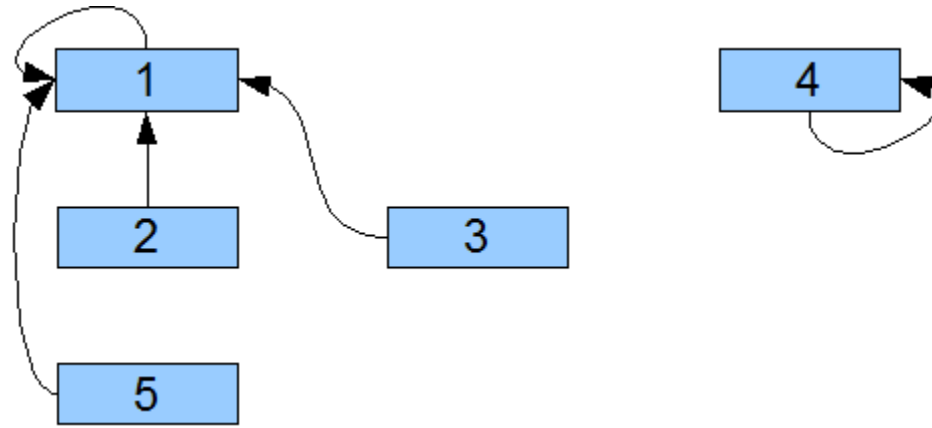
```

Implementing Union-by-Rank

Recall the diagram we had from earlier demonstrating one possible disjoint-set forest for the partition $\{1, 2, 3, 5\}, \{4\}$:



After applying path compression, we would end up with this disjoint-set forest:



Note how the element 5 now points directly to the element 1.

Suppose that at this point we want to merge the set $\{1, 2, 3, 5\}$ and $\{4\}$. We would do this by changing one of the representatives to point to the other representative. Now, we can do this either by having 1 refer to 4 as its parent, or by having 4 refer to 1 as its parent. Clearly it is better to have 4 point to 1, since 1 has multiple children that would need to have their parents recomputed on future calls to `find`. If, on the other hand, we change 4 to point to 1, we would only need to update a single link.

This observation motivates a final optimization we can implement on the `UnionFind` class call *union-by-rank*. The idea is that when trying to merge two representatives, we will try to make the element with fewer representatives point to the element with more representatives. One possible way to implement this is to tag each entry in the set with a *rank* representing the relative number of elements that treat that element as a representative. If we think of an element and all of the elements pointing to it as a tree, the rank is roughly the height of that tree before applying path compression. Our particular implementation of union-by-rank will use the following system:

1. Each node begins with rank 0, since no other nodes depend on it.
2. When merging two sets, if one representative has lower rank than the other, then that node is changed to point to the higher-rank node. This means that smaller trees (fewer dependencies) are always merged into larger trees (more dependencies). The rank of the high-rank tree doesn't change because its height has not increased.
3. When merging two sets, if both nodes have equal rank, one of the two nodes arbitrarily is chosen to point to the other node, and the other node has its rank increased by one. This means that equal-height trees are joined together to form a tree larger than both of the original trees.

We will not prove it here, but adding this change causes any m operations on the `UnionFind` data structure to complete in approximately $O(m \lg m)^*$ time.

To implement this set of changes, we simply need to change from using `ElemTypes` as the values in our map to using `pair<ElemType, int>`s as keys. The resulting code changes are shown here:

* Technically the asymptotic complexity is $O(m (\lg m) \alpha(m))$, where $\alpha(m)$ is an *extremely* slowly-growing function (for inputs smaller than the number of atoms in the known universe, it is less than five). If we were to switch from an STL-style tree map to a hash map as the main data structure backing the forest, this drops to $O(m\alpha(m))$.

```

template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* Returns the representative of the equivalence class containing the
     * specified element.
     */
    const ElemType& find(const ElemType& toFind) const;

    /* Unions the two sets containing the specified elements. */
    void link(const ElemType& one, const ElemType& two);
private:
    typedef map<ElemType, pair<ElemType, int> > mapT;
    mutable mapT forest; // The actual disjoint-set forest.
};

template <typename ElemType>
    template <typename IteratorType>
        UnionFind<ElemType>::UnionFind(IteratorType begin, IteratorType end)
    {
        /* Iterate over the range, setting each element equal to itself. */
        for(; begin != end; ++begin)
            forest[*begin] = make_pair(*begin, 0);
    }

template <typename ElemType>
const ElemType& UnionFind<ElemType>::find(const ElemType& val) const
{
    mapT::iterator itr = lookupTable.find(val);
    assert(itr != lookupTable.end());

    /* If this element cycles to itself, return it. */
    if(itr->second.first == val) return val;

    /* Update the link to point directly to the representative, then
     * return it.
     */
    itr->second.first = find(itr->second.first);
    return itr->second.first;
}

```

```

template <typename ElemType>
void UnionFind<ElemType>::link(const ElemType& one, const ElemType& two)
{
    /* Look up the representatives and their ranks. */
    pair<ElemType, int>& oneRep = lookupTable[find(one)];
    pair<ElemType, int>& twoRep = lookupTable[find(two)];

    /* If the two are equal, we're done. */
    if(oneRep.first == twoRep.first) return;

    /* Otherwise, union the two accordingly. */
    if(oneRep.second < twoRep.second)
        oneRep.first = twoRep.first;
    else if(oneRep.second > twoRep.second)
        twoRep.first = oneRep.first;
    /* On a tie, bump the rank. */
    else
    {
        oneRep.first = twoRep.first;
        ++twoRep.second;
    }
}

```

More to Explore

The Union-Find data structure we've introduced here can be used to implement all sorts of nifty algorithms that arise in nontrivial software systems. Consider looking into the following algorithms and seeing how you can use the UnionFind structure we've just created to implement them:

- **Kruskal's Algorithm:** A *graph* is a collection of *nodes* joined together by *arcs* that represents some sort of relationship. For example, a graph of the telephone system might have individual phones as nodes and telephone wires as arcs, while a graph of a social network might have people as nodes and arcs between individuals who know one another. *Kruskal's Minimum-Spanning Tree Algorithm* is a simple algorithm to find the minimum set of arcs such that every node is reachable from every other node. If we were to apply Kruskal's algorithm to the telephone system, for example, it would return the smallest number of wires we'd need to lay down to connect all of the world's telephones. Kruskal's algorithm hinges on the ability to track partitions of the set of all nodes, and is an excellent testbed for the Union-Find data structure. CS161 is also a great way to learn more about this and other graph algorithms.
- **Relational Unification:** One of the more powerful automated theorem proving techniques relies on the *unification algorithm*, which takes multiple predicates expressing true and false statements and combines them to try to derive a contradiction. The unification algorithm is sound and complete, meaning that given a set of input propositions that are consistent the algorithm can construct a proof of this consistency. Using the UnionFind data structure, it is relatively straightforward to perform the unification step, so if you're interested in automated theorem proving look into this algorithm (or CS157 for a more in-depth treatment).
- **Hindley-Milner Type Inference:** C++ is a statically-typed language, meaning that each variable is assigned a unique, immutable type at compile-time. However, you as a C++ programmer have to explicitly indicate the type of every variable and function, even if it's clear from context what types every variable must be. Other programming languages like ML and Haskell are also statically-typed, but use a *type inference* algorithm to automatically compute the types of each of the variables and expressions in the program. The particular algorithm used (the *Hindley-Milner Type Inference Algorithm*) is based on unification of type variables and is a great way to show off your UnionFind skills. Take CS242 if you're interested in learning more about type inference.

Complete UnionFind Listing

```

template <typename ElemType> class UnionFind
{
public:
    /* Accepts a range of elements and initializes the partition. */
    template <typename IteratorType>
        UnionFind(IteratorType begin, IteratorType end);

    /* Returns the representative of the equivalence class containing the
     * specified element.
     */
    const ElemType& find(const ElemType& toFind) const;

    /* Unions the two sets containing the specified elements. */
    void link(const ElemType& one, const ElemType& two);
private:
    typedef map<ElemType, pair<ElemType, int> > mapT;
    mutable mapT forest; // The actual disjoint-set forest.
};

template <typename ElemType>
    template <typename IteratorType>
        UnionFind<ElemType>::UnionFind(IteratorType begin, IteratorType end)
{
    /* Iterate over the range, setting each element equal to itself. */
    for(; begin != end; ++begin)
        forest[*begin] = make_pair(*begin, 0);
}

template <typename ElemType>
const ElemType& UnionFind<ElemType>::find(const ElemType& val) const
{
    mapT::iterator itr = lookupTable.find(val);
    assert(itr != lookupTable.end());

    /* If this element cycles to itself, return it. */
    if(itr->second.first == val) return val;

    /* Update the link to point directly to the representative, then
     * return it.
     */
    itr->second.first = find(itr->second.first);
    return itr->second.first;
}

```

```
template <typename ElemType>
void UnionFind<ElemType>::link(const ElemType& one, const ElemType& two)
{
    /* Look up the representatives and their ranks. */
    pair<ElemType, int>& oneRep = lookupTable[find(one)];
    pair<ElemType, int>& twoRep = lookupTable[find(two)];

    /* If the two are equal, we're done. */
    if(oneRep.first == twoRep.first) return;

    /* Otherwise, union the two accordingly. */
    if(oneRep.second < twoRep.second)
        oneRep.first = twoRep.first;
    else if(oneRep.second > twoRep.second)
        twoRep.first = oneRep.first;
    /* On a tie, bump the rank. */
    else
    {
        oneRep.first = twoRep.first;
        ++twoRep.second;
    }
}
```

Chapter 17: Member Initializer Lists

Normally, when you create a class, you'll initialize all of its instance variables inside the constructor. However, in some cases you'll need to initialize instance variables before the constructor begins running. Perhaps you'll have a `const` instance variable that you cannot assign a value, or maybe you have an object as an instance variable where you do not want to use the default constructor. For situations like these, C++ has a construct called the *member initializer list* that you can use to fine-tune the way your data members are set up. This chapter discusses initializer list syntax, situations where initializer lists are appropriate, and some of the subtleties of initializer lists.

How C++ Constructs Objects

To fully understand why initializer lists exist in the first place, you'll need to understand the way that C++ creates and initializes new objects.

Let's suppose you have the following class:

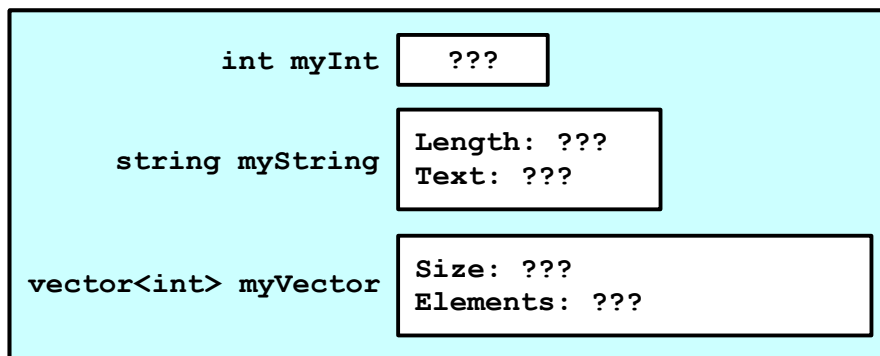
```
class SimpleClass
{
public:
    SimpleClass();
private:
    int myInt;
    string myString;
    vector<int> myVector;
};
```

Let's define the `SimpleClass` constructor as follows:

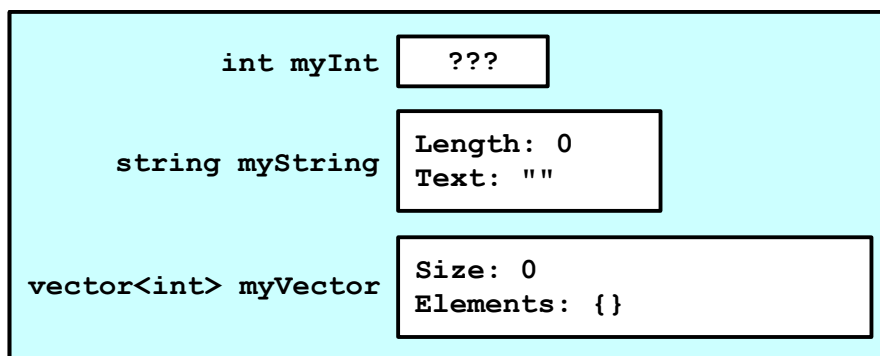
```
SimpleClass::SimpleClass()
{
    myInt = 5;
    myString = "C++!";
    myVector.resize(10);
}
```

What happens when you create a new instance of the class `MyClass`? It turns out that the simple line of code `MyClass mc` actually causes a cascade of events that goes on behind the scenes. Let's take a look at what happens, step-by-step.

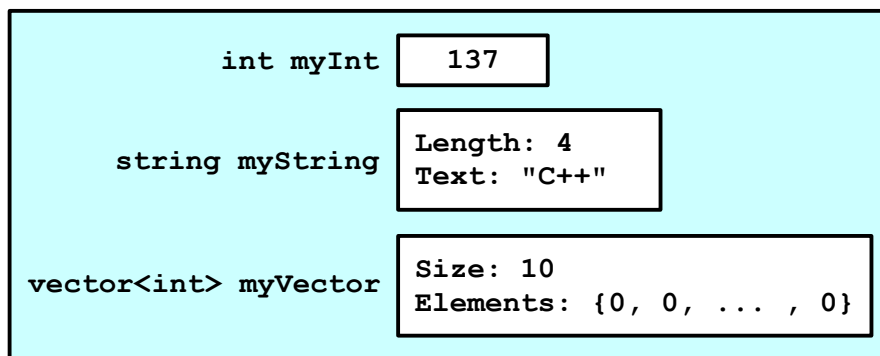
The first step in constructing a C++ object is simply to get enough space to hold all of the object's data members. The memory is not initialized to any particular value, so initially all of your object's data members hold garbage values. In memory, this looks something like this:



As you can see, none of the instance variables have been initialized, so they all contain junk. At this point, C++ calls the default constructor of each instance variable. For primitive types, this leaves the variables unchanged. After this step, our object looks something like this:



Finally, C++ will invoke the object's constructor so you can perform any additional initialization code. Using the constructor defined above, the final version of the new object will look like this:



At this point, our object is fully-constructed and ready to use.

However, there's one thing to consider here. Before we reached the `SimpleClass` constructor, C++ called the default constructor on both `myString` and `myVector`. `myString` was therefore initialized to the empty string, and `myVector` was constructed to hold no elements. However, in the `SimpleClass` constructor, we immediately assigned `myString` to hold "C++!" and resized `myVector` to hold ten elements. This means that we effectively initialized `myString` and `myVector` *twice* – once with their default constructors and once in the `SimpleClass` constructor.*

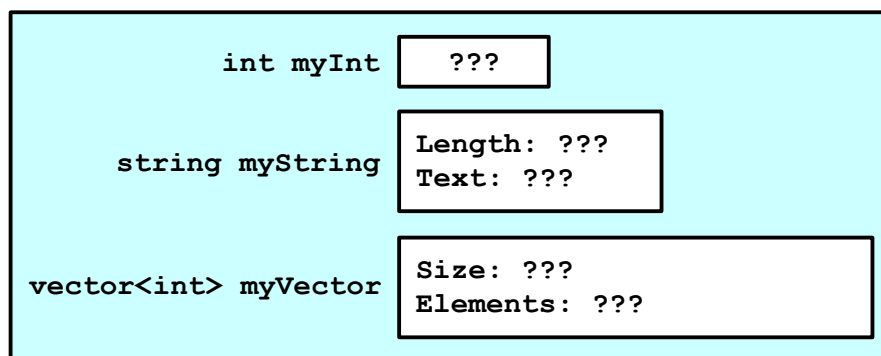
To improve efficiency and resolve certain other problems which we'll explore later, C++ has a feature called an *initializer list*. An initializer list is simply a series of values that C++ will use instead of the default values to initialize instance variables. For example, in the above example, you can use an initializer list to specify that the variable `myString` should be set to "C++!" before the constructor even begins running.

To use an initializer list, you add a colon after the constructor and then list which values to initialize which variables with. For example, here's a modified version of the `SimpleClass` constructor that initializes all the instance variables in an initializer list instead of in the constructor:

```
SimpleClass::SimpleClass() : myInt(5), myString("C++!"), myVector(10)
{
    // Note: Empty constructor
}
```

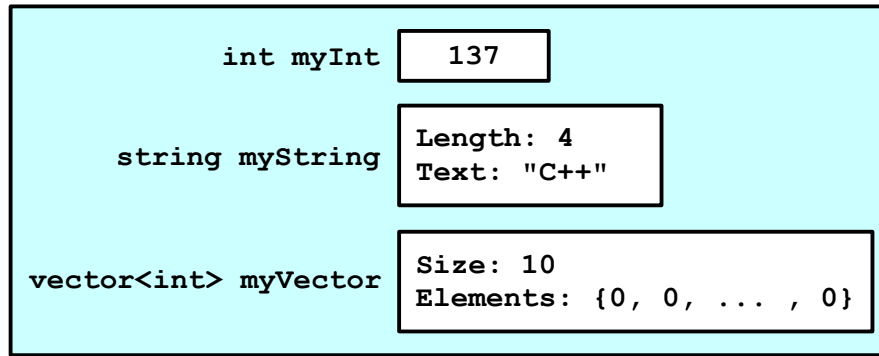
Here, we're telling C++ to initialize the variables `myInt` and `myString` to 5 and "C++!" respectively, before the class constructor is even called. Also, by writing `myVector(10)`, we're telling C++ to invoke the parametrized constructor of `myVector` passing in the value 10, which creates a `vector` with ten elements. This time, when we create a new object of type `myVector`, the creation steps will look like this:

First, as in the previous case, the object is allocated somewhere in memory and all variables have garbage values:



Next, C++ invokes all of the constructors for the object's data members using the values specified in the initializer list. The object now looks like this:

* Technically speaking, the objects are only initialized once, but the runtime efficiency is as though the objects were initialized multiple times. We'll talk about the differences between initialization and assignment in a later chapter.



Finally, C++ invokes the `MyClass` constructor, which does nothing. The final version of the class thus is identical to the above version.

As you can see, the values of the instance variables `myInt`, `myString`, and `myVector` are correctly set before the `SimpleClass` constructor is invoked. This is considerably more efficient than the previous version and will run much faster.

Note that while in this example we used initializer lists to initialize all of the object's instance variables, there is no requirement that you do so. However, in practice it's usually a good idea to set up all variables in an initializer list to make clear what values you want for each of your data members.

Parameters in Initializer Lists

In the above example, the initializer list we used specified constant values for each of the data members. However, it's both legal and useful to initialize data members with expressions instead of literal constants. For example, consider the following class, which encapsulates a C-style string:

```
class CString
{
public:
    CString(const char* input);
    ~CString();
    /* ... */
private:
    char* str;
};
```

Here, the constructor accepts a C-style string, then initializes the class to hold a copy of that string. Assuming that we have a `StringDuplicate` function like the one described in the chapter on C strings, we could write the constructor for `CString` as follows:

```
CString::CString(const char* input) : str(StringDuplicate(input))
{
    // Empty constructor
}
```

Notice that we were able to reference the constructor parameter `input` inside the initializer list. This allows us to use information specific to the current instance of the class to perform initialization and is ubiquitous in C++ code.

In some cases you may have a constructor that accepts a parameter with the same name as a class data member. For example, consider this the `RationalNumber` class, which encapsulates a rational number:

```
class RationalNumber
{
public:
    RationalNumber(int numerator = 0, int denominator = 1);
    /* ... */
private:
    int numerator, denominator;
};
```

The following is a perfectly legal constructor that initializes the data members to the values specified as parameters to the function:

```
RationalNumber::RationalNumber(int numerator, int denominator) :
    numerator(numerator), denominator(denominator)
{
    // Empty constructor
}
```

C++ is smart enough to realize that the syntax `numerator(numerator)` means to initialize the `numerator` data member to the value held by the `numerator` parameter, rather than causing a compile-time error or initializing the `numerator` data member to itself. Code of this form might indicate that you need to rename the parameters to the constructor, but is perfectly legal.

On an unrelated note, notice that in the `RationalNumber` class declaration we specified that the `numerator` and `denominator` parameters to `RationalNumber` were equal to zero and one, respectively. These are default arguments to the constructor and allow us to call the constructor with fewer than two parameters. If we don't specify the parameters, C++ will use these values instead. For example:

```
RationalNumber fiveHalves(5, 2);
RationalNumber three(3); // Calls constructor with arguments (3, 1)
RationalNumber zero; // Calls constructor with arguments (0, 1)
```

You can use default arguments in any function, provided that if a single parameter has a default argument every parameter after it also has a default. Thus the following code is illegal:

```
void DoSomething(int x = 5, int y); // Error - y needs a default
```

While the following is legal:

```
void DoSomething(int x, int y = 5); // Legal
```

When writing functions that take default arguments, you should *only* specify the default arguments in the function prototype, not the function definition. If you don't prototype the function, however, you should specify the defaults in the definition. C++ is very strict about this and even if you specify the same defaults in both the prototype and definition the compiler will complain.

When Initializer Lists are Mandatory

Initializer lists are useful from an efficiency standpoint. However, there are times where initializer lists are the only syntactically legal way to set up your instance variables.

Suppose we'd like to make an object called `Counter` that supports two functions, `increment` and `decrement`, that adjust an internal counter. However, we'd like to add the restriction that the `Counter` can't drop below 0 or exceed a user-defined limit. Thus we'll use a parametrized constructor that accepts an `int` representing the maximum value for the `Counter` and stores it as an instance variable. Since the value of the upper limit will never change, we'll mark it `const` so that we can't accidentally modify it in our code. The class definition for `Counter` thus looks something like this:

```
class Counter
{
public:
    Counter(int maxValue);
    void increment();
    void decrement();
    int getValue() const;
private:
    int value;
    const int maximum;
};
```

Then we'd *like* the constructor to look like this:

```
Counter::Counter(int maxValue)
{
    value = 0;
    maximum = maxValue; // ERROR!
}
```

Unfortunately, the above code isn't valid because in the second line we're assigning a value to a variable marked `const`. Even though we're in the constructor, we still cannot violate the sanctity of `constness`. To fix this, we'll initialize the value of `maximum` in the initializer list, so that `maximum` will be *initialized* to the value of `maxValue`, rather than *assigned* the value `maxValue`. This is a subtle distinction, so make sure to think about it before proceeding.

The correct version of the constructor is thus

```
Counter::Counter(int maxValue) : value(0), maximum(maxValue)
{
    // Empty constructor
}
```

Note that we initialized `maximum` based on the constructor parameter `maxValue`. Interestingly, if we had forgotten to initialize `maximum` in the initializer list, the compiler would have reported an error. In C++, it is *mandatory* to initialize all `const` primitive-type instance variables in an initializer list. Otherwise, you'd have constants whose values were total garbage.

Another case where initializer lists are mandatory arises when a class contains objects with no legal or meaningful default constructor. Suppose, for example, that you have an object that stores a CS106B/X `Set` of a custom type `customT` with comparison callback `MyCallback`. Since the `Set` requires you to specify the callback function in the constructor, and since you're always going to use `MyCallback` as that parameter, you might think that the syntax looks like this:


```
class SetWrapperClass
{
public:
    SetWrapperClass();
private:
    Set<customT> mySet(MyCallback); // ERROR!
};
```

Unfortunately, this isn't legal C++ syntax. However, you can fix this by rewriting the class as

```
class SetWrapperClass
{
public:
    SetWrapperClass();
private:
    Set<customT> mySet; // Note: no parameters specified
};
```

And then initializing `mySet` in the initializer list as

```
SetWrapperClass::SetWrapperClass() : mySet(MyCallback)
{
    // Yet another empty constructor!
}
```

Now, when the object is created, `mySet` will have `MyCallback` passed to its constructor and everything will work out correctly.

Multiple Constructors

If you write a class with multiple constructors (which, after we discuss of copy constructors, will be most of your classes), you'll need to make initializer lists for each of your constructors. That is, an initializer list for one constructor won't invoke if a different constructor is called.

Practice Problems

1. Explain each of the steps involved in object construction. Why do they occur in the order they do? Why are each of them necessary?
2. Based on your understanding of how classes are constructed, how do you think they are destructed? That is, when an object goes out of scope or is explicitly `deleted`, in what order do the following occur:
 - Memory for the object is reclaimed.
 - Data member destructors invoke.
 - Object destructor invokes.
3. Why must a function with a single parameter with default value must have default values specified for each parameter afterwards?
4. NASA is currently working on Project Constellation, which aims to resume the lunar landings and ultimately to land astronauts on Mars. The spacecraft under development consists of two parts – an orbital module called Orion and a landing vehicle called Altair. During a lunar mission, the Orion vehicle will orbit the Moon while the Altair vehicle descends to the surface. The Orion vehicle is designed such that it does not necessarily have to have an Altair landing module and consequently can be used for low Earth orbit missions in addition to lunar journeys. You have been hired to develop the systems software for the spacecraft. Because software correctness and safety are critically important, you want to design the system such that the compiler will alert you to as many potential software problems as possible.

Suppose that we have two classes, one called `OrionModule` and one called `AltairModule`. Since every Altair landing vehicle is associated with a single `OrionModule`, you want to define the `AltairModule` class such that it stores a pointer to its `OrionModule`. The `AltairModule` class should be allowed to modify the `OrionModule` it points to (since it needs to be able to dock/undock and possibly to borrow CPU power for critical landing maneuvers), but it should under no circumstance be allowed to change which `OrionModule` it's associated with. Here is a skeleton implementation of the `AltairModule` class:

```
class AltairModule
{
public:
    /* Constructor accepts an OrionModule representing the Orion spacecraft
     * this Altair is associated with, then sets up parentModule to point to
     * that OrionModule.
     */
    AltairModule(OrionModule* owner);

    /* ... */
private:
    OrionModule* parentModule;
};
```

Given the above description about what the `AltairModule` should be able to do with its owner `OrionModule`, appropriately insert `const` into the definition of the `parentModule` member variable. Then, implement the constructor `AltairModule` such that the `parentModule` variable is initialized to point to the `owner` parameter.

Chapter 18: static

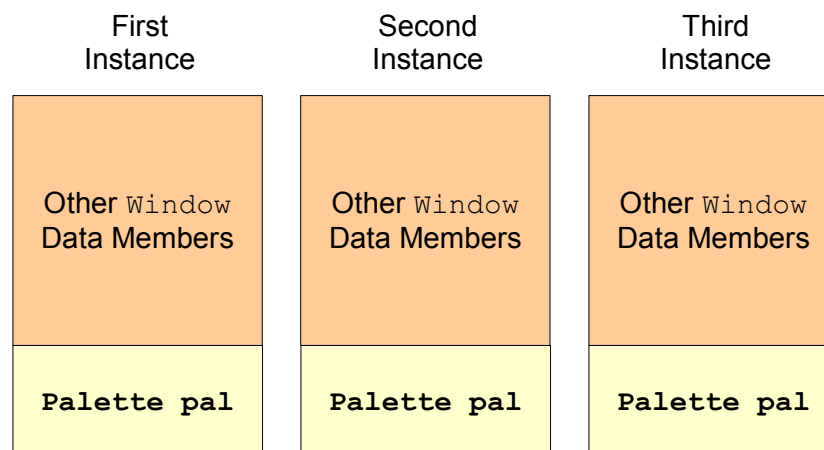
Suppose that we're developing a windowed operating system and want to write the code that draws windows on the screen. We decide to create a class `Window` that exports a `drawWindow` function. In order to display the window correctly, `drawWindow` needs access to a `Palette` object that performs primitive rendering operations like drawing lines, arcs, and filled polygons. Assume that we know that the window will always be drawn with the same `Palette`. Given this description, we might initially design `Window` so that it has a `Palette` as a data member, as shown here:

```
class Window
{
public:
    /* ... constructors, destructors, etc. ... */

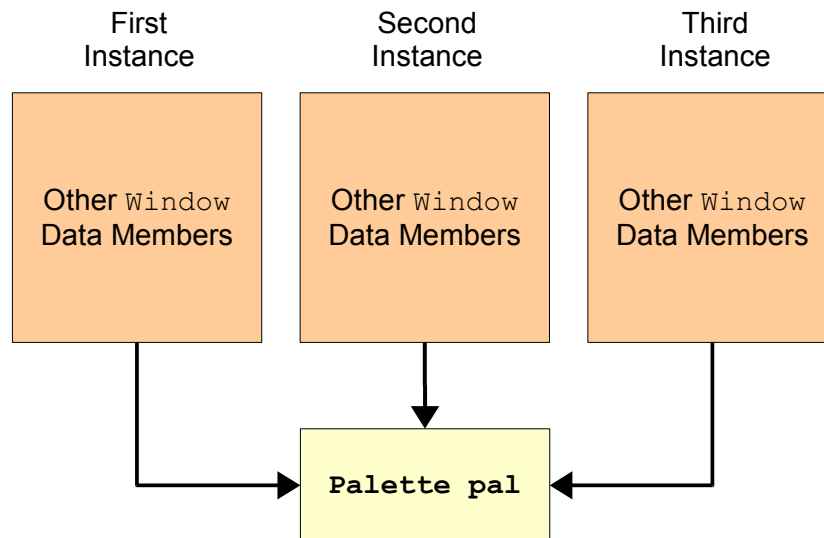
    /* All windows can draw themselves. */
    void drawWindow();
private:
    /* ... other data members ... */
    Palette pal;
};
```

Now, every window has its own palette and can draw itself appropriately.

There's nothing fundamentally wrong with this setup, but it contains a small flaw. Let's suppose that we have three different window objects. In memory, those objects would look like this:



Since `pal` is a data member of `Window`, every `Window` has its own `Palette`. There might be an arbitrary number of windows on screen at any time, but there's only one screen and it doesn't make sense for every window to have its own palette. After all, each window is likely to use a similar set of colors as those used by every other window, and it seems more reasonable for every window to share a single palette, as shown here:



How can we model this in code? Using the techniques so far, we have few options. First, we could create a global `Palette` object, then have each `Window` use this global `Palette`. This is a particularly bad choice for two reasons:

- **It uses global variables.** Independently of any other strengths and weaknesses of this approach, global variables are a big programming no-no. Globals can be accessed and modified anywhere in the program, making debugging difficult should problems arise. It is also possible to inadvertently reference global variables inside of unrelated functions, leading to subtle bugs that can take down the entire program.
- **It lacks encapsulation.** Because the `Palette` is a global variable, other parts of the program can modify the `Window` `Palette` without going through the `Window` class. This leads to the same sorts of problems possible with public data members: class invariants breaking unexpectedly, code written with one version of `Window` breaking when the `Window` is updated, etc.

Second, we could have each `Window` object contain a *pointer* to a `Palette` object, then pass a shared `Palette` as a parameter to each instance of `Window`. For example, we could design the class like this:

```

class Window
{
public:
    Window(Palette* p, /* ... */);
    /* ... other constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();
private:
    /* ... other data members ... */
    Palette* pal;
};
  
```

This allows us to share a single `Palette` across multiple `Windows` and looks remarkably like the above diagram. However, this approach has its weaknesses:

- **It complicates window creation.** Let's think about how we would go about creating Windows with this setup. Before creating our first Window, we'd need to create a Palette to associate with it, as shown here:

```
Palette* p = new Palette;  
Window* w = new Window(p, /* ... */);
```

If later we want to create more Windows, we'd need to keep track of the original Palette we used so that we can provide it as a parameter to the Window constructor. This means that any part of the program responsible for Window management needs to know about the shared Palette.

- **It violates encapsulation.** Clients of Window shouldn't have to know how Windows are implemented, and by requiring Window users to explicitly manage the shared Palette we're exposing too much detail about the Window class. This approach also locks us in to a fixed implementation. For example, what if we want to switch from Palette to a TurboPalette that draws twice as quickly? With the current approach all Window clients would need to upgrade their code to match the new implementation.
- **It complicates resource management.** Who is responsible for cleaning up the Window Palette at the end of the program? Window clients shouldn't have to, since the Palette really belongs to the Window class. But no particular Window owns the Palette, since each instance of Window shares a single Palette. There are systems we could use to make cleanup work correctly (see the later extended example on smart pointers for one possibility), but they increase program complexity.

Both of these approaches have their individual strengths, but have drawbacks that outweigh their benefits. Let's review exactly what we're trying to do. We'd like to have a single Palette that's shared across multiple different Windows. Moreover, we'd like this Palette to obey all of the rules normally applicable to class design: it should be encapsulated and it should be managed by the class rather than clients. Using the techniques we've covered so far it is difficult to construct a solution with these properties. For a clean solution, we'll need to introduce a new language feature: *static data members*.

Static Data Members

Static data members are data members associated with a class as a whole rather than a particular instance of that class. In the above example with Window and Palette, the window Palette is associated with *Windows in general* rather than any one specific Window object and is an ideal candidate for a static data member.

In many ways static data members behave similarly to regular data members. For example, if a class has a private static data member, only member functions of the class can access that variable. However, static data members behave differently from other data members because there is only one copy of each static data member. Each instance of a class containing a static data member shares the same version of that data member. That is, if a single class instance changes a static data member, the change affects all instances of that class.

The syntax for declaring static data members is slightly more complicated than for declaring nonstatic data members. There are two steps: *declaration* and *definition*. For example, if we want to create a static Palette object inside of Window, we could *declare* the variable as shown here:

```

class Window
{
public:
    /* ... constructors, destructors, etc. ... */

    /* All windows can draw themselves. */
    void drawWindow();
private:
    /* ... other data members ... */
    static Palette sharedPal;
};

```

Here, `sharedPal` is declared as a static data member using the `static` keyword. But while we've *declared* `sharedPal` as a static data member, we haven't *defined* `sharedPal` yet. Much in the same way that functions are separated into prototypes (declarations) and implementations (definitions), static data members have to be both declared inside the class in which they reside and defined inside the `.cpp` file associated with that class. For the above example, inside the `.cpp` file for the `Window` class, we would write

```

Palette Window::sharedPal;

```

There are two important points to note here. First, when defining the `static` variable, we must use the fully-qualified name (`Window::sharedPal`) instead of just its local name (`sharedPal`). Second, we do *not* repeat the `static` keyword during the variable declaration – otherwise, the compiler will think we're doing something completely different (see the “More to Explore” section). You may have noticed that even though `Window::sharedPal` is private we're still allowed to use it outside the class. This is only legal during definition, and outside of this one context it is illegal to use `Window::sharedPal` outside of the `Window` class.

In some circumstances you may want to create a class containing a static data member where the data member needs to take on an initial value. For example, if we want to create a class containing an `int` as a static data member, we would probably want to initialize the `int` to a particular value. Given the following class declaration:

```

class MyClass
{
public:
    void doSomething();
private:
    static int myStaticData;
};

```

It is perfectly legal to initialize `myStaticData` as follows:

```

int MyClass::myStaticData = 137;

```

As you'd expect, this means that `myStaticData` initially holds the value 137.

Although the syntax for creating a static data member can be intimidating, once initialized static data members look just like regular data members. For example, consider the following member function:

```

void MyClass::doSomething()
{
    ++myStaticData; // Modifies myStaticData for all classes
}

```

Nothing here seems all that out-of-the-ordinary and this code will work just fine. Note, however, that modifications to `myStaticData` are visible to all other instances of `MyClass`.

Let's consider another example where static data members can be useful. Suppose that you're debugging the windowing code from before and you're pretty sure that you've forgotten to `delete` all instances of `Window` that you've allocated with `new`. Since C++ won't give you any warnings about this, you'll need to do the instance counting yourself. The number of active instances of a class is class-specific information that doesn't pertain to any specific instance of the object, and this is the perfect spot to use static data members. To handle our instance counting, we'll modify the `Window` definition as follows:

```
class Window
{
public:
    /* ... constructors, destructors, etc. ... */

    void drawWindow();
private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};
```

We'll also define the variable outside the class as

```
int Window::numInstances = 0;
```

We know that whenever we create a new instance of a class, the class's constructor will be called. This means that if we increment `numInstances` inside the `Window` constructor, we'll correctly track the number of times the a `Window` has been created. Thus, we'll rewrite the `Window` constructor as follows:

```
Window::Window(/* ... */)
{
    /* ... All older initialization code ... */
    ++numInstances;
}
```

Similarly, we'll decrement `numInstances` in the `Window` destructor. We'll also have the destructor print out a message if this is the last remaining instance so we can see how many instances are left:

```
Window::~Window()
{
    /* ... All older cleanup code ... */
    --numInstances;
    if(numInstances == 0)
        cout << "No more Windows!" << endl;
}
```

Static Member Functions

Inside of member functions, a special variable called `this` acts as a pointer to the current object. Whenever you access a class's instance variables inside a member function, you're really accessing the instance variables of the `this` pointer. For example, given the following `Point` class:

* This is not meant as a slight to Microsoft.

```

class Point
{
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;
private:
    int x, y;
};

```

If we implement the `Point` constructor as follows:

```

Point::Point(int xLoc, int yLoc)
{
    x = xLoc;
    y = yLoc;
}

```

This code is equivalent to

```

Point::Point(int xLoc, int yLoc)
{
    this->x = xLoc;
    this->y = yLoc;
}

```

How does C++ know what value `this` refers to? The answer is subtle but important. Suppose that we have a `Point` object called `pt` and that we write the following code:

```
int x = pt.getX();
```

The C++ compiler converts this into code along the lines of

```
int x = Point::getX(&pt);
```

Where `Point::getX` is prototyped as

```
int Point::getX(Point *const this);
```

This is not legal C++ code, but illustrates what's going on behind the scenes whenever you call a member function.

The mechanism behind member function calls should rarely be of interest to you as a programmer. However, the fact that an N -argument member function is really an $(N+1)$ -argument free function can cause problems in a few places. For example, suppose that we want to provide a comparison function for `Points` that looks like this:

```

class Point
{
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

    bool compareTwoPoints(const Point& one, const Point& two) const
private:
    int x, y;
};

```



```
bool Point::compareTwoPoints(const Point& one, const Point& two) const
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

If you have a `vector<Point>` that you'd like to pass to the STL `sort` algorithm, you'll run into trouble if you try to use this syntax:

```
sort(myVector.begin(), myVector.end(), &Point::compareTwoPoints); // Error!
```

The problem is that `sort` expects a comparison function that takes two parameters and returns a `bool`. However, `Point::compareTwoPoints` takes *three* parameters: two points to compare and an invisible “this” pointer. Thus the above code will generate an error.

If you want to define a comparison or predicate function inside of a class, you'll want that member function to not have an invisible `this`. What does this mean from a practical standpoint? A member function without a `this` pointer does not have a receiver object, and thus can only operate on its parameters and any static data members of the class it's declared in (since that data is particular to the class rather than any particular instance). Functions of this sort are called *static member functions* and can be created using the `static` keyword. In the above example with `Point`, we could create the `Point` comparison function as a static member function using the following syntax:

```
class Point
{
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

    static bool compareTwoPoints(const Point& one, const Point& two) const;
private:
    int x, y;
};

bool Point::compareTwoPoints(const Point& one, const Point& two) const
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

Now, the above call to `sort` will work since `compareTwoPoints` would no longer have a `this` pointer.

Unlike static data members, when writing static member functions you do not need to separate the code out into a separate declaration and definition. You may want to do so anyway, though.

Let's return to our earlier example about tracking the number of `Window` instances currently in use. While it's nice that the destructor prints out a message when the last instance has been cleaned up, we'd prefer a more robust model where we can check how many more copies of the class exist. This function is not specific to a particular class instance, so we'll make this function static. We'll call this function `getRemainingInstances` and implement it as shown here:

```

class Window
{
public:
    /* ... constructors, destructors, etc. ... */
    void drawWindow();

    static int getRemainingInstances();
private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};

Palette Window::sharedPal;
int Window::numInstances = 0;

int Window::getRemainingInstances()
{
    return numInstances;
}

```

As with static data, note that when defining static member functions, you omit the `static` keyword. Only put `static` inside the class declaration.

You can invoke static member functions either using the familiar `object.method` syntax, or you can use the fully qualified name of the function. For example, with the above example, we could check how many remaining instances there were of the `MyClass` class by calling `getRemainingInstances` as follows:

```
cout << MyClass::getRemainingInstances() << endl;
```

const and static

Unfortunately, the `const` and `static` keywords do not always interact intuitively. One of the biggest issues to be aware of is that `const` member functions can modify `static` data. For example, consider the following class:

```

class ConstStaticClass
{
public:
    void constFn() const;
private:
    static int staticData;
};

int ConstStaticClass::staticData = 0;

```

Then the following implementation of `constFn` is completely valid:

```

void ConstStaticClass::constFn() const
{
    ++staticData;
}

```

Although the above implementation of `constFn` increments a static data member, the above code will compile and run without any problems. The reason is that the code doesn't modify the receiver object. Static data members are not associated with a particular class instance, so modifications to static data members do not change the state of any one instance of the class.

Additionally, since `static` member functions don't have a `this` pointer, they cannot be declared `const`. In the case of `getNumInstances`, this means that although the function doesn't modify any class data, we still cannot mark it `const`.

Integral Class Constants

There is one other topic concerning the interaction of `const` and `static`: class constants. Suppose we want to make a constant variable accessible only in the context of a class. What we want is a variable that's `const`, so it's immutable, and `static`, so that all copies of the class share the data. It's legal to declare these variables like this:

```
class ClassConstantExample
{
public:
    /* Omitted. */
private:
    static const int MyConstant;
};

const int ClassConstantExample::MyConstant = 137;
```

Note the `const` in the definition of `ClassConstantExample::MyConstant`.

However, since the double declaration/definition can be a bit tedious, C++ has a built-in shorthand you can use when declaring class constants of integral types. That is, if you have a `static const int` or a `static const char`, you can condense the definition and declaration into a single statement by writing:

```
class ClassConstantExample
{
public:
    /* Omitted. */
private:
    static const int MyConstant = 137; // Condense into a single line
};
```

This shorthand is common in professional code. Be careful when using the shorthand, though, because some older compilers won't correctly interpret it. Also, be aware that this only works with *integral types*, so you cannot initialize a `static const double` or `static const float` this way.

More to Explore

While this chapter discusses `static` in relation to classes, there are three other meanings of `static`. The first is *static linkage specification*, which means that the specified member function or global variable can only be accessed by functions defined in the current file. Another is *static local variables*, local variables whose values are preserved between function calls. These techniques are common in practice, so you should be sure to check a reference text for some more info on their subtleties.

Practice Problems

1. Explain why `static` member functions cannot be marked `const`.
2. Write a class `UniquelyIdentified` such that each instance of the class has a unique ID number determined by taking the ID number of the previous instance and adding one. The first instance should have ID number 1. Thus the third instance of the class will have ID 3, the ninety-sixth instance 96, etc. Also write a `const`-correct member function `getUniqueID` that returns the class's unique ID. Don't worry about reusing older IDs if their objects go out of scope. ♦

3. The C header file `<cstdlib>` exports two functions for random number generation – `srand`, which seeds the randomizer, and `rand`, which generates a pseudorandom `int` between 0 and the constant `RAND_MAX`. To make the pseudorandom values of `rand` appear truly random, you can seed the randomizer using the value returned by the `time` function exported from `<ctime>`. The syntax is `srand((unsigned int)time(NULL))`. Write a class `RandomGenerator` that exports a function `next` that returns a random double in the range `[0, 1)`. When created, the `RandomGenerator` class should seed the randomizer with `srand` only if a previous instance of `RandomGenerator` hasn't already seeded it.
4. Should you specify values for static data members in an initializer list? Why or why not?
5. Should you ever mark static data members `mutable`? Why or why not?

Chapter 19: Conversion Constructors

When designing classes, you might find that certain data types can logically be converted into objects of the type you're creating. For example, when designing the C++ `string` class, you might note that `char * C` strings could have a defined conversion to `string` objects. In these situations, it may be useful to define *implicit conversions* between the two types. To define implicit conversions, C++ uses *conversion constructors*, constructors that accept a single parameter and initialize an object to be a copy of that parameter.

While useful, conversion constructors have several major idiosyncrasies, especially when C++ interprets normal constructors as conversion constructors. This chapter explores implicit type conversions, conversion constructors, and how to prevent coding errors stemming from inadvertent conversion constructors.

Implicit Conversions

In C++, an *implicit conversion* is a conversion from one type to another that doesn't require an explicit typecast. Perhaps the simplest example is the following conversion from an `int` to a `double`:

```
double myDouble = 137 + 2.71828;
```

Here, even though 137 is an `int` while 2.71828 is a `double`, C++ will implicitly convert it to a `double` so the operation can proceed smoothly.

When C++ performs implicit conversions, it does not “magically” figure out how to transform one data type into another. Rather, it creates a temporary object of the correct type that's initialized to the value of the implicitly converted object. Thus the above code is equivalent to

```
double temp = (double)myInt;
double myDouble = temp + 2.71828;
```

It's important to remember that when using implicit conversions you are creating temporary objects. With primitive types this is hardly noticeable, but makes a difference when working with classes. For example, consider the following code:

```
string myString = "This ";
string myOtherString = myString + "is a string";
```

Note that in the second line, we're adding a C++ `string` to a C `char * string`. Thus C++ will implicitly convert “is a string” into a C++ `string` by storing it in a temporary object. The above code, therefore, is equivalent to

```
string myString = "This ";
string tempStr = "is a string";
string myOtherString = myString + tempStr;
```

Notice that in both of the above examples, at some point C++ needed a way to initialize a temporary object to be equal to an existing object of a different type. In the first example, we made a temporary `double` that was equal to an `int`, and in the second, a temporary `string` equal to a `char *`.^{*} When C++ performs these conversions, it uses a special function called a *conversion constructor* to initialize the new object. Conversion constructors

^{*} Technically speaking, this isn't quite what happens, since there's a special form of the `+` operator that works on a mix of C strings and C++ strings. However, for this purposes of this discussion, we can safely ignore this.

are simply class constructors that accept a single parameter and initialize the new object to a copy of the parameter. In the `double` example, the newly-created `double` had the same value as the `int` parameter. With the C++ `string`, the temporary `string` was equivalent to the C string.

C++ will invoke conversion constructors whenever an object of one type is used in an expression where an object of a different type is expected. Thus, if you pass a `char *` to a function accepting a C++ `string`, the `string` will be initialized to the `char *` in its conversion constructor. Similarly, if you have a function like this one:

```
string MyFunction()
{
    return "This is a string!";
}
```

The temporary object created for the return value will be initialized to the C string “This is a string!” using the conversion constructor.

Writing Conversion Constructors

To see how to write conversion constructors, we'll use the example of a `CString` class that's essentially our own version of the C++ `string` class. Internally, `CString` stores the string as a C string called `theString`. Since we'd like to define an implicit conversion from `char *` to `CString`, we'll declare a conversion constructor, as shown below:

```
class CString
{
public:
    CString(const char* other);
    /* Other member functions. */
private:
    char* theString;
};
```

Then we'd implement the conversion constructor as

```
CString::CString(const char* other)
{
    /* Allocate space and copy over the string. */
    theString = new char[strlen(other) + 1];
    strcpy(theString, other);
}
```

Now, whenever we have a `char *` C string, we can implicitly convert it to a `CString`.

In the above case, we defined an implicit conversion from `char *` C strings to our special class `CString`. However, it's possible to define a second conversion from a C++ `string` to our new `CString` class. In fact, C++ allows you to provide conversion constructors for any number of different types that may or may not be primitive types.

Here's a modified `CString` interface that provides two conversion constructors from `string` and `char *`:

```

class CString
{
public:
    CString(const string& other);
    CString(const char*   other);
    /* ... other member functions... */

private:
    char *theString;
};

```

A Word on Readability

When designing classes with conversion constructors, it's easy to get carried away by adding too many implicit conversions. For example, suppose that for the `CString` class we want to define a conversion constructor that converts `ints` to their string representations. This is completely legal, but can result in confusing or unreadable code. For example, if there's an implicit conversion from `ints` to `CStrings`, then we can write code like this:

```
CString myStr = myInt + 137;
```

The resulting `CString` would then hold a string version of the value of `myInt + 137`, not the string composed of the concatenation of the value of `myInt` and the string “137.” This can be a bit confusing and can lead to counterintuitive code. Worse, since C++ does not normally define implicit conversions between numeric and string types, people unfamiliar with the `CString` implementation might get confused by lines assigning `ints` to `CStrings`.

In general, when working with conversion constructors, make sure that the conversion is intuitive and consistent with major C++ conventions. If not, consider using non-constructor member functions. For example, if we would like to be able to convert `int` values into their string representations, we might want to make a global function `intToString` that performs the conversion. This way, someone reading the code could explicitly see that we're converting an `int` to a `CString`.

Problems with Conversion Constructors

While conversion constructors are quite useful in a wide number of circumstances, the fact that C++ automatically treats all single-parameter constructors as conversion constructors can lead to convoluted or nonsensical code.

One of my favorite examples of “conversion-constructors-gone-wrong” comes from an older version of the CS106B/X ADT class libraries. Originally, the CS106B/X `Vector` was defined as

```

template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 10); // Hint about the size of the Vector
    /* ... */
};

```

Nothing seems all that out-of-the-ordinary here – we have a `Vector` template class that lets you give the class a hint about the number of elements you will be storing in it. However, because the constructor accepts a single parameter, C++ will interpret it as a conversion constructor and thus will let us implicitly convert from `ints` to `Vectors`. This can lead to some very strange behavior. For example, given the above class definition, consider the following code:

```
Vector<int> myVector = 137;
```

This code, while nonsensical, is legal and equivalent to `Vector<int> myVector(137)`. Fortunately, this probably won't cause any problems at runtime – it just doesn't make sense in code.

However, suppose we have the following code:

```
void DoSomething(Vector<int>& myVector)
{
    myVector = NULL;
}
```

This code is totally legal even though it makes no logical sense. Since `NULL` is `#defined` to be `0`, the above code will create a new `Vector<int>` initialized with the parameter `0` and then assign it to `myVector`. In other words, the above code is equivalent to

```
void DoSomething(Vector<int>& myVector)
{
    Vector<int> tempVector(0);
    myVector = tempVector;
}
```

`tempVector` is empty when it's created, so when we assign `tempVector` to `myVector`, we'll set `myVector` to the empty vector. Thus the nonsensical line `myVector = 0` is effectively an obfuscated call to `myVector.clear()`.

This is a quintessential example of why conversion constructors can be dangerous. When writing single-argument constructors, you run the risk of letting C++ interpret your constructor as a conversion constructor.

explicit

To prevent problems like the one described above, C++ provides the `explicit` keyword to indicate that a constructor must not be interpreted as a conversion constructor. If a constructor is marked `explicit`, it indicates that the constructor should not be considered for the purposes of implicit conversions. For example, let's look at the current version of the CS106B/X `Vector`, which has its constructor marked `explicit`:

```
template <typename ElemType> class Vector
{
public:
    explicit Vector(int sizeHint = 10); // Hint the size of the Vector
    /* ... */
};
```

Now, if we write code like

```
Vector<int> myVector = 10;
```

We'll get a compile-time error since there's no implicit conversion from `int` to `Vector<int>`. However, we can still write

```
Vector<int> myVector(10);
```

Which is what we were trying to accomplish in the first place. Similarly, we eliminate the `myVector = 0` error, and a whole host of other nasty problems.

When designing classes, if you have a single-argument constructor that is not intended as a conversion function, you *must* mark it `explicit` to avoid running into the “implicit conversion” trap. While indeed this is more work for you as an implementer, it will make your code safer and more stable.

Practice Problems

These practice problems concern a `RationalNumber` class that encapsulates a rational number (that is, a number expressible as the quotient of two integers). `RationalNumber` is declared as follows:

```
class RationalNumber
{
public:
    RationalNumber(int num = 0, int denom = 1) :
        numerator(num), denominator(denom) {}

    double getValue() const
    {
        return static_cast<double>(numerator) / denominator;
    }

    void setNumerator(int value)
    {
        numerator = value;
    }
    void setDenominator(int value)
    {
        denominator = value;
    }
private:
    int numerator, denominator;
};
```

The constructor to `RationalNumber` accepts two parameters that have default values. This means that if you omit one or more of the parameters to `RationalNumber`, they'll be filled in using the defaults. Thus all three of the following lines of code are legal:

```
RationalNumber zero; // Value is 0 / 1 = 0
RationalNumber five(5); // Value is 5 / 1 = 5
RationalNumber piApprox(355, 113); // Value is 355/113 = 3.1415929203...
```

1. Explain why the `RationalNumber` constructor is a conversion constructor.
2. Write a `RealNumber` class that encapsulates a real number (any number on the number line). It should have a conversion constructor that accepts a `double` and a default constructor that sets the value to zero. (*Note: You only need to write one constructor. Use `RationalNumber` as an example*)
3. Write a conversion constructor that converts `RationalNumbers` into `RealNumbers`.

General questions:

4. If a constructor has two or more arguments and no default values, can it be a conversion constructor?
5. C++ will apply at most one implicit type conversion at a time. That is, if you define three types A, B, and C such that A is implicitly convertible to B and B is implicitly convertible to C, C++ will not automatically convert objects of type A to objects of type C. Give an reason for why this might be. (*Hint: Add another implicit conversion between these types*) ♦
6. Implement the conversion constructor converting a C++ string to our special `CString` class.

Chapter 20: Copy Constructors and Assignment Operators

Consider the STL `vector`. Internally, `vector` is backed by a dynamically-allocated array whose size grows when additional space is needed for more elements. For example, a ten-element `vector` might store those elements in an array of size sixteen, increasing the array size if we call `push_back` seven more times. Given this description, consider the following code:

```
vector<int> one(NUM_INTS);
for(int k = 0; k < one.size(); ++k)
    one.push_back(k);

vector<int> two = one;
```

In the first three lines, we fill `one` with the first `NUM_INTS` integers, and in the last line we create a new `vector` called `two` that's a copy of `one`. How does C++ know how to correctly copy the data from `one` into `two`? It can't simply copy the pointer to the dynamically-allocated array from `one` into `two`, since that would cause `one` and `two` to share the same data and changes to one `vector` would show up in the other. Somehow C++ is aware that to copy a `vector` it needs to dynamically allocate a new array of elements, then copy the elements from the source to the destination. This is not done by magic, but by two special functions called the *copy constructor* and the *assignment operator*, which control how to copy instances of a particular class.

Used correctly, copy constructors and assignment operators let you fine-tune the way that C++ copies instances of your custom types. As you'll see in a later extended example, these functions can also be used to define special copy behavior to elegantly and efficiently manage resources. However, copy constructors and assignment operators are among the most difficult features of the C++ language and unless you understand their subtleties and complexities you are likely to end up with code containing dangerous bugs. This chapter introduces copy semantics, discusses the copy constructor and assignment operator, shows how to implement them, and highlights potential sources of error.

Two Styles of Copying: Assignment and Initialization

Before discussing the particulars of the copy constructor and assignment operator, we first need to dissect exactly how an object can be copied. In order to copy an object, we first have to answer an important question – where do we put the copy? Do we store it in a new object, or do we reuse an existing object? These two options are fundamentally different from one another and C++ makes an explicit distinction between them. The first option – putting the copy into a new location – creates the copy by *initializing* the new object to the value of the object to copy. The second – storing the copy in an existing variable – creates the copy by *assigning* the existing object the value of the object to copy. What do these two copy mechanisms look like in C++? That is, when is an object initialized to a value, and when is it assigned a new value?

In C++, initialization can occur in three different places:

1. *A variable is created as a copy of an existing value.* For example, suppose we write the following code:

```
MyClass one;
MyClass two = one;
```

Here, since `two` is told to hold the value of `one`, C++ will *initialize* `two` as a copy of `one`. Although it looks like we're assigning a value to `two` using the `=` operator, since it is a newly-created object, the `=` indicates initialization, not assignment. In fact, the above code is equivalent to the more explicit initialization code below:

```
MyClass one;
MyClass two(one);    // Identical to above.
```

This syntax makes more clear that `two` is being created as a copy of `one`, indicating initialization rather than assignment.

2. *An object is passed by value to a function.* Consider the following function:

```
void MyFunction(MyClass arg)
{
    /* ... */
}
```

If we write

```
MyClass mc;
MyFunction(mc);
```

Then the function `MyFunction` somehow has to set up the value of `arg` inside the function to have the same value as `mc` outside the function. Since `arg` is a new variable, C++ will *initialize* it as a copy of `mc`.

3. *An object is returned from a function by value.* Suppose we have the following function:

```
MyClass MyFunction()
{
    MyClass mc;
    return mc;
}
```

When the statement `return mc` executes, C++ needs to return the `mc` object from the function. However, `mc` is a local variable inside the `MyFunction` function, and to communicate its value to the `MyFunction` caller C++ needs to create a copy of `mc` before it is lost. This is done by creating a temporary `MyClass` object for the return value, then *initializing* it to be a copy of `mc`.

Notice that in all three cases, initialization took place because some new object was created as a copy of an existing object. In the first case this was a new local variable, in the second a function parameter, and in the third a temporary object.

Assignment in C++ is much simpler than initialization and only occurs if an existing object is explicitly assigned a new value. For example, the following code will *assign* `two` the value of `one`, rather than *initializing* `two` to `one`:

```
MyClass one, two;
two = one;
```

It can be tricky to differentiate between initialization and assignment because in some cases the syntax is almost identical. For example, if we rewrite the above code as

```
MyClass one;
MyClass two = one;
```

`two` is now initialized to `one` because it is declared as a new variable. Always remember that the assignment only occurs when giving an existing object a new value.

Why is it important to differentiate between assignment and initialization? After all, they're quite similar; in both cases we end up with a new copy of an existing object. However, assignment and initialization are fundamentally different operations. When *initializing* a new object as a copy of an existing object, we simply need to copy the existing object into the new object. When *assigning* an existing object a new value, the existing object's value ceases to be and we must make sure to clean up any resources the object may have allocated before setting it to the new value. In other words, initialization is a straight copy, while assignment is cleanup followed by a copy. This distinction will become manifest in the code we will write for the copy functions later in this chapter.

Copy Functions: Copy Constructors and Assignment Operators

Because initialization and assignment are separate tasks, C++ handles them through two different functions called the *copy constructor* and the *assignment operator*. The copy constructor is a special constructor responsible for initializing new class instances as copies of existing instances of the class. The assignment operator is a special function called an *overloaded operator* (see the chapter on operator overloading for more details) responsible for assigning the receiver object the value of some other instance of the object. Thus the code

```
MyClass one;
MyClass two = one;
```

will initialize `two` to `one` using the copy constructor, while the code

```
MyClass one, two;
two = one;
```

will assign `one` to `two` using the assignment operator.

Syntactically, the copy constructor is written as a one-argument constructor whose parameter is another instance of the class accepted by reference-to-const. For example, given the following class:

```
class MyClass
{
public:
    MyClass();
    ~MyClass();

    /* ... */
};
```

The copy constructor would be declared as follows:

```
class MyClass
{
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor

    /* ... */
};
```

The syntax for the assignment operator is substantially more complex than that of the copy constructor because it is an *overloaded operator*, a special function that defines how the built-in `=` operator applies to objects of this class. We'll cover operator overloading in a later chapter, but for now we can use the fact that the assignment operator is a function named `operator =`. The function name `operator =` can be written with as much

whitespace as you'd like between operator and =, so operator= and operator = are both valid. For reasons that will become clearer later in the chapter, the assignment operator should accept as a parameter another instance of the class by reference-to-const and should return a non-const reference to an object of the class type. For a concrete example, here's the assignment operator for MyClass:

```
class MyClass
{
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor
    MyClass& operator = (const MyClass& other); // Assignment operator
    /* ... */
};
```

We'll defer discussing exactly why this syntax is correct until later, so for now you should take it on faith.

What C++ Does For You

Unless you specify otherwise, C++ will automatically provide any class you write with a basic copy constructor and assignment operator that invoke the copy constructors and assignment operators of all the class's data members. In many cases, this is exactly what you want. For example, consider the following class:

```
class DefaultClass
{
public:
    /* ... */
private:
    int myInt;
    string myString;
};
```

Suppose you have the following code:

```
DefaultClass one;
DefaultClass two = one;
```

The line `DefaultClass two = one` will invoke the copy constructor for `DefaultClass`. Since we haven't explicitly provided our own copy constructor, C++ will initialize `two.myInt` to the value of `one.myInt` and `two.myString` to `one.myString`. Since `int` is a primitive and `string` has a well-defined copy constructor, this code is totally fine.

However, in many cases this is not the behavior you want. Let's consider the example of a class `CString` that acts as a wrapper for a C string. Suppose we define `CString` as shown here:

```
class CString
{
public:
    CString();
    ~CString();
    /* Note: No copy constructor or assignment operator */
private:
    char* theString;
};
```

Here, if we rely on C++'s default copy constructor or assignment operator, we'll run into trouble. For example, consider the following code:

```
CString one;  
CString two = one;
```

Because we haven't provided a copy constructor, C++ will initialize `two.theString` to `one.theString`. Since `theString` is a `char *`, instead of getting a deep copy of the string, we'll end up with two pointers to the same C string. Thus changes to `one` will show up in `two` and vice-versa. This is dangerous, especially when the destructors for both `one` and `two` try to deallocate the memory for `theString`. In situations like these, you'll need to override C++'s default behavior by providing your own copy constructors and assignment operators.

There are a few circumstances where C++ does not automatically provide default copy constructors and assignment operators. If your class contains a reference or `const` variable as a data member, your class will not automatically get a copy constructor and assignment operator. Similarly, if your class has a data member that doesn't have a copy constructor or assignment operator (for example, an `ifstream`), your class won't be copyable. There is one other case involving inheritance where C++ won't automatically create the copy functions for you, and in the chapter on inheritance we'll see how to exploit this to disable copying.

The Rule of Three

There's a well-established C++ principle called the “rule of three” that identifies most spots where you'll need to write your own copy constructor and assignment operator. If this were a math textbook, you'd probably see the rule of three written out like this:

Theorem (*The Rule of Three*): If a class has any of the following three member functions:

- Destructor
- Copy Constructor
- Assignment Operator

Then that class should have all three of those functions.

Corollary: If a class has a destructor, it should also have a copy constructor and assignment operator.

The rule of three holds because in almost all situations where you have any of the above functions, C++'s default behavior won't correctly manage your objects. In the above example with `CString`, this is the case because copying the `char *` pointer doesn't actually duplicate the C string. Similarly, if you have a class holding an open file handle, making a shallow copy of the object might cause crashes further down the line as the destructor of one class closed the file handle, corrupting the internal state of all “copies” of that object.

Both C++ libraries and fellow C++ coders will expect that, barring special circumstances, all objects will correctly implement the three above functions, either by falling back on C++'s default versions or by explicitly providing correct implementations. Consequently, you *must* keep the rule of three in mind when designing classes or you will end up with insidious or seemingly untraceable bugs as your classes start to destructively interfere with each other.

Writing Copy Constructors

For the rest of this chapter, we'll discuss copy constructors and assignment operators through a case study of the `DebugVector` class. `DebugVector` is a modified version of the CS106B/X `Vector` whose constructor and destructor log information to `cout` whenever an instance of `DebugVector` is created or destroyed. That way, if you're writing a program that you think might be leaking memory, you can check the program output to make sure that all your `DebugVectors` are cleaned up properly.

The class definition for `DebugVector` looks like this:

```
template <typename T> class DebugVector
{
public:
    DebugVector();
    DebugVector(const DebugVector& other);           // Copy constructor
    DebugVector& operator =(const DebugVector& other); // Assignment operator
    ~DebugVector();

    /* ... other member functions ... */
private:
    T* array;
    int allocatedLength;
    int logicalLength;
    static const int BASE_SIZE = 16;
};
```

Internally, `DebugVector` is implemented as a dynamically-allocated array of elements. Two data members, `allocatedLength` and `logicalLength`, track the allocated size of the array and the number of elements stored in it, respectively. `DebugVector` also has a class constant `BASE_SIZE` that represents the default size of the allocated array.

The `DebugVector` constructor is defined as

```
template <typename T> DebugVector<T>::DebugVector() : array(new T[BASE_SIZE]),
    allocatedLength(BASE_SIZE), logicalLength(0)
{
    /* Log the creation using some functions from <ctime> */
    time_t currentTime;
    time(&currentTime); // Fill in the current time.
    cout << "DebugVector created: " << ctime(&currentTime) << endl;
}
```

Note that we initialize `array` to point to a dynamically-allocated array of `BASE_SIZE` elements in the initializer list. We also use the `time` and `ctime` functions from `<ctime>` to record the time at which the `DebugVector` is created; consult a reference for more information about these functions. Similarly, the `DebugVector` destructor is

```
template <typename T> DebugVector<T>::~DebugVector()
{
    delete [] array;
    array = NULL;
    logicalLength = allocatedLength = 0;

    time_t currentTime;
    time(&currentTime);
    cout << "Destructor invoked: " << ctime(&currentTime) << endl;
}
```

Now, let's write the copy constructor. We know that we need to match the prototype given in the class definition, so we'll write that part first:

```
template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    /* ... */
}
```


Inside the copy constructor, we need to do two things. First, we must initialize the object so that we're holding a deep copy of the other `DebugVector`. Second, we should log creation information since we're instantiating a new `DebugVector`. Let's first initialize the object, then handle the logging later.

As with a regular constructor, with a copy constructor we should try to initialize as many instance variables as possible in the initializer list, both for readability and speed. In the case of `DebugVector`, while we can't completely set up the data members in the initializer list, we can still copy over the values of `logicalLength` and `allocatedLength`. For this chapter, however, we'll initialize these variables inside the body of the constructor, since both `logicalLength` and `allocatedLength` are primitives and don't get a performance boost from the initializer list. We thus have the following partial implementation of the copy constructor:

```
template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    /* ... */
}
```

Note that this implementation of the copy constructor sets `logicalLength` to `other.logicalLength` and `allocatedLength` to `other.allocatedLength`, even though `other.logicalLength` and `other.allocatedLength` explicitly reference private data members of the `other` object. This is legal because `other` is an object of type `DebugVector` and the copy constructor is a member function of `DebugVector`. A class can access both its private fields and private fields of other objects of the same type. This is called *sibling access* and is true of any member function, not just the copy constructor. If the copy constructor were not a member of `DebugVector` or if `other` were not a `DebugVector`, this code would not be legal.

Now, we'll make a deep copy of the other `DebugVector`'s elements by allocating a new array that's the same size as `other`'s and then copying the elements over. The code looks something like this:

```
template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; ++i)
        array[i] = other.array[i];

    /* ... */
}
```

Interestingly, since `DebugVector` is a template, it's unclear what the line `array[i] = other.array[i]` will actually do. If we're storing primitive types, then the line will simply copy the values over, but if we're storing objects, the line invokes the class's assignment operator. Notice that in both cases the object will be correctly copied over. This is one of driving forces behind defining copy constructors and assignment operators, since template code can assume that expressions like `object1 = object2` will be meaningful.

An alternative means for copying data over from the other object uses the STL `copy` algorithm. Recall that `copy` takes three parameters – two delineating an input range of iterators and one denoting the beginning of an output range – then copies the specified iterator range to the destination. Although designed to work on iterators, it is possible to apply STL algorithms directly to ranges defined by raw C++ pointers. Thus we could rewrite the copy constructor as follows:

```
template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.array, other.array + other.logicalLength, array);

    /* ... */
}
```

Here, the range spanned by `other.array` and `other.array + other.logicalLength` is the contents of the `other DebugVector`, and `array` is the beginning of the newly-allocated data we've reserved for this `DebugVector`. I personally find this syntax preferable to the explicit `for` loop, since it increases readability.

Finally, we need to log the creation of the new `DebugVector`. If you'll notice, the code for logging the `DebugVector`'s creation is already written for us in the default constructor. Unfortunately, unlike other object-oriented languages, in C++, a class with multiple constructors can't invoke one constructor from the body of another. To avoid code duplication, we'll therefore move the logging code into a separate private member function called `logCreation` that looks like this:

```
template <typename T> void DebugVector<T>::logCreation()
{
    time_t currentTime;
    time(&currentTime); // Fill in the current time.
    cout << "DebugVector created: " << ctime(&currentTime) << endl;
}
```

We then rewrite the default constructor for `DebugVector` to call `logCreation`, as shown below:

```
template <typename T> DebugVector<T>::DebugVector() : array(new T[BASE_SIZE]),
    allocatedLength(BASE_SIZE), logicalLength(0)
{
    logCreation();
}
```

And finally, we insert a call to `logCreation` into the copy constructor, yielding the final version:

```
template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.array, other.array + other.logicalLength, array);

    logCreation();
}
```

Writing Assignment Operators

We've now successfully written a copy constructor for our `DebugVector` class. Unfortunately, writing an assignment operator is significantly more involved than writing a copy constructor. C++ is designed to give you maximum flexibility when designing an assignment operator, and thus won't alert you if you've written a syntactically legal assignment operator that is completely incorrect. For example, consider this legal but incorrect assignment operator for an object of type `MyClass`:

```
void MyClass::operator =(const MyClass& other)
{
    cout << "I'm sorry, Dave. I'm afraid I can't copy that object." << endl;
}
```

Here, if we write code like this:

```
MyClass one, two;
two = one;
```

Instead of making `two` a deep copy of `one`, instead we'll get a message printed to the screen and `two` will remain unchanged. This is one of the dangers of a poorly-written assignment operator – code that looks like it does one thing can instead do something totally different. This section discusses how to correctly implement an assignment operator by starting with invalid code and progressing towards a correct, final version.

Let's start off with a simple but incorrect version of the assignment operator for `DebugVector`:

```
/* Many major mistakes here. Do not use this code as a reference! */
template <typename T> void DebugVector<T>::operator =(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.array, other.array + other.logicalLength, array);
}
```

This code is based off the copy constructor, which we used to initialize the object as a copy of an existing object. Unfortunately, this code contains a substantial number of mistakes that we'll need to correct before we end up with the final version of the function. Perhaps the most serious error here is the line `array = new T[allocatedLength]`. When the assignment operator is invoked, this `DebugVector` is already holding on to its own array of elements. However, this line orphans the old array and leaks memory. To fix this, before we make this object a copy of the one specified by the parameter, we'll take care of the necessary deallocations. If you'll notice, we've already written the necessary cleanup code in the `DebugVector` destructor. Rather than rewriting this code, we'll decompose out the generic cleanup code into a `clear` function, as shown here:

```
template <typename T> void DebugVector<T>::clear()
{
    delete [] array;
    array = NULL;
    logicalLength = allocatedLength = 0;
}
```

We can then rewrite the destructor as

```
template <typename T> DebugVector<T>::~~DebugVector()
{
    clear();

    time_t currentTime;
    time(&currentTime);
    cout << "Destructor invoked: " << ctime(&currentTime) << endl;
}
```

And we can insert this call to `clear` into our assignment operator as follows:

```

/* Still contains errors. */
template <typename T> void DebugVector<T>::operator =(const DebugVector& other)
{
    clear() ;

    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.array, other.array + other.logicalLength, array);
}

```

Along the same lines, you might have noticed that all of the code after the call to `clear` is exactly the same code we wrote inside the body of the copy constructor. This isn't a coincidence – in fact, in most cases you'll have a good deal of overlap between the assignment operator and copy constructor. Since we can't invoke our own copy constructor directly (or *any* constructor, for that matter), instead we'll decompose the copying code into a member function called `copyOther` as follows:

```

template <typename T> void DebugVector<T>::copyOther(const DebugVector& other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.array, other.array + other.logicalLength, array);
}

```

Now we can rewrite the copy constructor as

```

template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    copyOther(other) ;
    logCreation();
}

```

And the assignment operator as

```

/* Not quite perfect yet. */
template <typename T> void DebugVector<T>::operator =(const DebugVector& other)
{
    clear();
    copyOther(other) ;
}

```

This simplifies the copy constructor and assignment operator and highlights the general pattern of what the two functions should do. With a copy constructor, you'll copy the contents of the other object, then perform any remaining initialization. With an assignment operator, you'll clear out the receiver, then copy over the data from another object.

However, we're still not done yet. There are two more issues we need to fix with our current implementation of the assignment operator. The first one has to do with *self-assignment*. Consider, for example, the following code:

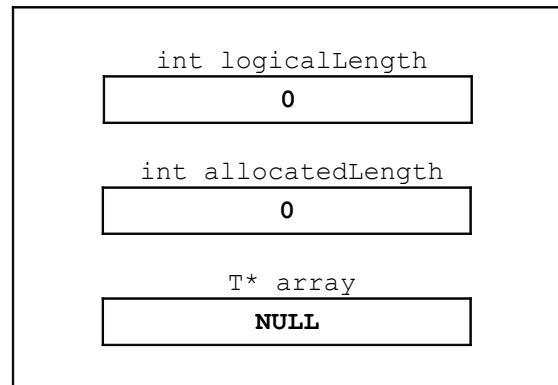
```

MyClass one;
one = one;

```

While this code might seem a bit silly, cases like this come up frequently when accessing elements indirectly through pointers or references. Unfortunately, with our current `DebugVector` assignment operator, this code will lead to unusual runtime behavior. To see why, let's trace out the state of our object when its assignment operator is invoked on itself.

At the start of the assignment operator, we call `clear` to clean out the object for the copy. During this call to `clear`, we deallocate the memory associated with the object and set both `logicalLength` and `allocatedLength` to zero. Thus our current object looks something like this:



Normally this isn't a problem, but remember that we're self-assigning, which means that the object referenced by `other` is the same object referenced by the `this` pointer. Consequently, since we erased all the contents of the current object, we also erased the contents of the `other` object. When we get to the code in `copyOther`, we'll end up duplicating an empty object, erasing the contents of the receiver object. The result is that our self-assignment is effectively a glorified call to `clear`.

When writing assignment operators, you must ensure that your code correctly handles self-assignment. While there are many ways we can do this, perhaps the simplest is to simply check to make sure that the object to copy isn't the same object pointed at by the `this` pointer. The code for this logic looks like this:

```
/* Not quite perfect yet - there's one more error */
template <typename T> void DebugVector<T>::operator =(const DebugVector& other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
}
```

Note that we check `if(this != &other)`. That is, we compare *the addresses* of the current object and the parameter. This will determine whether or not the object we're copying is exactly the same object as the one we're working with. Note that we did *not* write `if(*this != other)`, since this would do a semantic comparison between the objects, something we'll cover in a later chapter when discussing operator overloading. One last note is that code like this isn't required in the copy constructor, since we can't pass an object as a parameter to its own copy constructor.

There's one final bug we need to sort out, and it has to do with how we're legally allowed to use the `=` operator. Consider, for example, the following code:

```
MyClass one, two, three;
three = two = one;
```

This code is equivalent to `three = (two = one)`. Since our current assignment operator does not return a value, `(two = one)` does not have a value, so the above statement is meaningless and the code will not compile. We thus need to change our assignment operator so that performing an assignment like `two = one` yields a value that can then be assigned to other values. In particular, we should return a non-`const` reference to the receiver object; we'll discuss why this is in the chapter on operator overloading. The final version of our assignment operator is thus

```
/* The CORRECT version. */
template <typename T>
DebugVector<T>& DebugVector<T>::operator =(const DebugVector& other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```

One General Pattern

Although each class is different, in many cases the default constructor, copy constructor, assignment operator, and destructor will share a general pattern. Here is one possible skeleton you can fill in to get your copy constructor and assignment operator working.

```
MyClass::MyClass() : /* Fill in initializer list. */
{
    additionalInit();
}

MyClass::MyClass(const MyClass& other)
{
    copyOther(other);
    additionalInit();
}

MyClass& MyClass::operator =(const MyClass& other)
{
    if(this != &other)
    {
        clear();
        // Note: When we cover inheritance, there's one more step here.
        copyOther(other);
    }
    return *this;
}

MyClass::~MyClass()
{
    clear();
    additionalClosedown();
}
```

Semantic Equivalence and `copyOther` Strategies

Consider the following code snippet:

```
DebugVector<int> one;
DebugVector<int> two = one;
```

Here, we know that `two` is a copy of `one`, so the two objects should behave identically to one another. For example, if we access an element of `one`, we should get the same value as if we had accessed the corresponding element of `two` and vice-versa. However, while `one` and `two` are indistinguishable from each other in terms of functionality, their memory representations are not identical because `one` and `two` point to two different dynamically-allocated arrays. This raises the distinction between *semantic equivalence* and *bitwise equivalence*. Two objects are said to be bitwise equivalent if they have identical representations in memory. For example, any two `ints` with the value 137 are bitwise equivalent, and if we define a `pointT` struct as a pair of `ints`, any two `pointTs` holding the same values will be bitwise equivalent. Two objects are *semantically equivalent* if, like `one` and `two`, any operations performed on the objects will yield identical results. When writing a copy constructor and assignment operator, you attempt to convert an object into a semantically equivalent copy of another object. Consequently, you are free to pick any copying strategy that creates a semantically equivalent copy of the source object.

In the preceding section, we outlined one possible implementation strategy for a copy constructor and assignment operator that uses a shared function called `copyOther`. While in the case of the `DebugVector` it was relatively easy to come up with a working `copyOther` implementation, when working with more complicated objects, it can be difficult to devise a working `copyOther`. For example, consider the following class, which represents a mathematical set implemented as a linked list:

```
template <typename T> class ListSet
{
public:
    ListSet();
    ListSet(const ListSet& other);
    ListSet& operator =(const ListSet& other);
    ~ListSet();

    void insert(const T& toAdd);
    bool contains(const T& toFind) const;
private:
    struct cellT {
        T data;
        cellT* next;
    };
    cellT* list;

    void copyOther(const ListSet& other);
    void clear();
};
```

This `ListSet` class exports two functions, `insert` and `contains`, that insert an element into the list and determine whether the list contains an element, respectively. This class represents a mathematical set, an *unordered* collection of elements, so the underlying linked list need not be in any particular order. For example, the lists `{0, 1, 2, 3, 4}` and `{4, 3, 2, 1, 0}` are semantically equivalent because checking whether a number is an element of the first list yields the same result as checking whether the number is in the second. In fact, any two lists containing the same elements are semantically equivalent to one another. This means that there are multiple ways in which we could implement `copyOther`. Consider these two:

```

/* Version 1: Duplicate the list as it exists in the original ListSet. */
template <typename T> void ListSet<T>::copyOther(const ListSet& other)
{
    /* Keep track of what the current linked list cell is. */
    cellT** current = &list;

    /* Iterate over the source list. */
    for(cellT* source = other.list; source != NULL; source = source->next)
    {
        /* Duplicate the cell. */
        *current = new cellT;
        (*current)->data = source->data;
        (*current)->next = NULL;

        /* Advance to next element. */
        current = &((*current)->next);
    }
}

/* Version 2: Duplicate list in reverse order of original ListSet */
template <typename T> void ListSet<T>::copyOther(const ListSet& other)
{
    for(cellT* source = other.list; source != NULL; source = source->next)
    {
        cellT* newNode = new cellT;
        newNode->data = source->data;
        newNode->next = list;
        list = newNode;
    }
}

```

As you can see, the second version of this function is much, *much* cleaner than the first. There are no address-of operators floating around, so everything is expressed in terms of simpler pointer operations. But while the second version is cleaner than the first, it duplicates the list in reverse order. This may initially seem problematic but is actually perfectly safe. As the original object and the duplicate object contain the same elements in *some* order, they will be semantically equivalent, and from the class interface we would be unable to distinguish the original object and its copy.

There is one implementation of `copyOther` that is considerably more elegant than either of the two versions listed above:

```

/* Version 3: Duplicate list using the insert function */
template <typename T> void ListSet<T>::copyOther(const ListSet& other)
{
    for(cellT* source = other.list; source != NULL; source = source->next)
        insert(source->data);
}

```

Notice that this implementation uses the `ListSet`'s public interface to insert the elements from the source `ListSet` into the receiver object. This version of `copyOther` is unquestionably the cleanest. If you'll notice, it doesn't matter exactly how `insert` adds elements into the list (indeed, `insert` could insert the elements at random positions), but we're guaranteed that at the end of the `copyOther` call, the receiver object will be semantically equivalent to the parameter.

Conversion Assignment Operators

When working with copy constructors, we needed to define an additional function, the assignment operator, to handle all the cases in which an object can be copied or assigned. However, in the chapter on conversion constructors, we provided a conversion constructor without a matching “conversion assignment operator.” It turns out that this is not a problem because of how the assignment operator is invoked. Suppose that we have a `CString` class that has a defined copy constructor, assignment operator, and conversion constructor that converts raw C++ `char *` pointers into `CString` objects. Now, suppose we write the following code:

```
CString myCString;  
myCString = "This is a C string!";
```

Here, in the second line, we assign an existing `CString` a new value equal to a raw C string. Despite the fact that we haven't defined a special assignment operator to handle this case, the above is perfectly legal code. When we write the line

```
myCString = "This is a C string!";
```

C++ converts it into the equivalent code

```
myCString.operator =("This is a C string!");
```

This syntax may look entirely foreign, but is simply a direct call to the assignment operator. Recall that the assignment operator is a function named `operator =`, so this code passes the C string `"This is a C string!"` as a parameter to `operator =`. Because `operator =` accepts a `CString` object rather than a raw C string, C++ will invoke the `CString` conversion constructor to initialize the parameter to `operator =`. Thus this code is equivalent to

```
myCString.operator =(CString("This is a C string!"));
```

In other words, the conversion constructor converts the raw C string into a `CString` object, then the assignment operator sets the receiver object equal to this temporary `CString`.

In general, you need not provide a “conversion assignment operator” to pair with a conversion constructor. As long as you've provided well-defined copy behavior, C++ will link the conversion constructor and assignment operator together to perform the assignment.

Disabling Copying

In CS106B/X we provide you the `DISALLOW_COPYING` macro, which causes a compile error if you try to assign or copy objects of the specified type. `DISALLOW_COPYING`, however, is not a standard C++ feature. Without using the CS106B/X library, how can we replicate the functionality? We can't prevent object copying by simply not defining a copy constructor and assignment operator. All this will do is have C++ provide its own default version of these two functions, which is not at all what we want. To solve this problem, instead we'll provide an assignment operator and copy constructor, but declare them private so that class clients can't access them. For example:

```

class CannotBeCopied
{
public:
    CannotBeCopied();
    /* Other member functions. */
private:
    CannotBeCopied(const CannotBeCopied& other);
    CannotBeCopied& operator = (const CannotBeCopied& other);
};

```

Now, if we write code like this:

```

CannotBeCopied one;
CannotBeCopied two = one;

```

We'll get a compile-time error on the second line because we're trying to invoke the copy constructor, which has been declared private. We'll get similar behavior when trying to use the assignment operator.

This trick is almost one hundred percent correct, but does have one edge case: what if we try to invoke the copy constructor or assignment operator inside a member function of the class? The copy functions might be private, but that doesn't mean that they don't exist, and if we call them inside a member function might accidentally create a copy of an otherwise uncopyable object. To prevent this from happening, we'll use a cute trick. Although we'll *prototype* the copy functions inside the private section of the class, we won't *implement* them. This means that if we accidentally do manage to call either function, we will get a linker error because the compiler can't find code for either function. This is admittedly a bit hackish, so in C++0x, the next revision of C++, there will be a way to explicitly indicate that a class is uncopyable. In the meantime, though, the above approach is perhaps your best option. We'll see another way to do this later when we cover inheritance.

Practice Problems

The only way to learn copy constructors and assignment operators is to play around with them to gain experience. Here are some practice problems and thought questions to get you started:

1. Realizing that the copy constructor and assignment operator for most classes have several commonalities, you decide to implement a class's copy constructor using the class's assignment operator. For example, you try implementing the `DebugVector`'s copy constructor as

```

template <typename T> DebugVector<T>::DebugVector(const DebugVector& other)
{
    *this = other;
}

```

(Since `this` is a pointer to the receiver object, `*this` is the receiver object, so `*this = other` means to assign the receiver object the value of the parameter `other`)

This idea, while well-intentioned, has a serious flaw that causes the copy constructor to almost always cause a crash. Why is this? (*Hint: Were any of the `DebugVector` data members initialized before calling the assignment operator? Walk through the assignment operator and see what happens if the receiver object's data members haven't been initialized.*) ♦

2. It is illegal to write a copy constructor that accepts its parameter by value. Why is this? However, it's perfectly acceptable to have an assignment operator that accepts its parameter by value. Why is this legal? Why the difference? ♦
3. In the previous chapter, we saw the class `CString` used as a case study for conversion constructors. `CString` is simply a class that wraps a C string, storing as data members the `char *` pointer to the C string. Write a copy constructor and assignment operator for `CString`.

4. Suppose you're implementing the `Lexicon` class and, for efficiency reasons, you decide to store the words in a sorted `vector` of dynamically-allocated C strings (`vector<char *>`). Assume that the constructor has been provided to you and that it correctly initializes the `vector` by filling it with strings dynamically allocated from `new[]`. Write the copy constructor, assignment operator, and destructor for `Lexicon`. (*Hint: When the vector destructor invokes, it will **not** call `delete []` on all of the internally stored `char *`s, so you will have to do this yourself*)
5. If the above `Lexicon` used a `vector<string>` instead of a `vector<char *>`, would you need any of the functions mentioned in the rule of three?
6. An alternative implementation of the assignment operator uses a technique called *copy-and-swap*. The copy-and-swap approach is broken down into two steps. First, we write a member function that accepts a reference to another instance of the class, then exchanges the data members of the receiver object and the parameter. For example, when working with the `DebugVector`, we might write a function called `swapWith` as follows:

```
template <typename ElemType>
void DebugVector<ElemType>::swapWith(DebugVector& other)
{
    swap(array, other.array);
    swap(logicalLength, other.logicalLength);
    swap(allocatedLength, other.allocatedLength);
}
```

Here, we use the STL `swap` algorithm to exchange data members. Notice that we never actually make a deep-copy of any of the elements in the array – we simply swap pointers with the other `DebugVector`. We can then implement the assignment operator as follows:

```
template <typename T>
DebugVector<T>& DebugVector<T>::operator= (const DebugVector& other)
{
    DebugVector temp(other);
    swapWith(temp);
    return *this;
}
```

Trace through this implementation of the assignment operator and explain how it sets the receiver object to be a deep-copy of the parameter. What function actually deep-copies the data? What function is responsible for cleaning up the old data members?

7. When writing an assignment operator using the pattern covered earlier in the chapter, we had to explicitly check for self-assignment in the body of the assignment operator. Explain why this is no longer necessary using the copy-and-swap approach, but why it still might be a good idea to insert the self-assignment check anyway.

8. A *singleton class* is a class that can have at most one instance. Typically, a singleton class has its default constructor and destructor marked private so that clients cannot instantiate the class directly, and exports a static member function called `getInstance()` that returns a reference to the only instance of the class. That one instance is typically a private static data member of the class. For example:

```
class Singleton
{
public:
    static Singleton& getInstance();

private:
    Singleton(); // Clients cannot call this function because it's private
    ~Singleton(); // ... or this one

    static Singleton instance; // ... but they can be used here because
                                // instance is part of the class.
};

Singleton Singleton::instance;
```

Does it make sense for a singleton class to have a copy constructor or assignment operator? If so, implement them. If not, modify the `Singleton` interface so that they are disabled.

9. Given this chapter's description about how to disable copying in a class, implement a macro `DISALLOW_COPYING` that accepts as a parameter the name of the current class such that if `DISALLOW_COPYING` is placed into the private section of a class, that class is uncopyable. Note that it is legal to create macros that span multiple lines by ending each line with the `\` character. For example, the following is all one macro:

```
#define CREATE_PRINTER(str) void Print##str()\
{\
    cout << #str << endl;\
}
```

10. Consider the following alternative mechanism for disabling copying in a class: instead of marking those functions private, instead we implement those functions, but have them call `abort` (the function from `<cstdlib>` that immediately terminates the program) after printing out an error message. For example:

```
class PseudoUncopyable
{
public:
    PseudoUncopyable(const PseudoUncopyable& other)
    {
        abort();
    }
    PseudoUncopyable& operator= (const PseudoUncopyable& other)
    {
        abort();
        return *this; // Never reached; suppresses compiler warnings
    }
};
```

Why is this approach a bad idea?

11. Should you copy static data members in a copy constructor or assignment operator? Why or why not?

Chapter 21: Extended Example: Critiquing Class Design

Over the past few chapters we've introduced many important language concepts – `const`, templates, initializer lists, `static`, conversion constructors, and copy functions to name a few. Each of these chapters had their own set of rules and caveats, and it feels overwhelming trying to commit each to memory. These are skills that can only come with practice, and hopefully this extended example can help exercise your newfound C++ knowledge.

The other extended examples in this course reader demonstrate how to solve problems elegantly and effectively in C++. This example, however, is quite the reverse. We'll begin with a broken solution to a problem, then explore how we can transform it into a working solution by applying the techniques of the past chapters.

Suppose that we want to implement a class encapsulating a forward-linked list. Each element is stored in a cell that contains a pointer to the next cell in the list, and the final element in the list points to `NULL`. Because the list is singly-linked, the overhead for each item is as small as possible and we can rapidly traverse over the entire container. However, because the list is singly-linked, insertions and deletions at arbitrary points are expensive, so we will only concern ourselves with addition and deletion at the front of the list. In particular, our singly-linked list will support the following operations:

- Addition and deletion of elements at the beginning of the list.
- Iteration over the entire list.
- Size and empty queries.

There are several other functions we might want to add to this list, but for simplicity we'll only concern ourselves with this set. If you'd like an implementation challenge, feel free to design and implement some other operations.

For this extended example, we will design a class called `ForwardList` that supports these above operations. However, we will only concern ourselves with the *interface* for the `ForwardList` class rather than the *implementation*. This will let us exercise the techniques in the previous chapters more directly. This is not to say that implementing a forward-linked list class is unimportant, of course, and I encourage you to do so on your own.

Suppose that we need to implement this `ForwardList` class for a software project we're working on and that another student in the group, who has a rudimentary understanding of C++, volunteers to design and implement the class. After several hours, he returns with the following code:

```

/* A class representing a forward-linked list of strings. This code contains
 * several serious errors, so don't use it as a reference!
 */
class ForwardList
{
public:
    /**
     * Constructor: ForwardList(numElems, defaultValue)
     * Usage: ForwardList myList(10); // Contains ten copies of the empty string
     * Usage: ForwardList myList(10, "Hello!"); // Contains ten copies of "Hello!"
     * -----
     * Constructs a new list of the specified size where each element is
     * initialized to the specified value. If no size is specified, defaults to
     * zero. If no default value is specified, the empty string is used.
     */
    ForwardList(int numElems = 0, string defaultValue = "");

    /**
     * Constructor: ForwardList(vector<string>::iterator begin,
     *                          vector<string>::iterator end);
     * Usage: ForwardList myList(v.begin(), v.end());
     * -----
     * Constructs a new list whose elements are equal to those specified by
     * the input pair of vector<string>::iterators.
     */
    ForwardList(vector<string>::iterator begin, vector<string>::iterator end);

    /**
     * Destructor: ~ForwardList();
     * Usage: delete myListPtr;
     * -----
     * Cleans up any memory allocated by this ForwardList object.
     */
    ~ForwardList();

    /**
     * Type: iterator
     * -----
     * A forward iterator that can read and write values stored in the
     * ForwardList.
     */
    typedef something-implementation-specific iterator;

    /**
     * Functions begin() and end()
     * Usage: for(ForwardList::iterator itr = l.begin(); itr != l.end(); ++itr)
     * -----
     * Returns iterators to the first element in the list and the first element
     * past the end of the list, respectively.
     */
    iterator begin();
    iterator end();

```

```

/**
 * Functions: push_front(val), pop_front(), front()
 * Usage: v.push_front("This is a string!"); s = v.front(); v.pop_front();
 * -----
 * push_front prepends the specified element to the linked list,
 * increasing its size by one. pop_front removes the first element of the
 * list without returning its value; use front() before calling pop_front()
 * if you want to retrieve the value.
 *
 * front() returns a reference to the first element in the list, so it's
 * legal to write either val = front() or front() = val.
 */
void push_front(string value);
void pop_front();
string& front();

/**
 * Functions: empty(), size()
 * Usage: if(v.empty()) ... if(v.size() > 10) ...
 * -----
 * empty() returns whether the list is empty (i.e. begin() == end()).
 * size() returns the number of elements stored in the list.
 */
bool empty();
int size();

private:
    /* ... implementation details ... */
};

```

The class is certainly well-commented, but contains several errors in its interface. Before turning to the next page where we'll discuss exactly what's wrong, take a few minutes to look over the interface. Mark all the errors you find. Not all of the mistakes are misuses of language features; at least one of the functions defined here is fine as written but could be dramatically improved with the help of additional language features.

Ready? Great! Let's take a look at what we can fix about this class.

Problem 0: const-Correctness (or lack thereof)

The chapter on `const`-correctness should have instilled in you the fact that classes should be `const`-correct. Hopefully you noticed that this class is in dire need of `const`ifying. None of the member functions are marked `const`, and all the `string` function parameters are all accepted by value instead of `reference-to-const`. Let's see how we can fix this. For simplicity, here's the entire class interface with the commenting removed:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, string defaultValue = "");
    ForwardList(vector<string>::iterator begin, vector<string>::iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;

    iterator begin();
    iterator end();

    void push_front(string value);
    void pop_front();
    string& front();

    bool empty();
    int size();
};
```

Our first order of business is to modify the class so that functions take their parameters by `reference-to-const` instead of by value. This yields the following updated class:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::iterator begin, vector<string>::iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;

    iterator begin();
    iterator end();

    void push_front(const string& value);
    void pop_front();
    string& front();

    bool empty();
    int size();
};
```

Notice that the parameters to the second constructor are still passed by value instead of by `reference-to-const`. This is deliberate. Because iterators are almost always modified inside of functions that require them, it is acceptable to pass them by value. Iterators tend to be lightweight objects, so the cost of the pass-by-value is usually not a problem. However, these iterator parameters do present another `constness` problem. The parameters to this function are used to construct a new `ForwardList`, so none of the elements in the range they delimit should be modified. To explicitly indicate that we won't modify elements in the range, we'll accept the paramet-

ers as `vector<string>::const_iterator` instead of regular `vector<string>::iterator`. This leads to the following interface:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::const_iterator begin,
                vector<string>::const_iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;

    iterator begin();
    iterator end();

    void push_front(const string& value);
    void pop_front();
    string& front();

    bool empty();
    int size();
};
```

Next, let's mark non-mutating member functions `const`. `empty` and `size` clearly should not modify the `ForwardList`, so we'll mark them `const`, as shown here:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::const_iterator begin,
                vector<string>::const_iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;

    iterator begin();
    iterator end();

    void push_front(const string& value);
    void pop_front();
    string& front();

    bool empty() const;
    int size() const;
};
```

Let's now consider `push_front`, `pop_front`, and `front`. `push_front` and `pop_front` are clearly mutating because they change the size of the container, but what about `front`? The description of `front` given in the comments says that `front` should return a reference to the first element in the container, meaning that clients of `ForwardList` can modify the first element in the list by writing code similar to this:

```
myList.front() = "Modified!";
```

We therefore cannot mark `front` `const`, since this would subvert `const`ness. However, we should definitely consider a `const` overload for `front` that returns a `const` reference to the first element of the list. If we do not, clients of a `const ForwardList` could not read the front element without having to use iterators. As a general

rule, if you have a member function that returns a reference to an object, you should probably also provide a `const` overload for that function that returns a `const` reference. The updated `ForwardList` looks like this:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::const_iterator begin,
                vector<string>::const_iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;

    iterator begin();
    iterator end();

    void push_front(const string& value);
    void pop_front();
    string& front();
    const string& front() const;

    bool empty() const;
    int size() const;
};
```

All that's left now are the `begin` and `end` functions. As mentioned in the first chapter on `const`, these functions should be overloaded so that we can get `const_iterator`s when the list is marked `const`. This will require us to also provide a `const_iterator` type. The resulting interface is shown here:

```
class ForwardList
{
public:
    ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::const_iterator begin,
                vector<string>::const_iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const string& value);
    void pop_front();
    string& front();
    const string& front() const;

    bool empty() const;
    int size() const;
};
```

At this point our `ForwardList` class is `const`-corrected and we can move on to consider other issues with the class design.

Problem 1: Constructors Gone Bad

The `ForwardList` class has two constructors, one of which contains a serious error and the other of which is not designed particularly well. Let's take a look at how these constructors are prototyped and what we can do to make them better.

Let's take a look at the first `ForwardList` constructor. It has the following prototype:

```
ForwardList(int numElems = 0, const string& defaultValue = "");
```

Recall that C++ interprets any constructor that can be called with one argument as a conversion constructor. This constructor has two parameters, but because the second has a default value (the empty string) it's legal to construct a `ForwardList` by writing

```
ForwardList myList(137); // Constructs myList(137, "");
```

Consequently, C++ treats this as a conversion constructor and it's now legal to write

```
ForwardList myList = 137;
```

This is not intended, and to prevent this problem we need to indicate that this constructor is not an implicit conversion constructor. This is a job for the `explicit` keyword, resulting in the following interface:

```
class ForwardList
{
public:
    explicit ForwardList(int numElems = 0, const string& defaultValue = "");
    ForwardList(vector<string>::const_iterator begin,
                vector<string>::const_iterator end);
    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const string& value);
    void pop_front();
    string& front();
    const string& front() const;

    bool empty() const;
    int size() const;
};
```

Let's take a look at the other constructor, which is declared as

```
ForwardList(vector<string>::const_iterator begin,
            vector<string>::const_iterator end);
```

This constructor exists so that we can take an existing list of strings and convert it into a `ForwardList`. In particular, if we have a `vector<string>` that contains the strings we want, we can use its elements as initial values for our `ForwardList`. But why single out `vector<string>`? What if we wanted to construct a `For-`

wardList from a `deque<string>` or a `set<string>`? And as long as we're using iterators, why not allow iterators from another `ForwardList` or even a pair of `istream_iterator<string>`s? The idea of using iterators is a noble one, but the approach in this implementation is incorrect.

To fix this, we'll rewrite the constructor so that it accepts a pair of iterators of any type. As in the `UnionFind` example, we accomplish this by rewriting the constructor as a constructor *template* parameterized over the particular type of iterators used. This is shown here:

```
class ForwardList
{
public:
    explicit ForwardList(int numElems = 0, const string& defaultValue = "");
    template <typename InputIterator>
        ForwardList(InputIterator begin, InputIterator end);

    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const string& value);
    void pop_front();
    string& front();
    const string& front() const;

    bool empty() const;
    int size() const;
};
```

Notice that we've named the type of the template argument `InputIterator` to indicate that this function should take in an iterator that's at least as powerful as an `InputIterator`. After all, to construct a forward-linked list, we don't need to be able to write to the iterator range or jump around randomly.

Problem 2: Copy Behavior

The `ForwardList` class declared above doesn't have a declared copy constructor or assignment operator, so it relies on C++'s automatically-generated copy functions. The class encapsulates a forward-linked list, which usually requires explicit memory management. Because of this, relying on the automatically-generated implementation of the copy functions might be dangerous. But then again, we're just given the *interface* of `ForwardList`, not the *implementation*, so it's possible that the class is implemented so that the default copy behavior might be correct. Is there a way that we can check if the interface has a problem without looking at the implementation? Absolutely, because the class has a destructor. Remember that the Rule of Three states that a class with a destructor should also have a copy constructor and assignment operator, and the lone destructor in this class definition suggests that the class contains an error. We'll thus update the class to add the copy functions, as shown here:

```

class ForwardList
{
public:
    explicit ForwardList(int numElems = 0, const string& defaultValue = "");
    template <typename InputIterator>
        ForwardList(InputIterator begin, InputIterator end);

    ForwardList(const ForwardList& other);
    ForwardList& operator= (const ForwardList& other);

    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const string& value);
    void pop_front();
    string& front();
    const string& front() const;

    bool empty() const;
    int size() const;
};

```

Problem 3: Overspecialization

The `ForwardList` class defined above is considerably better than the initial version. It's `const` correct, sports a generic constructor to make a `ForwardList` out of any range of strings, and has support for deep copying. But there's one more problem with the class. As written, the class is hardcoded to only work with strings. Why is this the case? There's nothing in this interface that requires the list to store strings, and we could use this exact interface with `string` substituted for `int` to have a `ForwardList` that works for ints. As a last step, we'll templatize the `ForwardList` class over the type of element to store in it. This is shown here:

```

template <typename ElemType> class ForwardList
{
public:
    explicit ForwardList(int numElems = 0,
                        const ElemType& defaultValue = /* see below */);
    template <typename InputIterator>
        ForwardList(InputIterator begin, InputIterator end);

    ForwardList(const ForwardList& other);
    ForwardList& operator= (const ForwardList& other);

    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const ElemType& value);
    void pop_front();
    ElemType& front();
    const ElemType& front() const;

    bool empty() const;
    int size() const;
};

```

We're almost done and all that's left is to specify the default value in the first constructor. In the case of the string version of the `ForwardList` we could hardcode that the default value should be the empty string, but what about the more general template case? That is, what should the default value be for elements in this list?

One elegant solution requires a new piece of syntax called the *temporary object syntax*. In C++, it is legal to explicitly call a class's constructor to create a temporary instance of that class constructed with the specified arguments. You have already seen this with `istream_iterator` and `ostream_iterator`. For example, in the code `copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"))`, the third parameter is a temporary `ostream_iterator<int>` constructed with parameters `cout` and `"\n"`. `ostream_iterator` is a *type*, not a *function*, so the syntax `ostream_iterator<int>(cout, "\n")` is a constructor call rather than a function call.

We can also use this syntax to construct temporary primitive values. For example, `int()` constructs a temporary integer, and `double()` constructs a temporary double. Interestingly, if you construct a temporary primitive type this way, the primitive will be initialized to zero. That is, `int()` always produces an integer of value zero, rather than an integer with a garbage value.

The temporary object syntax is curious, but how is it relevant to our above discussion? Simple – we'll specify that the default value for the final parameter to the constructor is a temporary instance of an object of type `ElemType`. For example:

```

template <typename ElemType> class ForwardList
{
public:
    explicit ForwardList(int numElems = 0,
                        const ElemType& defaultValue = ElemType());
    template <typename InputIterator>
        ForwardList(InputIterator begin, InputIterator end);

    ForwardList(const ForwardList& other);
    ForwardList& operator= (const ForwardList& other);

    ~ForwardList();

    typedef something-implementation-specific iterator;
    typedef something-implementation-specific const_iterator;

    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;

    void push_front(const ElemType& value);
    void pop_front();
    ElemType& front();
    const ElemType& front() const;

    bool empty() const;
    int size() const;
};

```

Provided that the type being stored in the `ForwardList` has a zero-parameter constructor, this code will work correctly.

Summary

This extended example focused exclusively on `ForwardList` as an example, but the techniques we used were completely general. Make sure that you understand the sorts of thought processes that went into transforming `ForwardList` from its initial state into its sleek final version. If you take the time to think through class design, you will end up with code that is more robust and less error-prone.

An interesting observation is that we didn't need to look at the implementation of the `ForwardList` class at any point during our analysis. This is no coincidence. The decisions we made here to remedy a broken class can also be used during class design to construct a class interface before even deciding on a particular implementation. In fact, this is how you should design your classes. Think about what operations you will need to support, and whether or not they should be `const`. If your class can be parameterized over an arbitrary type, make it a template. Only after you've thought this through should you actually sit down to implement the necessary features.

Chapter 22: Operator Overloading

Consider the following C++ code snippet:

```
vector<string> myVector(NUM_STRINGS);
for(vector<string>::iterator itr = myVector.begin(); itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

Here, we create a `vector<string>` of a certain size, then iterate over it concatenating “Now longer!” to each of the strings. Code like this is ubiquitous in C++, and initially does not appear all that exciting. However, let's take a closer look at how this code is structured. First, let's look at exactly what operations we're performing on the iterator:

```
vector<string> myVector(NUM_STRINGS);
for(vector<string>::iterator itr = myVector.begin(); itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

In this simple piece of code, we're comparing the iterator against `myVector.end()` using the `!=` operator, incrementing the iterator with the `++` operator, and dereferencing the iterator with the `*` operator. At a high level, this doesn't seem all that out of the ordinary since STL iterators are designed to look like regular pointers and these operators are all well-defined on pointers. But the key thing to notice is that STL iterators *aren't* pointers, they're *objects*, and `!=`, `*`, and `++` aren't normally defined on objects. We can't write code like `++myVector` or `*myMap = 137`, so why can these operations be applied to `vector<string>::iterator`?

Similarly, notice how we're concatenating the string “Now longer!” onto the end of the string:

```
vector<string> myVector(NUM_STRINGS);
for(vector<string>::iterator itr = myVector.begin(); itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

Despite the fact that `string` is an object, somehow C++ “knows” what it means to apply `+=` to strings.

All of the above examples are instances of *operator overloading*, the ability to specify how operators normally applicable to primitive types can interact with custom classes. Operator overloading is ubiquitous in professional C++ code and, used correctly, can make your programs more concise, more readable, and more template-friendly.

There are two overarching purposes of operator overloading. First, operator overloading enables your custom classes to act like primitive types. That is, if you have a class like `vector` that mimics a standard C++ array, you can allow clients to use array notation to access individual elements. Similarly, when designing a class encapsulating a mathematical entity (for example, a complex number), you can let clients apply mathematical operators like `+`, `-`, and `*` to your type as though it were built into the language. Second, operator overloading enables your code to interact correctly with template and library code. For example, you can overload the `<<` operator to make a class compatible with the streams library, or the `<` operator to interface with STL containers.

This chapter discusses general topics in operator overloading, demonstrating how to overload some of the more common operators. It also includes tricks and pitfalls to be aware of when overloading certain operators. However, we will not focus extensively on how to *implement* most of the operators outlined in this chapter, and will instead defer this discussion for the next three chapters, each of which showcases one particular aspect of operator overloading.

A Word of Warning

I would be remiss to discuss operator overloading without first prefacing it with a warning: operator overloading is a double-edged sword. When used correctly, operator overloading can lead to intuitive, template-friendly code that elegantly performs complex operations behind the scenes. However, incorrectly overloaded operators can lead to incredibly subtle bugs – just think of the problems associated with the assignment operator. Now that we're delving deeper into operator overloading, you'll encounter more potential for these sorts of mistakes.

There is a pearl of design wisdom that is particularly applicable to operator overloading:

Theorem (*The Principle of Least Astonishment*): A function's name should communicate its behavior and should be consistent with other naming conventions and idioms.

The principle of least astonishment should be fairly obvious – you should design functions so that clients can understand what those functions do simply by looking at the functions' names; that is, clients of your code should not be “astonished” that a function with one name does something entirely different. For example, a function named `DoSomething` violates the principle of least astonishment because it doesn't communicate what it does, and a function called `ComputePrimes` that reads a grocery list from a file violates the principle because the name of the function is completely different from the operation it performs. However, other violations of the principle of least astonishment are not as blatant. For example, a custom container class whose `empty` member function erases the contents of the container violates the principle of least astonishment because C++ programmers expect `empty` to mean “is the container empty?” rather than “empty the container.” Similarly, a class with a copy constructor but no assignment operator violates the principle of least astonishment because programmers expect those functions to come in pairs.

When working with operator overloading, it is crucial to adhere to the principle of least astonishment. C++ lets you redefine almost all of the built-in operators however you choose, meaning that it's possible to create code that does something completely different from what C++ programmers might expect. For example, suppose that you are working on a group project and that one of your teammates writes a class `CustomVector` that acts like the STL `vector` but which performs some additional operations behind the scenes. Your program contains a small bug, so you look over your teammate's code and find the following code at one point:

```
CustomVector one = /* ... */, two = /* ... */;
one %= two;
```

What does the line `one %= two` do? Syntactically, this says “take the remainder when dividing `one` by `two`, then store the result back in `one`.” But this makes no sense – how can you divide one `CustomVector` by another? You ask your teammate about this, and he informs you that the `%=` operator means “remove all elements from `one` that are also in `two`.” This is an egregious violation of the principle of least astonishment. The code neither communicates what it does nor adheres to existing convention, since the semantics of the `%=` operator are meaningless when applied to linear data structures. This is not to say, of course, that having the ability to remove all elements from one `CustomVector` that are contained in another is a bad idea – in fact, it can be quite useful – but this functionality should be provided by a properly-named member function rather than a cryptically-overloaded operator. For example, consider the following code:

```
CustomVector one = /* ... */, two = /* ... */;
one.removeAllIn(two);
```

This code accomplishes the same task as the above code, but does so by explicitly indicating what operation is being performed. This code is much less likely to confuse readers and is far more descriptive than before.

As another example, suppose that your teammate also implements a class called `CustomString` that acts like the standard `string` type, but provides additional functionality. You write the following code:

```
CustomString one = "Hello", two = " World";  
one += two;  
cout << one + "!" << endl;
```

Intuitively, this should create two strings, then concatenate the second onto the end of the first. Next, the code prints out `one` followed by an exclamation point, yielding “Hello World!” Unfortunately, when you compile this code, you get an unusual error message – for some reason the code `one += two` compiles correctly, but the compiler rejects the code `one + "!"`. In other words, your teammate has made it legal to concatenate two strings using `+=`, but not by using `+`. Again, think back to the principle of least astonishment. Programmers tacitly expect that objects that can be added with `+` can be added with `+=` and vice-versa, and providing only half of this functionality is likely to confuse code clients.

The moral of this story is simple: when overloading operators, make sure that you adhere to existing conventions. If you don't, you will end up with code that compiles and is either incorrect or confusing.

Hopefully this grim introduction has not discouraged you from using operator overloading. Operator overloading is extraordinarily useful and you will not be disappointed with the possibilities that are about to open up to you. With that said, let's begin discussing the mechanics behind this powerful technique.

General Operator Overloading Principles

To define an overloaded operator, declare a function whose name is `operator op`, where `op` represents whatever operator you're overloading. Thus the overloaded assignment operator is `operator =`, while the overloaded `<< operator` is `operator <<`. Whenever you use a built-in operator on instances of a class, C++ will replace the use of that operator with a call to the proper `operator op` function. For example, the following code:

```
string one, two, three;  
one = two + three;
```

is equivalent to

```
string one, two, three;  
one.operator= (operator+ (two, three));
```

The second version, while syntactically legal, is extremely rare in practice since the point of operator overloading is to make the syntax more intuitive. Also note that `operator=` is a member function of `string` and that `operator+` is a free function. With a few exceptions, overloaded operators can be defined either as class members or free functions, and we will explore both cases in this chapter.

Notice that operator overloading is simply *syntax sugar*, a way of rewriting one operation (in this case, function calls) using a different syntax. Overloaded operators are not somehow “more efficient” than other functions simply because the function calls aren't explicit, nor are they treated any different from regular functions.

When overloading operators, you cannot define brand-new operators like `#` or `@`. After all, C++ wouldn't know the associativity or proper syntax for the operator (is `one # two + three` interpreted as `(one # two) + three` or `one # (two + three)`?) Additionally, you cannot overload any of the following operators, because they are needed at compile-time:

<code>::</code>	<code>MyClass::value</code>	Scope resolution
<code>.</code>	<code>one.value</code>	Member selection
<code>?:</code>	<code>a > b ? -1 : 1</code>	Ternary conditional
<code>.*</code>	<code>a.*myClassPtr;</code>	Pointer-to-member selection (beyond the scope of this text)
<code>sizeof</code>	<code>sizeof(MyClass)</code>	Size of object
<code>typeid</code>	<code>typeid(MyClass)</code>	Runtime type information operator (beyond the scope of this text)

You also cannot overload any of C++'s typecasting operators (e.g. `static_cast`).

The number of parameters to an overloaded operator depends on the particular operator being overloaded. As you saw in the chapter on copy constructors and assignment operators, the assignment operator (`operator =`) takes a single parameter representing the value being assigned. Other operators like `operator +` usually have two parameters, one for each operand, while some operators like the pointer dereference operator `operator *` take none.

Note that operator overloading only lets you define what built-in operators mean when applied to *objects*. You cannot use operator overloading to redefine what addition means as applied to `ints`, nor can you change how pointers are dereferenced. Then again, by the principle of least astonishment, you wouldn't want to do this anyway.

Lvalues and Rvalues

Before discussing the specifics of operator overloading, we need to take a quick detour to explore two concepts from programming language theory called *lvalues* and *rvalues*. Lvalues and rvalues stand for “left values” and “right values” are so-named because of where they can appear in an assignment statement. In particular, an lvalue is a value that can be on the left-hand side of an assignment, and an rvalue is a value that can only be on the right-hand side of an assignment. For example, in the statement `x = 5`, `x` is an lvalue and `5` is an rvalue. Similarly, in `*itr = 137`, `*itr` is the lvalue and `137` is the rvalue.

It is illegal to put an rvalue on the left-hand side of an assignment statement. For example, `137 = 42` is illegal because `137` is an rvalue, and `GetInteger() = x` is illegal because the return value of `GetInteger()` is an rvalue. However, it *is* legal to put an lvalue on the right-hand side of an assignment, as in `x = y` or `x = *itr`.

At this point the distinction between lvalues and rvalues seems purely academic. “Okay,” you might say, “some things can be put on the left-hand side of an assignment statement and some things can't. So what?” When writing overloaded operators, the lvalue/rvalue distinction is extremely important. Because operator overloading lets us define what the built-in operators mean when applied to objects of class type, we have to be very careful that overloaded operators return lvalues and rvalues appropriately. For example, by default the `+` operator returns an rvalue; that is, it makes no sense to write

```
(x + y) = 5;
```

Since this would assign the value `5` to the result of adding `x` and `y`. However, if we're not careful when overloading the `+` operator, we might accidentally make the above statement legal and pave the way for nonsensical but legal code. Similarly, it *is* legal to write

```
myArray[5] = 137;
```

So the element-selection operator (the brackets operator) should be sure to return an lvalue instead of an rvalue. Failure to do so will make the above code illegal when applied to custom classes.

Recall that an overloaded operator is a specially-named function invoked when the operator is applied to an object of a custom class type. Thus the code

```
(x + y) = 5;
```

is equivalent to

```
operator+ (x, y) = 5;
```

if either `x` or `y` is an object. Similarly, if `myArray` is an object, the code

```
myArray[5] = 137;
```

is equivalent to

```
myArray.operator[] (5) = 137;
```

To ensure that these functions have the correct semantics, we need to make sure that `operator+` returns an rvalue and that `operator[]` returns an lvalue. How can we enforce this restriction? The answer has to do with the return type of the two functions. To make a function that returns an lvalue, have that function return a non-`const` reference. For example, the following function returns an lvalue:

```
string& LValueFunction();
```

Because a reference is just another name for a variable or memory location, this function hands back a reference to an lvalue and its return value can be treated as such. To have a function return an rvalue, have that function return a `const` object by value. Thus the function

```
const string RValueFunction();
```

returns an rvalue.* The reason that this trick works is that if we have a function that returns a `const` object, then code like

```
RValueFunction() = 137;
```

is illegal because the return value of `RValueFunction` is marked `const`.

Lvalues and rvalues are difficult to understand in the abstract, but as we begin to actually overload particular operators the difference should become clearer.

Overloading the Element Selection Operator

Let's begin our descent into the realm of operator overloading by discussing the overloaded element selection operator (the `[]` operator, used to select elements from arrays). You've been using the overloaded element selection operator ever since you encountered the `string` and `vector` classes. For example, the following code uses the `vector`'s overloaded element selection operator:

```
for(int i = 0; i < myVector.size(); ++i)
    myVector[i] = 137;
```

* Technically speaking any non-reference value returned from a function is an rvalue. However, when returning objects from a function, the rvalue/lvalue distinction is blurred because the assignment operator and other operators are member functions that can be invoked regardless of whether the receiver is an rvalue or lvalue. The additional `const` closes this loophole.

In the above example, while it looks like we're treating the `vector` as a primitive array, we are instead calling the a function named `operator []`, passing `i` as a parameter. Thus the above code is equivalent to

```
for(int i = 0; i < myVector.size(); ++i)
    myVector.operator [] (i) = 137;
```

To write a custom element selection operator, you write a member function called `operator []` that accepts as its parameter the value that goes inside the brackets. Note that while this parameter can be of any type (think of the STL `map`), you can only have a single value inside the brackets. This may seem like an arbitrary restriction, but makes sense in the context of the principle of least astonishment: you can't put multiple values inside the brackets when working with raw C++ arrays, so you shouldn't do so when working with custom objects.

When writing `operator []`, as with all overloaded operators, you're free to return objects of whatever type you'd like. However, remember that when overloading operators, it's essential to maintain the same functionality you'd expect from the naturally-occurring uses of the operator. In the case of the element selection operator, this means that the return value should be an lvalue, and in particular a reference to some internal class data determined by the index. For example, here's one possible prototype of the C++ `string`'s element selection operator:

```
class string
{
public:
    /* ... */
    char& operator [] (int position);
};
```

Here, `operator[]` takes in an `int` representing the index and returns a reference to the character at that position in the string. If `string` is implemented as a wrapper for a raw C string, then one possible implementation for `operator[]` might be

```
char& string::operator[] (int index)
{
    return theString[index]; // Assuming theString is a raw C string
}
```

Because `operator[]` returns a reference to an element, it is common to find `operator[]` paired with a `const`-overload that returns a `const` reference to an element in the case where the receiver object is immutable. There are exceptions to this rule, such as the STL `map`, but in the case of `string` we should provide a `const` overload, as shown here:

```
class string
{
public:
    /* ... */
    char& operator [] (int position);
    const char& operator [] (int position) const;
};
```

The implementation of the `const operator[]` function is identical to the non-`const` version.

When writing the element selection operator, it's completely legal to modify the receiver object in response to a request. For example, with the STL `map`, `operator[]` will silently create a new object and return a reference to it if the key isn't already in the map. This is part of the beauty of overloaded operators – you're allowed to perform any necessary steps to ensure that the operator makes sense.

Unfortunately, if your class encapsulates a multidimensional object, such as a matrix or hierarchical key-value system, you cannot “overload the `[] []` operator.” A class is only allowed to overload one level of the bracket syntax; it's not legal to design objects that doubly-overload `[]`.*

Overloading Compound Assignment Operators

The compound assignment operators are operators of the form `op=` (for example, `+=` and `*=`) that update an object's value but do not overwrite it. Compound assignment operators are often declared as member functions with the following basic prototype:

```
MyClass& operator += (const ParameterType& param)
```

For example, suppose we have the following class, which represents a vector in three-dimensional space:

```
class Vector3D
{
public:
    /* ... */
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

It is legal to add two mathematical vectors to one another; the result is the vector whose components are the pairwise sum of each of the components of the source vectors. If we wanted to define a `+=` operator for `Vector3D` to let us perform this addition, we would modify the interface of `Vector3D` as follows:

```
class Vector3D
{
public:
    /* ... */
    Vector3D& operator+= (const Vector3D& other);
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

This could then be implemented as

```
Vector3D& Vector3D::operator +=(const Vector3D& other)
{
    for(int i = 0; i < NUM_COORDINATES; ++i)
        coordinates[i] += other.coordinates[i];
    return *this;
}
```

If you'll notice, `operator+=` returns `*this`, a reference to the receiver object. Recall that when overloading operators, you should make sure to define your operators such that they work identically to the C++ built-in operators. It turns out that the `+=` operator yields an lvalue, so the code below, though the quintessence of abysmal style, is perfectly legal:

```
int one, two, three, four;
(one += two) += (three += four);
```

* There is a technique called *proxy objects* that can make code along the lines of `myObject[x][y]` legal. The trick is to define an `operator[]` function for the class that returns another object that itself overloads `operator[]`. We'll see this trick used in the upcoming chapter on a custom `grid` class.

Since overloaded operators let custom types act like primitives, the following code should also compile:

```
Vector3D one, two, three, four;
(one += two) += (three += four);
```

If we expand out the calls to the overloaded += operator, we find that this is equivalent to

```
Vector3D one, two, three, four;
one.operator+=(two).operator +=(three.operator +=(four)) ;
```

Note that the reference returned by `one.operator+=(two)` then has its own += operator invoked. Since `operator +=` is not marked `const`, had the += operator returned a `const` reference, this code would have been illegal. Make sure to have any (compound) assignment operator return `*this` as a non-`const` reference.

Unlike the regular assignment operator, with the compound assignment operator it's commonly meaningful to accept objects of different types as parameters. For example, we might want to make expressions like `myVector *= 137` for `Vector3Ds` meaningful as a scaling operation. In this case, we can simply define an operator `*` that accepts a double as its parameter. For example:

```
class Vector3D
{
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator *= (double scaleFactor);
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

Despite the fact that the receiver and parameter have different types, this is perfectly legal C++. Here's one possible implementation:

```
Vector3D& Vector3D::operator*= (double scaleFactor)
{
    for(int k = 0; k < NUM_COORDINATES; ++k)
        coordinates[k] *= scaleFactor;
    return *this;
}
```

Although we have implemented `operator+=` and `operator*=` for the `Vector3D` class, C++ will not automatically provide us an implementation of `operator-=` and `operator/=`, despite the fact that those functions can easily be implemented as wrapped calls to the operators we've already implemented. This might seem counterintuitive, but prevents errors from cases where seemingly symmetric operations are undefined. For example, it is legal to multiply a vector and a matrix, though the division is undefined. For completeness' sake, we'll prototype `operator-=` and `operator/=` as shown here:


```

class Vector3D
{
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator -= (const Vector3D& other);

    Vector3D& operator *= (double scaleFactor);
    Vector3D& operator /= (double scaleFactor);
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};

```

Now, how might we go about implementing these operators? `operator/=` is the simplest of the two and can be implemented as follows:

```

Vector3D& Vector3D::operator /= (double scaleFactor)
{
    *this *= 1.0 / scaleFactor;
    return *this;
}

```

This implementation, though cryptic, is actually quite elegant. The first line, `*this *= 1.0 / scaleFactor`, says that we should multiply the receiver object (`*this`) by the reciprocal of `scaleFactor`. The `*=` operator is the compound multiplication assignment operator that we wrote above, so this code invokes `operator*=` on the receiver. In fact, this code is equivalent to

```

Vector3D& Vector3D::operator /= (double scaleFactor)
{
    operator*= (1.0 / scaleFactor);
    return *this;
}

```

Depending on your taste, you might find this particular syntax more readable than the first version. Feel free to use either version.

Now, how would we implement `operator-=`, which performs a componentwise subtraction of two `Vector3Ds`? At a high level, subtracting one vector from another is equal to adding the inverse of the second vector to the first, so we might want to write code like this:

```

Vector3D& Vector3D::operator -= (const Vector3D& other)
{
    *this += -other;
    return *this;
}

```

That is, we add `-other` to the receiver object. But this code is illegal because we haven't defined the unary minus operator as applied to `Vector3Ds`. Not to worry – we can overload this operator as well. The syntax for this function is as follows:

```

class Vector3D
{
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator -= (const Vector3D& other);

    Vector3D& operator *= (double scaleFactor);
    Vector3D& operator /= (double scaleFactor);

    const Vector3D operator- () const;
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};

```

There are four pieces of information about this function that deserve attention:

- The name of the unary minus function is `operator -`.
- The function takes no parameters. This lets C++ know that the function is the *unary* minus function (I.e. `-myVector`) rather than the *binary* minus function (`myVector - myOtherVector`).
- The function returns a `const Vector3D`. The unary minus function returns an *rvalue* rather than an *lvalue*, since code like `-x = 137` is illegal. As mentioned above, this means that the return value of this function should be a `const Vector3D`.
- The function is marked `const`. Applying the unary minus to an object doesn't change its value, and to enforce this restriction we'll mark `operator - const`.

One possible implementation of `operator-` is as follows:

```

const Vector3D Vector3D::operator- () const
{
    Vector3D result;
    for(int k = 0; k < NUM_COORDINATES; ++k)
        result.coordinates[k] = -coordinates[k];
    return result;
}

```

Note that the return type of this function is `const Vector3D` while the type of `result` inside the function is `Vector3D`. This isn't a problem, since returning an object from a function yields a new temporary object and it's legal to initialize a `const Vector3D` using a `Vector3D`.

When writing compound assignment operators, as when writing regular assignment operators, you must be careful that self-assignment works correctly. In the above example with `Vector3D`'s compound assignment operators we didn't need to worry about this because the code was structured correctly. However, when working with the C++ `string`'s `+=` operator, since the `string` needs to allocate a new buffer capable of holding the current `string` appended to itself, it would need to handle the self-assignment case, either by explicitly checking for self-assignment or through some other means.

Overloading Mathematical Operators

In the previous section, we provided overloaded versions of the `+=` family of operators. Thus, we can now write classes for which expressions of the form `one += two` are valid. However, the seemingly equivalent expression `one = one + two` will still not compile, since we haven't provided an implementation of the lone `+` oper-

ator. C++ will not automatically provide implementations of related operators given a single overloaded operator, since in some cases this could result in nonsensical or meaningless behavior.

The built-in mathematical operators yield rvalues, so code like `(x + y) = 137` will not compile. Consequently, when overloading the mathematical operators, make sure they return rvalues as well by having them return `const` objects.

Let's consider an implementation of operator `+` for a `CString` type that encapsulates a dynamically-allocated C-style string. `CString` has the following definition:

```
class CString
{
public:
    /* ... constructor, destructor, etc ... */

    CString& operator += (const CString& other);
private:
    char* theString;
};
```

We want to write an implementation of operator `+`. Because the operator yields an rvalue, we're supposed to return a `const CString`, and based on our knowledge of parameter passing, we know that we should accept a `const CString &` as a parameter. There's one more bit we're forgetting, though, and that's to mark the operator `+` function `const`, since operator `+` creates a new object and doesn't modify either of the values used in the arithmetic statement. This results in the following code:

```
class CString
{
public:
    /* ... constructor, destructor, etc ... */

    CString& operator += (const CString& other);
    const CString operator+ (const CString& other) const;
private:
    char* theString;
};
```

However, we might run into some trouble writing operator `+` since the code for concatenating two C strings is tricky. If you'll notice, though, we already have a working version of string concatenation in the body of operator `+=`. To unify our code, we'll therefore implement operator `+` in terms of operator `+=`. The full version of this code is shown below:

```
const CString CString::operator +(const CString& other) const
{
    CString result = *this; // Make a deep copy of this CString.
    result += other;         // Use existing concatenation code.
    return result;
}
```

Now, all of the code for operator `+` is unified, which helps cut down on coding errors.

There is an interesting and common case we haven't addressed yet – what if one of the operands isn't of the same type as the class? For example, if you have a `Matrix` class that encapsulates a 3x3 matrix, as shown here:

```

class Matrix
{
public:
    /* ... */

    Matrix& operator *= (double scalar); // Scale all entries

private:
    static const int MATRIX_SIZE = 3;
    double entries[MATRIX_SIZE][MATRIX_SIZE];
};

```

Note that there is a defined `*=` operator that scales all elements in the matrix by a double factor. Thus code like `myMatrix *= 2.71828` is well-defined. However, since there's no defined operator `*`, currently we cannot write `myMatrix = myMatrix * 2.71828`.

Initially, you might think that we could define operator `*` just as we did operator `+` in the previous example. While this will work in most cases, it will lead to some problems we'll need to address later. For now, however, let's add the member function operator `*` to `Matrix`, which is defined as

```

const Matrix Matrix::operator *(double scalar) const
{
    MyMatrix result = *this;
    result *= scalar;
    return result;
}

```

Now, we can write expressions like `myMatrix = myMatrix * 2.71828`. However, what happens if we write code like `myMatrix = 2.71828 * myMatrix`? This is a semantically meaningful expression, but unfortunately it won't compile. When interpreting overloaded operators, C++ will always preserve the order of values in an expression.* Thus `2.71828 * myMatrix` is *not* the same as `myMatrix * 2.71828`. Remember that the reason that `myMatrix * 2.71828` is legal is because it's equivalent to `myMatrix.operator *(2.71828)`. The expression `2.71828 * myMatrix`, on the other hand, is illegal because C++ will try to expand it into `(2.71828).operator *(myMatrix)`, which makes no sense.

Up to this point, we've only seen overloaded operators as member functions, usually because the operators act relative to some receiving object, but in C++ it's also legal to define overloaded operators as free functions. When defining an overloaded operator as a free function, you simply define a global function named `operator op` (where `op` is the operator in question) that accepts the proper number of parameters. When using this overloaded operator in code, it will expand to a call to the global function. For example, if there is a global operator `+` function, then the line `one + two` would expand into `operator +(one, two)`. This is exactly what we need to solve this problem. Let's make operator `*` a free function that accepts two parameters, a double and a `Matrix`, and returns a `const Matrix`. Thus code like `2.71828 * myMatrix` will expand into calls to `operator *(2.71828, myMatrix)`. The new version of operator `*` is defined below:

```

const Matrix operator * (double scalar, const Matrix& matrix)
{
    Matrix result = *matrix;
    matrix *= scalar;
    return result;
}

```

* One major reason for this is that sometimes the arithmetic operators won't be commutative. For example, given matrices **A** and **B**, **AB** is not necessarily the same as **BA**, and if C++ were to arbitrarily flip parameters it could result in some extremely difficult-to-track bugs.

But here we run into the same problem as before if we write `myMatrix * 2.71828`, since we haven't defined a function accepting a `Matrix` as its first parameter and an `double` as its second. To fix this, we'll define a *second* free function `operator *` with the parameters reversed that's implemented as a call to the other version:

```
const Matrix operator *(const Matrix& matrix, double scalar)
{
    return scalar * matrix; // Calls operator* (scalar, matrix)
}
```

As a general rule, mathematical operators like `+` should always be implemented as free functions. This prevents problems like those described here.

One important point to notice about overloading the mathematical operators versus the compound assignment operators is that it's considerably faster to use the compound assignment operators over the standalone mathematical operators. Not only do the compound assignment operators work in-place (that is, they modify existing objects), but they also return references instead of full objects. From a performance standpoint, this means that given these three strings:

```
string one = "This ";
string two = "is a ";
string three = "string!";
```

Consider these two code snippets to concatenate all three strings:

```
/* Using += */
string myString = one;
myString += two;
myString += three;

/* Using + */
string myString = one + two + three
```

Oddly, the second version of this code is considerably slower than the first because the `+` operator generates temporary objects. Remember that `one + two + three` is equivalent to

```
operator +(one, operator +(two, three))
```

Each call to `operator +` returns a new `string` formed by concatenating the parameters, so the code `one + two + three` creates two temporary `string` objects. The first version, on the other hand, generates no temporary objects since the `+=` operator works in-place. Thus while the first version is less sightly, it is significantly faster than the second.

Overloading `++` and `--`

Overloading the increment and decrement operators can be a bit tricky because there are two versions of each operator: *prefix* and *postfix*. Recall that `x++` and `++x` are different operations – the first will evaluate to the value of `x`, then increment `x`, while the second will increment `x` and then evaluate to the updated value of `x`. You can see this below:

```
int x = 0
cout << x++ << endl; // Prints: 0
cout << x << endl;   // Prints: 1

x = 0;
cout << ++x << endl; // Prints: 1
cout << x << endl;   // Prints: 1
```

Although this distinction is subtle, it's tremendously important for efficiency reasons. In the postfix version of `++`, since we have to return the value of the variable was before it was incremented, we'll need to make a full copy of the old version and then return it. With the prefix `++`, since we're returning the current value of the variable, we can simply return a reference to it. Thus the postfix `++` can be noticeably slower than the prefix version; this is the reason that when advancing an STL iterator it's faster to use the prefix increment operator.

The next question we need to address is how we can legally use `++` and `--` in regular code. Unfortunately, it can get a bit complicated. For example, the following code is totally legal:

```
int x = 0;
+++++++x; // Increments x seven times.
```

This is legal because it's equivalent to

```
++(++(++(++(++(++x)))));
```

The prefix `++` operator returns the variable being incremented as an *lvalue*, so this statement means “increment `x`, then increment `x` again, etc.”

However, if we use the postfix version of `++`, as seen here:

```
x+++++++; // Error
```

We get a compile-time error because `x++` returns the original value of `x` as an *rvalue*, which can't be incremented because that would require putting the *rvalue* on the left side of an assignment (in particular, `x = x + 1`).

Now, let's actually get into some code. Unfortunately, we can't just sit down and write `operator ++`, since it's unclear *which* `operator ++` we'd be overloading. C++ uses a hack to differentiate between the prefix and postfix versions of the increment operator: when overloading the prefix version of `++` or `--`, you write `operator ++` as a function that takes no parameters. To overload the postfix version, you'll overload `operator ++`, but the overloaded operator will accept as a parameter the integer value 0. In code, these two declarations look like

```
MyClass& operator ++(); // Prefix
const MyClass operator ++(int dummy); // Postfix
```

Note that the prefix version returns a `MyClass&` as an *lvalue* and the postfix version a `const MyClass` as an *rvalue*.

We're allowed to implement `++` and `--` in any way we see fit. However, one of the more common tricks is to write the `++` implementation as a wrapped call to `operator +=`. Assuming you've provided this function, we can then write the prefix `operator ++` as

```
MyClass& MyClass::operator ++()
{
    *this += 1;
    return *this;
}
```

And the postfix `operator ++` as

```
const MyClass MyClass::operator ++(int dummy)
{
    MyClass oldValue = *this; // Store the current value of the object.
    *this += 1;
    return oldValue;
}
```

Overloading Relational Operators

Perhaps the most commonly overloaded operators (other than `operator =`) are the relational operators; for example, `<` and `==`. Unlike the assignment operator, by default C++ does not provide relational operators for your objects. This means that you must explicitly overload the `==` and related operators to use them in code. The prototype for the relational operators looks like this (written for `<`, but can be for any of the relational operators):

```
bool operator < (const MyClass& other) const;
```

You're free to choose any means for defining what it means for one object to be “less than” another. What's important is consistency. That is, if `one < two`, we should also have `one != two` and `!(one >= two)`. In fact, you may want to consider defining just the `<` operator and then implementing `==`, `<=`, `!=`, `>`, and `>=` as wrapper calls.

Storing Objects in STL maps

Up to this point we've avoided storing objects as keys in STL maps. Now that we've covered operator overloading, though, you have the necessary knowledge to store objects in the STL `map` and `set` containers.

Internally, the STL `map` and `set` are layered on binary trees that use the relational operators to compare elements. Due to some clever design decisions, STL containers and algorithms only require the `<` operator to compare two objects. Thus, to store a custom class inside a `map` or `set`, you simply need to overload the `<` operator and the STL will handle the rest. For example, here's some code to store a `Point` struct in a `map`:

```
struct pointT
{
    int x, y;
    bool operator < (const pointT& other) const
    {
        if(x != other.x)
            return x < other.x;
        return y < other.y;
    }
};
map<pointT, int> myMap; // Now works using the default < operator.
```

You can use a similar trick to store objects as values in a `set`.

friend

Normally, when you mark a class's data members private, only instances of that class are allowed to access them. However, in some cases you might want to allow specific other classes or functions to modify private data. For example, if you were implementing the STL `map` and wanted to provide an iterator class to traverse it, you'd want that iterator to have access to the `map`'s underlying binary tree. There's a slight problem here, though. Although the iterator is an integral component of the `map`, like all other classes, the iterator cannot access private data and thus cannot traverse the tree.

How are we to resolve this problem? Your initial thought might be to make some public accessor methods that would let the iterator modify the object's internal data representation. Unfortunately, this won't work particularly well, since then *any* class would be allowed to use those functions, something that violates the principle of encapsulation. Instead, to solve this problem, we can use the C++ `friend` keyword to grant the iterator class access to the `map` or `set`'s internals. Inside the `map` declaration, we can write the following:

```
template <typename KeyType, typename ValueType> class map
{
public:
    /* ... */

    friend class iterator;
    class iterator
    {
        /* ... iterator implementation here ... */
    };
};
```

Now, since `iterator` is a friend of `map`, it can read and modify the `map`'s private data members.

Just as we can grant other classes friend access to a class, we can give friend access to global functions. For example, if we had a free function `ModifyMyClass` that accepted a `MyClass` object as a reference parameter, we could let `ModifyMyClass` modify the internal data of `MyClass` if inside the `MyClass` declaration we added the line

```
class MyClass
{
public:
    /* ... */

    friend void ModifyMyClass(MyClass& param);
};
```

The syntax for `friend` can be misleading. Even though we're prototyping `ModifyMyClass` inside the `MyClass` function, because `ModifyMyClass` is a friend of `MyClass` it is **not** a member function of `MyClass`. After all, the purpose of the `friend` declaration is to give outside classes and functions access to the `MyClass` internals.

When using `friend`, there are two key points to be aware of. First, the `friend` declaration must precede the actual implementation of the `friend` class or function. Since C++ compilers only make a single pass over the source file, if they haven't seen a `friend` declaration for a function or class, when the function or class tries to modify your object's internals, the compiler will generate an error. Second, note that while `friend` is quite useful in some circumstances, it can quickly lead to code that entirely defeats the purpose of encapsulation. Before you grant `friend` access to a piece of code, make sure that the code has a legitimate reason to be modifying your object. That is, don't make code a `friend` simply because it's easier to write that way. Think of `friend` as a way of extending a class definition to include other pieces of code. The class, together with all its `friend` code, should comprise a logical unit of encapsulation.

When overloading an operator as a free function, you might want to consider giving that function `friend` access to your class. That way, the functions can efficiently read your object's private data without having to go through getters and setters.

Unfortunately, `friend` does not interact particularly intuitively with template classes. Suppose we want to provide a `friend` function `PQueueFriend` for a template version of the CS106B/X `PQueue`. If `PQueueFriend` is declared like this:


```
template <typename T> void PQueueFriend(const PQueue<T>& pq)
{
    /* ... */
}
```

You'll notice that `PQueueFriend` itself is a template function. This means that when declaring `PQueueFriend` a friend of the template `PQueue`, we'll need to make the friend declaration templated, as shown here:

```
template <typename T> class PQueue
{
public:
    template <typename T> friend PQueueFriend(const PQueue<T>& pq);
    /* ... */
};
```

If you forget the `template` declaration, then your code will compile correctly but will generate a linker error. While this can be a bit of nuisance, it's important to remember since it arises frequently when overloading the stream operators, as you'll see below.

Overloading the Stream Insertion Operator

Have you ever wondered why `cout << "Hello, world!" << endl` is syntactically legal? It's through the overloaded `<<` operator in conjunction with `ostreams`.^{*} In fact, the entire streams library can be thought of as a gigantic library of massively overloaded `<<` and `>>` operators.

The C++ streams library is designed to give you maximum flexibility with your input and output routines and even lets you define your own stream insertion and extraction operators. This means that you are allowed to define the `<<` and `>>` operators so that expressions like `cout << myClass << endl` and `cin >> myClass` are well-defined. However, when writing stream insertion and extraction operators, there are huge number of considerations to keep in mind, many of which are beyond the scope of this text. This next section will discuss basic strategies for overloading the `<<` operator, along with some limitations of the simple approach.

As with all overloaded operators, we need to consider what the parameters and return type should be for our overloaded `<<` operator. Before considering parameters, let's think of the return type. We know that it should be legal to chain stream insertions together – that is, code like `cout << 1 << 2 << endl` should compile correctly. The `<<` operator associates to the left, so the above code is equal to

```
((cout << 1) << 2) << endl);
```

Thus, we need the `<<` operator to return an `ostream`. Now, we don't want this stream to be `const`, since then we couldn't write code like this:

```
cout << "This is a string!" << setw(10) << endl;
```

Since if `cout << "This is a string!"` evaluated to a `const` object, we couldn't set the width of the next operation to 10. Also, we cannot return the stream by value, since stream classes have their copy functions marked private. Putting these two things together, we see that the stream operators should return a non-`const` reference to whatever stream they're referencing.

Now let's consider what parameters we need. We need to know what stream we want to write to or read from, so initially you might think that we'd define overloaded stream operators as member functions that look like this:

^{*} As a reminder, the `ostream` class is the base class for output streams. This has to do with inheritance, which we'll cover in a later chapter, but for now just realize that it means that both `stringstream` and `ofstream` are specializations of the more generic `ostream` class.

```
class MyClass
{
public:
    ostream& operator <<(ostream& input) const;
};
```

Unfortunately, this isn't correct. Consider the following two code snippets:

```
cout << myClass;
myClass << cout;
```

The first of these two versions makes sense, while the second is backwards. Unfortunately, with the above definition of `operator <<`, we've accidentally made the second version syntactically legal. The reason is that these two lines expand into calls to

```
cout.operator <<(myClass);
myClass.operator <<(cout);
```

The first of these two isn't defined, since `cout` doesn't have a member function capable of writing our object (if it did, we wouldn't need to write a stream operator in the first place!). However, based on our previous definition, the second version, while semantically incorrect, is syntactically legal. Somehow we need to change how we define the stream operator so that we are allowed to write `cout << myClass`. To fix this, we'll make the overloaded stream operator a free function that takes two parameters – an `ostream` to write to and a `myClass` object to write. The code for this is:

```
ostream& operator << (ostream& stream, const MyClass& mc)
{
    /* ... implementation ... */
    return stream;
}
```

While this code will work correctly, because `operator <<` is a free function, it doesn't have access to any of the private data members of `MyClass`. This can be a nuisance, since we'd like to directly write the contents of `MyClass` out to the stream without having to go through the (possibly inefficient) getters and setters. Thus, we'll declare `operator <<` a friend inside the `MyClass` declaration, as shown here:

```
class MyClass
{
public:
    /* More functions. */
    friend ostream& operator <<(ostream& stream, const MyClass& mc);
};
```

Now, we're all set to do reading and writing inside the body of the insertion operator. It's not particularly difficult to write the stream insertion operator – all that we need to do is print out all of the meaningful class information with some formatting information. So, for example, given a `Point` class representing a point in 2-D space, we could write the insertion operator as

```
ostream& operator <<(ostream& stream, const Point& pt)
{
    stream << '(' << pt.x << ", " << pt.y << ')';
    return stream;
}
```

While this code will work in most cases, there are a few spots where it just won't work correctly. For example, suppose we write the following code:

```
cout << "01234567890123456789" << endl; // To see the number of characters.
cout << setw(20) << myPoint << endl;
```

Looking at this code, you'd expect that it would cause `myPoint` to be printed out and padded with space characters until it is at least twenty characters wide. Unfortunately, this isn't what happens. Since `operator <<` writes the object one piece at a time, the output will look something like this:

```
01234567890123456789
          (0, 4)
```

That's nineteen spaces, followed by the actual `Point` data. The problem is that when we invoke `operator <<`, the function writes a single `(` character to stream. It's this operation, *not the `Point` as a whole*, that will get aligned to 20 characters. There are many ways to circumvent this problem, but perhaps the simplest is to buffer the output into a `stringstream` and then write the contents of the `stringstream` to the destination in a single operation. This can get a bit complicated, especially since you'll need to copy the stream formatting information over.

Writing a correct stream extraction operator (`operator >>`) is complicated. For more information on writing stream extraction operators, consult a reference.

Overloading `*` and `->`

Consider the following code snippet:

```
for(set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << " has length " << itr->length() << endl;
```

Here, we traverse a `set<string>` using iterators, printing out each string and its length. Interestingly, even though `set` iterators are not raw pointers (they're objects capable of traversing binary trees), thanks to operator overloading, they can respond to the `*` and `->` operators as though they were regular C++ pointers.

If you create a custom class that acts like a C++ pointer (perhaps a custom iterator or “smart pointer,” a topic we'll return to later), you can provide implementations of the pointer dereference and member selection operators `*` and `->` by overloading their respective operator functions. The simpler of these two functions is the pointer dereference operator. To make an object that can be dereferenced to yield an object of type `T`, the syntax for its `*` operator is

```
T& operator *() const;
```

You can invoke the `operator *` function by “dereferencing” the custom pointer object. For example, the following code:

```
*myCustomPointer = 137;
```

is completely equivalent to

```
myCustomPointer.operator *() = 137;
```

Because we can assign a value to the result of `operator *`, the `operator *` function should return an lvalue (a non-const reference).

There are two other points worth noting here. First, how can C++ distinguish this `operator *` for pointer dereference from the `operator *` used for multiplication? The answer has to do with the number of parameters to the function. Since a pointer dereference is a unary operator, the function prototype for the pointer-

dereferencing `operator *` takes no parameters. Had we wanted to write `operator *` for multiplication, we would have written a function `operator *` that accepts a parameter (or a free function accepting two parameters). Second, why is `operator *` marked `const`? This has to do with the difference between `const` pointers and pointers-to-`const`. Suppose that we have a `const` instance of a custom pointer class. Since the pointer *object* is `const`, it acts as though it is a `const` pointer rather than a pointer-to-`const`. Consequently, we should be able to dereference the object and modify its stored pointer without affecting its `constness`.

The arrow operator `operator ->` is slightly more complicated than `operator *`. Initially, you might think that `operator ->` would be a binary operator, since you use the arrow operator in statements like `myCppClassPtr->myElement`. However, C++ has a rather clever mechanism for `operator ->` that makes it a unary operator. A class's `operator ->` function should return a pointer to the object that the arrow operator should actually be applied to. This may be a bit confusing, so an example is in order. Suppose we have a class `CustomStringPointer` that acts as though it's a pointer to a C++ string object. Then if we have the following code:

```
CustomStringPointer myCustomPointer;
cout << myCustomPointer->length() << endl;
```

This code is equivalent to

```
CustomStringPointer myCustomPointer;
cout << (myCustomPointer.operator ->())->length() << endl;
```

In the first version of the code, we treated the `myCustomPointer` object as though it was a real pointer by using the arrow operator to select the `length` function. This code expands out into two smaller steps:

1. The `CustomStringPointer`'s `operator ->` function is called to determine which pointer the arrow should be applied to.
2. The returned pointer then has the `->` operator applied to select the `length` function.

Thus when writing the `operator ->` function, you simply need to return the pointer that the arrow operator should be applied to. If you're writing a custom iterator class, for example, this is probably the element being iterator over.

We'll explore one example of overloading these operators in a later chapter.

List of Overloadable Operators

The following table lists the most commonly-used operators you're legally allowed to overload in C++, along with any restrictions about how you should define the operator.

Operator	Yields	Usage
=	Lvalue	<pre>MyClass& operator =(const MyClass& other);</pre> See the the earlier chapter for details.
+= -= *= /= %=	Lvalue	<pre>MyClass& operator +=(const MyClass& other);</pre> When writing compound assignment operators, make sure that you correctly handle “self-compound-assignment.”
+ - * / % (etc.)	Rvalue	<pre>const MyClass operator + (const MyClass& one, const MyClass& two);</pre> These operator should be defined as a free functions.
< <= == > >= !=	Rvalue	<pre>bool operator < (const MyClass& other) const; bool operator < (const MyClass& one, const MyClass& two);</pre> If you're planning to use relational operators only for the STL container classes, you just need to overload the < operator. Otherwise, you should overload all six so that users aren't surprised that one != two is illegal while !(one == two) is defined.
[]	Lvalue	<pre>ElemType& operator [] (const KeyType& key); const ElemType& operator [] (const KeyType& key) const;</pre> Most of the time you'll need a const-overloaded version of the bracket operator. Forgetting to provide one can lead to a real headache!
++ --	Prefix: Lvalue Postfix: Rvalue	<pre>Prefix version: MyClass& operator ++(); Postfix version: const MyClass operator ++(int dummy);</pre>
-	Rvalue	<pre>const MyClass operator -() const;</pre>
*	Lvalue	<pre>ElemType& operator *() const;</pre> With this function, you're allowing your class to act as though it's a pointer. The return type should be a reference to the object it's “pointing” at. This is how the STL iterators and smart pointers work. Note that this is the unary * operator and is not the same as the * multiplicative operator.
->	Lvalue	<pre>ElemType* operator ->() const;</pre> If the -> is overloaded for a class, whenever you write myClass->myMember, it's equivalent to myClass.operator ->()->myMember. Note that the function should be const even though the object returned can still modify data. This has to do with how pointers can legally be used in C++. For more information, refer to the chapter on const.
<< >>	Lvalue	<pre>friend ostream& operator << (ostream& out, const MyClass& mc); friend istream& operator >> (istream& in, MyClass& mc);</pre>
()	Varies	See the chapter on functors.

More to Explore

Operator overloading is an enormous topic in C++ and there's simply not enough space to cover it all in this chapter. If you're interested in some more advanced topics, consider reading into the following:

1. **Overloaded new and delete:** You are allowed to overload the `new` and `delete` operators, in case you want to change how memory is allocated for your class. Note that the overloaded `new` and `delete` operators simply change how memory is allocated, not what it means to write `new MyClass`. Overloading `new` and `delete` is a complicated task and requires a solid understanding of how C++ memory management works, so be sure to consult a reference for details.
2. **Conversion functions:** In an earlier chapter, we covered how to write conversion constructors, functions that convert objects of other types into instances of your new class. However, it's possible to use operator overloading to define an implicit conversion from objects of your class into objects of other types. The syntax is `operator Type()`, where `Type` is the data type to convert your object to. Many professional programmers advise against conversion functions, so make sure that they're really the best option before proceeding.

Practice Problems

Operator overloading is quite difficult because your functions must act as though they're the built-in operators. Here are some practice problems to get you used to overloading operators:

1. In Python, it is legal to use negative array indices to mean “the element that many positions from the end of the array.” For example, `myArray[-1]` would be the last element of an array, `myArray[-2]` the penultimate element, etc. Using operator overloading, it's possible to implement this functionality for a custom array class. Do you think it's a good idea to do so? Why or why not? Think about the principle of least astonishment when answering.
2. Why is it better to implement `+` in terms of `+=` instead of `+=` in terms of `+`? (*Hint: Think about the number of objects created using `+=` and using `+`.*)
3. Explain how it's possible to define all of the relational operators in terms of `<`. ♦
4. Recall that the `Vector3D` has the following data members in its private section:

```
static const int NUM_COORDINATES = 3;
double coordinates[NUM_COORDINATES];
```

Suppose that you want to be able to store `Vector3Ds` inside an STL set. You delegate this task to a friend who returns with the following implementation of `operator <`:

```
bool Vector3D::operator< (const Vector3D& other) const
{
    for(int k = 0; k < NUM_COORDINATES; ++k)
        if(coordinates[k] < other.coordinates[k]) return true;
    return false;
}
```

This implementation of `operator <` will cause serious problems if `Vector3Ds` are stored in an STL set or map. Why is this? (*Hint: What are the mathematical properties of the less-than operator, and do they all apply to this implementation?*)

5. Given a `CString` class that stores a C string as `char* theString`, implement `operator+=` for the `CString` class to perform a string concatenation. Make sure that you handle the case where a `CString` is concatenated with itself.

6. Consider the following definition of a `Span` struct:

```
struct Span
{
    int start, stop;
    Span(int begin, int end) : start(begin), stop(end) {}
};
```

The `Span` struct allows us to define the range of elements from `[start, stop)` as a single variable. Given this definition of `Span` and assuming `start` and `stop` are both non-negative, provide another bracket operator for `CString` that accepts a `Span` by reference-to-const and returns another `CString` equal to the substring of the initial `CString` from `start` to `stop`.

7. Consider the following interface for a class that iterates over a container of `ElemTypes`:

```
class iterator
{
public:
    bool operator== (const iterator& other);
    bool operator!= (const iterator& other);

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

There are several mistakes in the definition of this iterator. What are they? How would you fix them? ♦

8. `valarray` is a template class declared in `<valarray>` that encapsulates a fixed-size sequence of elements. Unlike `vector`, `valarray` is designed to be treated like a mathematical matrix or vector. For example, given two `valarrays` `one` and `two`, the code `one + two` yields a new `valarray` that is the componentwise sum of the two `valarrays`, while `one * two` would be their componentwise product. It is also possible to write code like `one += 5` to add five to each element in the `valarray`. Suppose that you are implementing `valarray<ElemType>` and you choose to implement it as a wrapper around a `vector<ElemType>` called `array`.^{*} Implement the `+` and `+=` functions for `valarray` so that it is possible to add two `valarrays` or a `valarray` and an `ElemType`. You can assume that any two `valarrays` being added have the same size.
9. The `find_if` algorithm takes three parameters – two iterators and a predicate function – and returns an iterator to the first element in the input range for which the predicate returns true. If the predicate never returns true, `find_if` returns the end of the range as a sentinel. One possible implementation of `find_if` is shown here:

```
template <typename InputIterator, typename Predicate> inline
InputIterator find_if(InputIterator start, InputIterator end, Predicate fn)
{
    while(start != end)
    {
        if(fn(*start)) return start;
        ++start;
    }
    return end;
}
```

Assuming that `find_if` is called on a pair of `vector<int>::iterators`, identify three places in this implementation that are calls to overloaded operators.

^{*} `valarray` would almost certainly not be implemented in terms of `vector` for performance reasons, but it simplifies this practice problem.

Chapter 23: Extended Example: SmartPointer

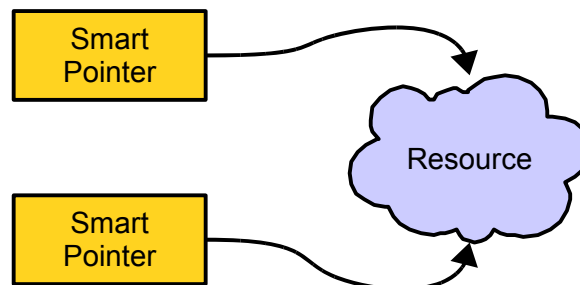
In C++ parlance, a raw pointer like an `int*` or a `char*` is sometimes called a *dumb pointer* because the pointer has no “knowledge” of the resource it owns. If an `int*` goes out of scope, it doesn't inform the object it's pointing at and makes no attempt whatsoever to clean it up. The `int*` doesn't own its resource, and assigning one `int*` to another doesn't make a deep copy of the resource or inform the other `int*` that another pointer now references its pointee.

Because raw pointers are so problematic, many C++ programmers prefer to use *smart pointers*, objects that mimic raw pointers but which perform functions beyond merely pointing at a resource. For example, the C++ standard library class `auto_ptr`, which we'll cover in the chapter on exception handling, acts like a regular pointer except that it automatically calls `delete` on the resource it owns when it goes out of scope. Other smart pointers are custom-tuned for specific applications and might perform functions like logging access, synchronizing multithreaded applications, or preventing accidental null pointer dereferences. Thanks to operator overloading, smart pointers can be built to look very similar to regular C++ pointers. We can provide an implementation of operator `*` to support dereferences like `*mySmartPointer`, and can define operator `->` to let clients write code to the effect of `mySmartPointer->clear()`. Similarly, we can write copy constructors and assignment operators for smart pointers that do more than just transfer a resource.

Reference Counting

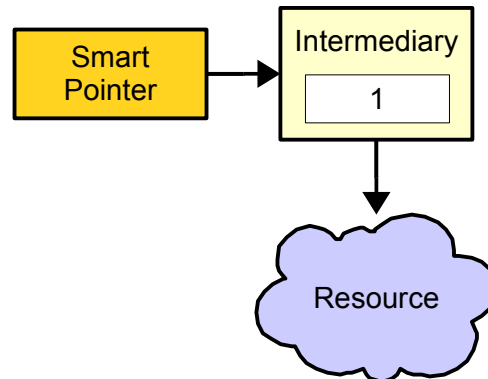
Memory management in C++ is tricky. You must be careful to balance every `new` with exactly one `delete`, and must make sure that no other pointers to the resource exist after `delete`-ing it to ensure that later on you don't access invalid memory. If you `delete` memory too many times you run into undefined behavior, and if you `delete` it too few you have a memory leak. Is there a better way to manage memory? In many cases, yes, and in this extended example we'll see one way to accomplish this using a technique called *reference counting*. In particular, we'll design a smart pointer class called `SmartPointer` which acts like a regular C++ pointer, except that it uses reference counting to prevent resource leaks.

To motivate reference counting, let's suppose that we have a smart pointer class that stores a pointer to a resource. The destructor for this smart pointer class can then `delete` the resource automatically, so clients of the smart pointer never need to explicitly clean up any resources. This system is fine in restricted circumstances, but runs into trouble as soon as we have several smart pointers pointing to the same resource. Consider the scenario below:



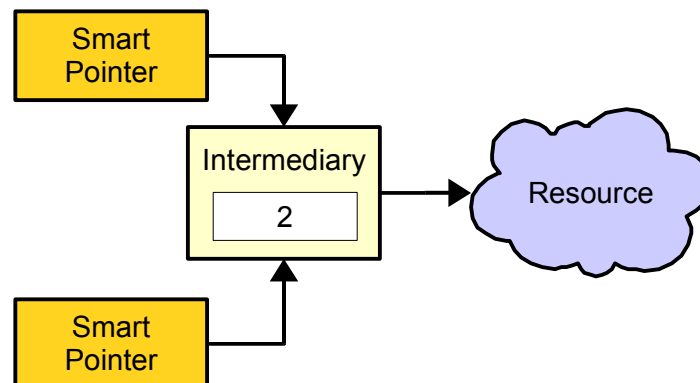
Both of these pointers can access the stored resource, but unfortunately neither smart pointer knows of the other's existence. Here we hit a snag. If one smart pointer cleans up the resource while the other still points to it, then the other smart pointer will point to invalid memory. If both of the pointers try to reclaim the dynamically-allocated memory, we will encounter a runtime error from double-`delete`-ing a resource. Finally, if neither pointer tries to clean up the memory, we'll get a memory leak.

To resolve this problem, we'll use a system called *reference counting* where we will explicitly keep track of the number of pointers to a dynamically-allocated resource. While there are several ways to make such a system work, perhaps the simplest is to use an intermediary object. This can be seen visually:



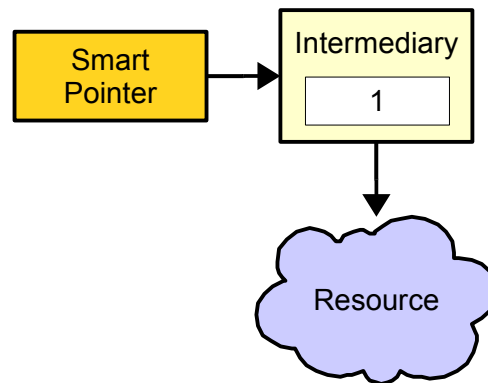
Now, the smart pointer stores a pointer to an intermediary object rather than a pointer directly to the resource. This intermediary object has a counter (called a *reference counter*) that tracks the number of smart pointers accessing the resource, as well as a pointer to the managed resource. This intermediary object lets the smart pointers tell whether or not they are the only pointer to the stored resource; if the reference count is anything other than one, some other pointer shares the resource. Provided that we accurately track the reference count, each pointer can tell if it's the last pointer that knows about the resource and can determine whether to deallocate it.

To see how reference counting works, let's walk through an example. Given the above system, suppose that we want to share the resource with another smart pointer. We simply make this new smart pointer point to the same intermediary object as our original pointer, then update the reference count. The resulting scenario looks like this:

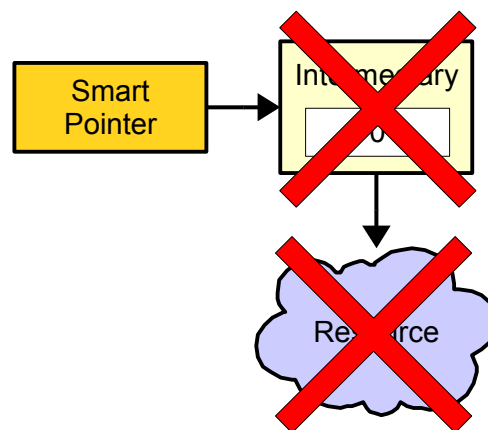


Although in this diagram we only have two objects pointing to the intermediary, the reference-counting system allows for any number of smart pointers to share a single resource.

Now, suppose one of these smart pointers needs to stop pointing to the resource – maybe it's being assigned to a different resource, or perhaps it's going out of scope. That pointer decrements the reference count of the intermediary variable and notices that the reference count is nonzero. This means that at least one smart pointer still references the resource, so the smart pointer simply leaves the resource as it is. Memory now looks like this:



Finally, suppose this last smart pointer needs to stop pointing to this resource. It decrements the reference count, but this time notices that the reference count is zero. This means that no other smart pointers reference this resource, and the smart pointer knows that it needs to deallocate the resource and the intermediary object, as shown here:



The resource has now been deallocated and no other pointers reference the memory. We've safely and effectively cleaned up our resources. Moreover, this process is completely automatic – the user never needs to explicitly deallocate any memory.

The following summarizes the reference-counting scheme described above:

- When creating a smart pointer to manage newly-allocated memory, first create an intermediary object and make the intermediary point to the resource. Then, attach the smart pointer to the intermediary and set the reference count to one.
- To make a new smart pointer point to the same resource as an existing one, make the new smart pointer point to the old smart pointer's intermediary object and increment the intermediary's reference count.
- To remove a smart pointer from a resource (either because the pointer goes out of scope or because it's being reassigned), decrement the intermediary object's reference count. If the count reaches zero, deallocate the resource and the intermediary object.

While reference counting is an excellent system for managing memory automatically, it does have its limitations. In particular, reference counting can sometimes fail to clean up memory in “reference cycles,” situations where multiple reference-counted pointers hold references to one another. If this happens, none of the reference counters can ever drop to zero, since the cyclically-linked elements always refer to one another. But barring this sort of setup, reference counting is an excellent way to automatically manage memory. In this extended example, we'll see how to implement a reference-counted pointer, which we'll call `SmartPointer`, and will explore how the correct cocktail of C++ constructs can make the resulting class slick and efficient.

Designing SmartPointer

The above section details the *implementation* the SmartPointer class, but we have not talked about its *interface*. What functions should we provide? We'll try to make SmartPointer resemble a raw C++ pointer as closely as possible, meaning that it should support operator `*` and operator `->` so that the client can dereference the SmartPointer. Here is one possible interface for the SmartPointer class:

```
template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;
};
```

Here is a breakdown of what each of these functions should do:

```
explicit SmartPointer(T* memory);
```

Constructs a new SmartPointer that manages the resource specified as the parameter. The reference count is initially set to one. We will assume that the provided pointer came from a call to `new`.

This function is marked `explicit` so that we cannot accidentally convert a regular C++ pointer to a SmartPointer. At first this might seem like a strange design decision, but it prevents a wide range of subtle bugs. For example, suppose that this constructor is not `explicit` and consider the following function:

```
void PrintString(const SmartPointer<string>& ptr)
{
    cout << *ptr << endl;
}
```

This function accepts a SmartPointer by reference-to-const, then prints out the stored string. Now, what happens if we write the following code?

```
string* ptr = new string("Yay!");
PrintString(ptr);
delete ptr;
```

The first line dynamically-allocates a string, passes it to PrintString, and finally deallocates it. Unfortunately, this code will almost certainly cause a runtime crash. The problem is that PrintString expects a SmartPointer<string> as a parameter, but we've provided a string*. C++ notices that the SmartPointer<string> has a conversion constructor that accepts a string*, and constructs a temporary SmartPointer<string> using the pointer we passed as a parameter. This new SmartPointer starts tracking the pointer with a reference count of one. After the function returns, the parameter is cleaned up and its destructor invokes. This decrements the reference count to zero, and then deallocates the pointer stored in the SmartPointer. The above code then tries to delete ptr a second time, causing a runtime crash.

To prevent this problem, we'll mark the constructor `explicit`, which makes the implicit conversion illegal and prevents this buggy code from compiling.

SmartPointer functions, contd.

SmartPointer(const SmartPointer& other);
Constructs a new <code>SmartPointer</code> that shares the resource contained in another <code>SmartPointer</code> , updating the reference count appropriately.
SmartPointer& operator=(const SmartPointer& other);
Causes this <code>SmartPointer</code> to stop pointing to the resource it's currently managing and to share the resource held by another <code>SmartPointer</code> . If the smart pointer was the last pointer to its resource, it deletes it.
~SmartPointer();
Detaches the <code>SmartPointer</code> from the resource it's sharing, freeing the associated memory if necessary.
T& operator* () const;
“Dereferences” the pointer and returns a reference to the object being pointed at. Note that <code>operator*</code> is <code>const</code> ; see the last chapter for more information why.
T* operator-> () const;
Returns the object that the arrow operator should really be applied to if the arrow is used on the <code>SmartPointer</code> . Again, see the last chapter for more information on this.

Given this public interface for `SmartPointer`, we can now begin implementing the class. We first need to decide on how we should represent the reference-counting information. One simple method is to define a private struct inside `SmartPointer` that represents the reference-counting intermediary. This looks as follows:

```
template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;
};
```

Here, the `resource` field of the `Intermediary` is the actual pointer to the stored resource and `refCount` is the reference count. Given this setup, we can implement the `SmartPointer` constructor by creating a new `Intermediary` that points to the specified resource and has an initial reference count of one:

```
template <typename T> SmartPointer<T>::SmartPointer(T* res)
{
    data = new Intermediary;
    data->resource = res;
    data->refCount = 1;
}
```

It's very important that we allocate the `Intermediary` object on the heap rather than as a data member. That way, when the `SmartPointer` is cleaned up (either by going out of scope or by an explicit call to `delete`), if it isn't the last pointer to the shared resource, the intermediary object isn't cleaned up.

We can similarly implement the destructor by decrementing the reference count, then cleaning up memory if appropriate. Note that if the reference count hits zero, we need to delete both the resource *and* the intermediary. Forgetting to deallocate either of these leads to memory leaks, the exact problem we wanted to avoid. The code for this is shown here:

```
template <typename T> SmartPointer<T>::~~SmartPointer()
{
    --data->refCount;
    if(data->refCount == 0)
    {
        delete data->resource;
        delete data;
    }
}
```

This is an interesting destructor in that it isn't guaranteed to actually clean up any memory. Of course, this is exactly the behavior we want, since the memory might be shared among multiple `SmartPointers`.

Implementing operator `*` and operator `->` simply requires us to access the pointer stored inside the `SmartPointer`. These two functions can be implemented as follows:*

```
template <typename T> T& SmartPointer<T>::operator * () const
{
    return *data->resource;
}
template <typename T> T* SmartPointer<T>::operator -> () const
{
    return data->resource;
}
```

Now, we need to implement the copy behavior for this `SmartPointer`. As you saw in the chapter on copy constructors and assignment operators, one way to do this is to write helper functions `clear` and `copyOther` which perform deallocation and copying. We will use a similar approach here, except using functions named `detach` and `attach` to make explicit the operations we're performing. This leads to the following definition of `SmartPointer`:

* It is common to see operator `->` implemented as

```
RetType* MyClass::operator -> () const
{
    return &***this;
}
```

`&***this` is interpreted by the compiler as `&(*(*this))`, which means “dereference the `this` pointer to get the receiver object, then dereference the receiver. Finally, return the address of the referenced object.” At times this may be the best way to implement operator `->`, but I advise against it in general because it's fairly cryptic.

```

template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

```

Now, what should these functions do? The first of these, `detach`, should detach the `SmartPointer` from the shared intermediary and clean up the memory if it was the last pointer to the shared resource. In case this sounds familiar, it's because this is exactly the behavior of the `SmartPointer` destructor. To avoid code duplication, we'll move the code from the destructor into `detach` as shown here:

```

template <typename T> void SmartPointer<T>::detach()
{
    --data->refCount;
    if(data->refCount == 0)
    {
        delete data->resource;
        delete data;
    }
}

```

We can then implement the destructor as a wrapped call to `detach`, as seen here:

```

template <typename T> SmartPointer<T>::~~SmartPointer()
{
    detach();
}

```

The `attach` function, on the other hand, makes this `SmartPointer` begin pointing to the specified `Intermediary` and increments the reference count. Here's one possible implementation of `attach`:

```

template <typename T> void SmartPointer<T>::attach(Intermediary* to)
{
    data = to;
    ++data->refCount;
}

```

Given these two functions, we can implement the copy constructor and assignment operator for `SmartPointer` as follows:

```

template <typename T> SmartPointer<T>::SmartPointer(const SmartPointer& other)
{
    attach(other.data);
}

template <typename T>
SmartPointer<T>& SmartPointer<T>::operator= (const SmartPointer& other)
{
    if(this != &other)
    {
        detach();
        attach(other.data);
    }
    return *this;
}

```

It is crucial that we check for self-assignment inside the `operator=` function, since otherwise we might destroy the data that we're trying to keep track of!

At this point we have a rather slick `SmartPointer` class. Here's some code demonstrating how a client might use `SmartPointer`:

```

SmartPointer<string> myPtr(new string);
*myPtr = "This is a string!";
cout << *myPtr << endl;

SmartPointer<string> other = myPtr;
cout << *other << endl;
cout << other->length() << endl;

```

The beauty of this code is that client code using a `SmartPointer<string>` looks almost identical to code using a regular C++ pointer. Isn't operator overloading wonderful?

Extending SmartPointer

The `SmartPointer` defined above is useful but lacks some important functionality. For example, suppose that we have the following function:

```
void DoSomething(string* ptr);
```

Suppose that we have a `SmartPointer<string>` managing a resource and that we want to pass the stored string as a parameter to `DoSomething`. Despite the fact that `SmartPointer<string>` mimics a `string*`, it technically is not a `string*` and C++ won't allow us to pass the `SmartPointer` into `DoSomething`. Somehow we need a way to have the `SmartPointer` hand back the resource it manages.

Notice that the only `SmartPointer` member functions that give back a pointer or reference to the actual resource are `operator*` and `operator->`. Technically speaking, we *could* use these functions to pass the stored string into `DoSomething`, but the syntax would be messy (in the case of `operator*`) or nightmarish (for `operator->`). For example:


```
SmartPointer<string> myPtr(new string);

/* To use operator* to get the stored resource, we have to first dereference the
 * SmartPointer, then use the address-of operator to convert the returned reference
 * into a pointer.
 */
DoSomething(&*myPtr);

/* To use operator-> to get the stored resource, we have to explicitly call the
 * operator-> function. Yikes!
 */
DoSomething(myPtr.operator-> ());
```

Something is clearly amiss and we cannot reasonably expect clients to write code like this routinely. We'll need to extend the `SmartPointer` class to provide a way to return the stored pointer directly.

Initially, we might consider adding a member function to the `SmartPointer` interface to returned the stored pointer. For example, we might add a `get` member function so that we could use `SmartPointer` as follows:

```
DoSomething(myPtr.get());
```

There is nothing technically wrong with this approach and in fact most smart pointer classes export a function like this one. However, this approach opens up the potential for extremely hard-to-detect bugs. In particular, suppose that we have a `SmartPointer` that points to an object that itself defines a `get` function. Then we can write both `myPtr.get()`, which retrieves the stored pointer, and `myPtr->get()`, which invokes the `get` function of the stored resource. `SmartPointer` clients might accidentally call the wrong function but have trouble tracking down the source of the error because the two are almost syntactically identical. To sidestep these sorts of problems, instead we'll define a *free function* which accepts the `SmartPointer` as a parameter and returns the stored resource.* For example, if the function were called `Get`, we could do the following:

```
DoSomething(Get(myPtr));
```

Which is clearly syntactically distinct from `myPtr->get()` and avoids this sort of problem.

The main problem with this approach is that this free function `Get` needs to be able to access the private data members of `SmartPointer` in order to return the stored resource. Consequently, we'll define `Get` as a friend of `SmartPointer`, as shown here:

* The idea of making this operation a free function is based on the discussion of smart pointers in Andrei Alexandrescu's excellent book *Modern C++ Design*.

```

template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

    template <typename U> friend U* Get(const SmartPointer<U>& toGet);

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

```

Since the `Get` function we'll be writing is itself a template, we have to make the `friend` declaration a template as well. Given this declaration, we can then implement `Get` as follows:

```

template <typename T> T* Get(const SmartPointer<T>& toGet)
{
    return toGet.data->resource;
}

```

Further Extensions

There are several more extensions to the `SmartPointer` class that we might want to consider, of which this section explores two. The first is rather straightforward. At times, we might want to know exactly how many `SmartPointers` share a resource. This might enable us to perform some optimizations, in particular a technique called *copy-on-write*. We will not explore this technique here, though you are encouraged to do so on your own.

Using the same logic as above, we'll define another friend function called `GetShareCount` which accepts a `SmartPointer` and returns the number of `SmartPointers` accessing that resource, including the `SmartPointer` itself. This results in the following class definition:

```

template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

    template <typename U> friend U* Get(const SmartPointer<U>& toGet);
    template <typename U> friend int GetShareCount(const SmartPointer<U>& toGet);

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

```

And the following implementation:

```

template <typename T> int GetShareCount(const SmartPointer<T>& toGet)
{
    return toGet.data->refCount;
}

```

The last piece of functionality we'll consider is the ability to “reset” the `SmartPointer` to point to a different resource. When working with a `SmartPointer`, at times we may just want to drop whatever resource we're holding and begin managing a new one. As you might have suspected, we'll add yet another `friend` function called `Reset` which resets the `SmartPointer` to point to a new resource. The final interface and code for `Reset` is shown here:

```

template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

    template <typename U> friend U* Get(const SmartPointer<U>& toGet);
    template <typename U> friend int GetShareCount(const SmartPointer<U>& toGet);
    template <typename U> friend void Reset(SmartPointer<U>& toSet, U* resource);

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

template <typename T> void Reset(SmartPointer<T>& toSet, T* resource)
{
    /* We're no longer associated with our current resource, so drop it. */
    toSet.detach();

    /* Attach to a new intermediary object. */
    toSet.data = new typename SmartPointer<T>::Intermediary;
    toSet.data->resource = resource;
    toSet.data->refCount = 1
}

```

When we change the data field to point to a new Intermediary object, we use the syntax `new typename SmartPointer<T>::Intermediary`. Intermediary is defined inside SmartPointer, a template class, and so we must use the `typename` keyword to indicate that `SmartPointer<T>::Intermediary` is the name of a type. See the chapter on templates if you need a refresher on `typename`.

More to Explore

Smart pointers extend beyond the interface we've provided here. If you're up for a challenge and want to flex your C++ muscles, try implementing some of these changes to `SmartPointer`:

1. The `SmartPointer` class designed here simply performs resource management, but it's possible to build smart pointers that perform other tasks as well. For example, the smart pointer might check that the resource is non-NULL before dereferencing it, or might perform usage logging to help monitor performance. Modify the existing `SmartPointer` class to support one of these extra pieces of functionality.
2. As mentioned at the start of this chapter, reference counting can lead to memory leaks due to reference cycles. One method for breaking these cycles is to pair smart pointers with *weak pointers*. Weak pointers can look at the resource pointed at by a smart pointer, but do not increment its reference count. In other words, weak pointers can *view* shared resources, but they don't *own* them. Weak pointers can then be “locked,” returning a new `SmartPointer` pointing to the shared resource. Implement a `WeakPointer` class to pair with the `SmartPointer` implementation shown here.
3. The implementation of reference counting we implemented here allocates an intermediary object that keeps track of the number of smart pointers sharing the stored resource. Alternatively, we can design the classes that will be pointed at by our `SmartPointer` class so that they contain their own internal reference count which can be accessed only by the `SmartPointer` class. This eliminates the need for the intermediary object and increases the overall efficiency of the reference counting. Experiment with this means of reference counting – do you notice any performance improvements?
4. Another way of performing reference counting without explicitly allocating an intermediary object uses a technique called *reference linking*. In addition to tracking the shared resource, each smart pointer acts as a cell in a doubly-linked list, storing a pointer to the next and previous smart pointer that points to the resource. Whenever a new smart pointer is attached to an existing pointer, it is spliced into the list, and each time a smart pointer is detached from the resource it is removed from the list. A resource is cleaned up when a smart pointer with no next or previous pointer is detached from it. Rewrite the `SmartPointer` class to use reference linking. What changes to its public interface do you need to make?
5. `SmartPointer` cleans up resources allocated by `new` by ensuring that there is exactly one call to `delete`. However, there are other pairs of allocator/deallocator functions for which a `SmartPointer`-like class might be useful. For example, the C-style file reading routine `fopen` returns an object which must be closed with a call to `fclose`. Extend the `SmartPointer` so that the stored resource can be allocated and deallocated with an arbitrary allocator and deallocator.
6. If you are familiar with inheritance (or have read the later chapters of this reader), then you have seen how in some cases it is possible to convert pointers of one type into pointers of a different type. See if you can use the template system to devise a way of converting `SmartPointers` that point to a derived class type to `SmartPointers` which point to a base class type.

Complete SmartPointer Implementation

```

template <typename T> class SmartPointer
{
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *   () const;
    T* operator -> () const;

    template <typename U> friend U* Get(const SmartPointer<U>& toGet);
    template <typename U> friend int GetShareCount(const SmartPointer<U>& toGet);
    template <typename U> friend void Reset(SmartPointer<U>& toSet, U* resource);

private:
    struct Intermediary
    {
        T* resource;
        int refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

/* Constructor sets the SmartPointer to manage a resource, which now has reference
 * count one.
 */
template <typename T> SmartPointer<T>::SmartPointer(T* resource)
{
    data = new Intermediary;
    data->resource = resource;
    data->refCount = 1;
}

/* Destructor pulls us off of this resource and cleans it up if necessary. */
template <typename T> SmartPointer<T>::~~SmartPointer()
{
    detach();
}

/* attach accepts an intermediary and sets the SmartPointer to point to it. It
 * then increments the reference-count.
 */
template <typename T> void SmartPointer<T>::attach(Intermediary* to)
{
    data = to;
    ++data->refCount;
}

```

```

/* detach removes this pointer from the resource, dropping the reference count, and
 * cleaning up the resource if necessary.
 */
template <typename T> void SmartPointer<T>::detach()
{
    --data->refCount;
    if(data->refCount == 0)
    {
        /* Don't forget to delete both the resource and the intermediary! */
        delete data->resource;
        delete data;
    }
}

/* Copy constructor just attaches us to the resource stored in the other
 * SmartPointer.
 */
template <typename T> SmartPointer<T>::SmartPointer(const SmartPointer& other)
{
    attach(other.data);
}

/* Assignment operator does all of the quirky assignment operator mechanics,
 * but ultimately detaches us from the current resource and attaches us to the
 * other resource.
 */
template <typename T>
SmartPointer<T>& SmartPointer<T>::operator= (const SmartPointer& other)
{
    if(this != &other)
    {
        detach();
        attach(other.data);
    }
    return *this;
}

/* Pointer-dereference operator returns a reference to the resource. */
template <typename T> T& SmartPointer<T>::operator * () const
{
    return *data->resource;
}

/* Arrow operator just returns the stored resource. */
template <typename T> T* SmartPointer<T>::operator -> () const
{
    return data->resource;
}

/* Free function Get retrieves the stored resource. */
template <typename T> T* Get(const SmartPointer<T>& toGet)
{
    return toGet.data->resource;
}

/* Free function GetShareCount returns the reference count. */
template <typename T> int GetShareCount(const SmartPointer<T>& toGet)
{
    return toGet.data->refCount;
}

```

```
/* Free function Reset resets the pointer to refer to a new resource. */
template <typename T> void Reset(SmartPointer<T>& toSet, T* resource)
{
    /* We're no longer associated with our current resource, so drop it. */
    toSet.detach();

    /* Attach to a new intermediary object. */
    toSet.data = new typename SmartPointer<T>::Intermediary;
    toSet.data->resource = resource;
    toSet.data->refCount = 1;
}
```


Chapter 24: Extended Example: DimensionType

Never put off until run time what you can do at compile time.

– David Gries

Compilers are excellent at detecting type errors. If you try to treat an `int` as a `string`, the compiler will require you to fix the error before you can run the program. Provided that you stay away from unsafe typecasts and make sure not to use garbage pointers, if the compiler compiles your program, you can be reasonably confident that you didn't mix and match variables of different types.

However, when it comes to other sorts of bugs, C++ compilers are silent. For example, the following is legal C++ code, even though it's guaranteed to cause a runtime crash:

```
int* myPtr = NULL;
*myPtr = 137;           // Dereference NULL!
```

Similarly, not all compilers will detect this simple error:

```
int zero = 0;
int error = 1 / zero; // Divide by zero!
```

Debugging is never pleasant and we'd like as much as possible for the compiler to automatically detect bugs for us. But as mentioned above, the compiler only checks for a small set of errors, mostly involving breaches of the type system. What if there was some way that we could encode information about what the program is doing at the type level? That way, in the course of checking the types of the variables in the program, the compiler would also check for bugs in the code. You have already seen an example of this sort of checking with `const`, which allows the compiler to verify that code that should not overwrite data does not overwrite data. In this extended example, we'll explore one instance where harnessing the type system leads to better, simpler, and less error-prone code. In doing so, we'll see a novel use of templates that is sure to change the way you think about coding in C++.

Dimensional Analysis

Suppose that we want to write a physics simulation where objects interact with each other according to Newton's laws. For example, we might drop a 5kg mass onto a lever of length 10m with the fulcrum 3m from one side with gravity operating at 9.8m/s². Since each of these quantities has a real-numbered value, we initially store them as `doubles`. For example:

```
double mass = 5;           // 5kg
double length = 10;        // 10m
double gravity = 9.8;       // 9.8m/s2
```

Now, let's suppose that we want to compute the weight of the 5kg block in the gravity field. This value is computed by taking the product of the mass of the object and gravitational acceleration, so we could write

```
double weight = mass * gravity;
```

However, suppose that we accidentally make a typo in our code and instead write

```
double weight = mass + gravity;
```

This value is incorrect and if we try to use it in a critical system we might have a catastrophic failure on our hands. But worse, the C++ compiler won't give us any indication that this code contains an error because it's perfectly legal C++. There's nothing wrong with adding `doubles` to `doubles`, and if the formula we're using is incorrect there's no way for the compiler to figure this out.

The problem here is that the values we're storing are not real-numbered values. The value of 5kg is five *kilograms*, not just *five*, and similarly 9.8m/s² is 9.8 *meters per second squared*, not just 9.8. This additional information is lost if we store the values as `doubles`. If the compiler *did* have this information, it could point out that the sum of 5kg and 9.8m/s² is meaningless. Moreover, if we indicated that the type of the `weight` variable should be in newtons,* the compiler could check that the expression being assigned to `weight` was of the proper type. Thus we could not write code to the effect of `weight = mass` since `mass` has units kg and `weight` has units kg·m/s².

This system of checking that the units in a computation agree is known as *dimensional analysis* and will be the focus of this extended example. Our goal will be to develop a system for encoding the units associated with each variable into the type system so that we can detect unit errors effectively. The question, of course, is how we can do this.

A First Approach

One solution to the dimensional analysis problem is to construct a series of types representing values with specific units, then to define conversions between each of them. For example, a variable representing a quantity of kilograms could be typed as a `kilogramT`, whereas a variable holding a duration of time would be in `secondT`. Ideally, we could write code to this effect:

```
kilogramT mass(5.0);           // 5.0 kg
accelerationT gravity(9.8);    // 9.8 m/s2
newtonT weight = mass * gravity;
```

We can make the syntax `mass * gravity` compile by overloading the `*` operator between `kilogramT` and `accelerationT`, and could similarly define addition, subtraction, etc. As an example, here's a possible implementation of `kilogramT`, `accelerationT`, and `newtonT`:

```
class kilogramT
{
public:
    explicit kilogramT(double amount) : quantity(amount) {}

    /* Accesses the stored value. */
    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};
```

* One newton (N) is equal to one kilogram meter per second squared and is a unit of force. It may help to remember that a newton is roughly the weight of an apple.

```

class accelerationT
{
public:
    explicit accelerationT(double amount) : quantity(amount) {}

    /* Accesses the stored value. */
    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};

class newtonT
{
public:
    explicit newtonT(double amount) : quantity(amount) {}

    /* Accesses the stored value. */
    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};

/* Define multiplication as a free function. Note that there are two versions here
 * because we can write mass * acceleration or acceleration * mass.
 */
const newtonT operator * (const kilogramT& m, const accelerationT& a)
{
    return newtonT(m.getQuantity() * a.getQuantity());
}
const newtonT operator * (const accelerationT& a, const kilogramT& m)
{
    return newtonT(m.getQuantity() * a.getQuantity());
}

```

The code above should be mostly self-explanatory – each class has a constructor that accepts a `double` and stores it internally and exports a `getQuantity` member function which returns the stored quantity. The two implementations of `operator *` each accept a `kilogramT` and an `accelerationT`, call the `getQuantity` member functions to yield the stored quantity, and return a `newtonT` object with quantity equal to the product of the two quantities.

In case you're not familiar with the syntax `return newtonT(m.getQuantity() * a.getQuantity())`, this is perfectly legal C++ and is a use of the *temporary object syntax*. The syntax `newtonT (m.getQuantity() * a.getQuantity())` looks like a call to the constructor of `newtonT` and this is not that far from the truth. In C++, you are allowed to create temporary objects for the duration of a single line of code by explicitly calling the object's constructor. In this code, we construct a temporary `newtonT` object and immediately return it.

Initially this approach seems like it's exactly what we're looking for – a way of encoding units into the type system to prevent accidental misuse. For example, we cannot write code like the following:

```

kilogramT mass(5.0);
newtonT weight = mass; // Error – no conversion from kilogramT to newtonT

```

This is an error because the line `newtonT weight = mass` will try to find a way of converting a `kilogramT` to a `newtonT` when no such conversion exists. Similarly, if we try writing

```
kilogramT mass(5.0);
accelerationT gravity(9.8);
newtonT weight = mass + gravity; // Error: operator+ not defined here.
```

We'll get a compile-time error because there is no `+` operator defined on arguments of type `kilogramT` and `accelerationT`.

However, this approach has its limitations. For example, let's suppose that we now want to introduce two more types into the mix, `secondT` representing seconds and `meterT` representing distance. We can follow the above pattern and create the following types:

```
class meterT
{
public:
    explicit meterT(double amount) : quantity(amount) {}

    /* Accesses the stored value. */
    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};

class secondT
{
public:
    explicit secondT(double amount) : quantity(amount) {}

    /* Accesses the stored value. */
    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};
```

Now, one newton is equal to one kilogram meter per second squared, so if we multiply together a `kilogramT`, a `meterT`, and then divide by two `secondTs`, we should end up with a `newtonT`. But how can we represent this in code? We'd like to be able to write

```
kilogramT mass(1.0);
meterT distance(2.0);
secondT time(3.0);
newtonT force = mass * distance / (time * time);
```

If we want to be able to do this using overloaded operators, we'd need to overload the `*` operator applied to `kilogramT` and `meterT` as well as between `secondT` and `secondT`, plus the `/` operator applied to the results of these operations. But we don't currently have a type representing the product of a `kilogramT` and a `meterT`, so we'd need to introduce a `kilogramMeterT` type to hold the product of a `kilogramT` and a `meterT`, and similarly would need to make a `secondSquaredT` type to hold the product of a `secondT` and a `secondT`. If we were designing these classes as a library, we'd need to enumerate all reasonable combinations of kilograms,

meters, and seconds; define a type for each of these combinations of units; and then implement conversion operators between all of them. This is neither feasible nor sound. For example, if we define classes representing all combinations of kilograms, meters, and seconds from $\text{kg}^{-10} \cdot \text{m}^{-10} \cdot \text{s}^{-10}$ to $\text{kg}^{10} \cdot \text{m}^{10} \cdot \text{s}^{10}$, all it takes is one application requiring a kg^{11} and our code would be insufficient.

Moreover, think about the number of different implementations of `operator*` that would be necessary for this approach to work. Any two values can be multiplied regardless of their units, so if we define N classes, we'd need $O(N^2)$ different `operator*` functions defined between them. We'd similarly need $O(N)$ different `operator +` and `operator -` functions so that we can add and subtract units of the same type. That's an absurdly large amount of code and quickly becomes infeasible to write or maintain.

In short, this idea of creating custom classes for each combination of units is well-intentioned and works in small examples, but rapidly blows up into an unmanageable mess of code. We're going to have to try something else.

A Second Approach

One observation that can greatly simplify the implementation of types with units is exactly how values of different units can be multiplied and divided. For simplicity, this next discussion assumes that we're only dealing with kilograms, meters, seconds, and units derived from them (i.e. $\text{kg} \cdot \text{m}$ and $\text{kg} \cdot \text{m}^2/\text{s}^3$), but this analysis easily scales to more base types.

Suppose that we have two variables x and y , each representing a value with some associated units. For notational simplicity, we'll say that $x = (q_1, \text{kg}_1, \text{m}_1, \text{s}_1)$ if $x = q_1 \text{ kg}^{\text{kg}_1} \cdot \text{m}^{\text{m}_1} \cdot \text{s}^{\text{s}_1}$, where q_1 is a scalar. For example, if $x = 5\text{kg}$, then we'd represent this as $x = (5, 1, 0, 0)$ and similarly if $x = 20\text{kg} \cdot \text{m} \cdot \text{s}^{-2}$, we'd write that $x = (20, 1, 1, -2)$. Given this notation, we can define addition of dimensioned types as follows (subtraction can be defined similarly):

- $(q_1, \text{kg}_1, \text{m}_1, \text{s}_1) + (q_2, \text{kg}_2, \text{m}_2, \text{s}_2) = (q_1 + q_2, \text{kg}_1, \text{m}_1, \text{s}_1)$ if $\text{kg}_1 = \text{kg}_2$, $\text{m}_1 = \text{m}_2$, and $\text{s}_1 = \text{s}_2$. In other words, adding two values with the same units forms a new value that's the sum of the two quantities with the same units.
- $(q_1, \text{kg}_1, \text{m}_1, \text{s}_1) + (q_2, \text{kg}_2, \text{m}_2, \text{s}_2)$ is undefined otherwise. That is, you cannot add two values unless the units agree.

We can also define multiplication and division of dimensioned types like this:

- $(q_1, \text{kg}_1, \text{m}_1, \text{s}_1) \times (q_2, \text{kg}_2, \text{m}_2, \text{s}_2) = (q_1 q_2, \text{kg}_1 + \text{kg}_2, \text{m}_1 + \text{m}_2, \text{s}_1 + \text{s}_2)$. That is, multiplying two dimensioned values yields a new dimensioned value that's the product of the two quantities where the units are the sum of the units in the two dimensioned values. For example, $5\text{kg} \times 20\text{kg} \cdot \text{s} = 100\text{kg}^2 \cdot \text{s}$.
- $(q_1, \text{kg}_1, \text{m}_1, \text{s}_1) / (q_2, \text{kg}_2, \text{m}_2, \text{s}_2) = (q_1 / q_2, \text{kg}_1 - \text{kg}_2, \text{m}_1 - \text{m}_2, \text{s}_1 - \text{s}_2)$. That is, dividing two dimensioned values yields a new dimensioned value that's the quotient of the two quantities where the units are the difference of the units in the two dimensioned values. For example, $20\text{kg} / 5\text{kg} \cdot \text{s} = 4\text{s}^{-1}$.

These mathematical relations between dimensioned types suggests an implementation of a dimensioned type object in C++. The object internally stores both a quantity and the exponents of the kilograms, meters, and seconds of the units. We can then define the `*` and `/` operators by computing the product or quotient of the numbers and then adding or subtracting the units from the operands. Finally, we can define the `+` and `-` operators to check if the types agree and then return a new value with the same units and the value computed appropriately.

In code, this looks like this:

```

class DimensionType
{
public:
    /* DimensionType constructor accepts four values - the quantity and the three
     * unit exponents - then initializes the DimensionType appropriately.
     */
    DimensionType(double quantity, int kg, int m, int s) :
        quantity(quantity), kg(kg), m(m), s(s) {}

    /* Accessor methods. */
    double getQuantity() const
    {
        return quantity;
    }
    int getKilograms() const
    {
        return kg;
    }
    int getMeters() const
    {
        return m;
    }
    int getSeconds() const
    {
        return s;
    }
private:
    double quantity;
    int kg, m, s;
};

/* To add two DimensionTypes, we assert() that the types agree and then return the
 * proper value. Subtraction is defined similarly.
 */
const DimensionType operator+ (const DimensionType& one, const DimensionType& two)
{
    assert(one.getKilograms() == two.getKilograms());
    assert(one.getMeters() == two.getMeters());
    assert(one.getSeconds() == two.getSeconds());

    return DimensionType(one.getQuantity() + two.getQuantity(),
                          one.getKilograms(), one.getMeters(), one.getSeconds());
}

const DimensionType operator- (const DimensionType& one, const DimensionType& two)
{
    assert(one.getKilograms() == two.getKilograms());
    assert(one.getMeters() == two.getMeters());
    assert(one.getSeconds() == two.getSeconds());

    return DimensionType(one.getQuantity() - two.getQuantity(),
                          one.getKilograms(), one.getMeters(), one.getSeconds());
}

```

```

/* Multiplication and division are done by multiplying values and then updating the
 * units appropriately. Division is defined similarly.
 */
const DimensionType operator* (const DimensionType& one, const DimensionType& two)
{
    return DimensionType(one.getQuantity() * two.getQuantity(),
                          one.getKilograms() + two.getKilograms(),
                          one.getMeters() + two.getMeters(),
                          one.getSeconds() + two.getSeconds());
}

const DimensionType operator/ (const DimensionType& one, const DimensionType& two)
{
    return DimensionType(one.getQuantity() / two.getQuantity(),
                          one.getKilograms() - two.getKilograms(),
                          one.getMeters() - two.getMeters(),
                          one.getSeconds() - two.getSeconds());
}

```

There's a lot of code here, so let's take a few seconds to go over it. The `DimensionType` class exports a single constructor which takes as input a quantity along with the exponents of the units, then stores these values. It also defines four accessor methods to retrieve these values later. The implementation of `operator+` and `operator-` first assert that the units on the parameters agree, then perform the addition or subtraction. Finally, the implementation of `operator*` and `operator/` compute the new values and units, then return an appropriately-constructed `DimensionType`.

The main advantage of this solution over the previous one is that it allows us to have units of any values that we see fit. We can create a `DimensionType(5.0, 2, 5, -3)` to represent $5 \text{ kg}^2 \cdot \text{m}^5 / \text{s}^3$ just as easily as we can create a `DimensionType(5.0, 0, 0, 1)` to represent five seconds. However, this new `DimensionType` has several serious weaknesses. The most prominent error is that this implementation of `DimensionType` defers unit checking to runtime. For example, suppose that we write the following code:

```

DimensionType mass(5.0, 1, 0, 0);           // 5kg
DimensionType acceleration(9.8, 0, 1, -2);  // 9.8m/s²
DimensionType weight = mass + acceleration; // Whoops! Undefined!

```

The program will compile correctly, but will trigger an assert failure at runtime when we trying adding `mass` and `acceleration`. This is an improvement over using raw doubles to hold the values since we at least get a notification that there is a type error, but we have to actually execute the program to find the bug. If the incorrect code is in a rarely-used part of the program, we might not notice the error, and furthermore if we can't test the code in isolation (perhaps because it's part of a complex system) we have no way of detecting the error until it's too late.

The other major problem with this system is that we have no way of communicating the *expected* type of a value to the compiler. For example, suppose that we want to compute the weight of a block as follows:

```

DimensionType mass(5.0, 1, 0, 0);           // 5kg
DimensionType acceleration(9.8, 0, 1, -2);  // 9.8m·s⁻²
DimensionType weight = mass / acceleration; // Well-defined, but incorrect

```

Here, we've said that the weight of the block is the *quotient* of the mass and acceleration rather than their *product*. This is incorrect, but the above code will work compile and run without errors since all of the operations it uses are well-defined. The problem is that we haven't indicated anywhere that the `weight` variable should be in newtons. While we can circumvent this problem by adding more machinery into the mix, it's not worth the effort. This solution, while clever, is little better than using raw doubles. Again, we'll need to come up with a different approach.

Templates to the Rescue

The two approaches listed above have complementary strengths and weaknesses. Defining a multitude of types for each combination of units allows the compiler to confirm that all operations are well-defined, but requires us to write an infeasible amount of code. Writing a single `DimensionType` that encapsulates the units allows us to write only a few functions which manage the units, but offloads unit-checking until runtime. What if there was some way to hybridize this approach? That is, what if we could use the generic unit computations from `DimensionType` to create an assortment of types in the style of `kilogramT` and `meterT`?

There is indeed a way to do this, thanks to the C++ template system. As you've seen before, we can parameterize classes or functions over variable types – just think of `vector` or `map`. However, it is also possible to parameterize classes and functions over *integer values*. For example, the following declaration is perfectly legal:

```
template <typename T, int N> struct arrayT
{
    T array[N];
};
```

We could then use `arrayT` as follows:

```
arrayT<int, 137> myArray;           // Array of 137 ints
arrayT<double, 3> myOtherArray;    // Array of 3 doubles
```

Pause for a moment to absorb what implications integer template arguments have with regards to dimensional analysis. We want to create a theoretically unbounded number of types, each corresponding to some combination of units, with the mathematical operators well-defined over them. Moreover, we have a simple formula for how these types should be related based on what combination of units they store. This suggests the following solution. We'll create a single *template* class called `DimensionType` that's parameterized over the exponents of the kilograms, meters, and seconds units. Then, we'll define *template* implementations of the mathematical operators that perform all of the requisite operations. Before moving on to the actual code, let's quickly summarize the advantages of this approach:

- Because the units are hardcoded into the type of each `DimensionType` variable, the compiler can check the units in the computation at compile-time rather than runtime.
- Because the class and operators are templates, we only need to write a single implementation of the class and operators and the compiler will handle the rest. We don't have to worry about using units outside of some fixed bounds because the compiler will instantiate the template wherever it's required.

Implementing the new `DimensionType`

Now that we've familiarized ourselves with why this new approach is effective, let's begin writing the code for it. We'll start by defining the actual `DimensionType` class. We first write the template header:

```
template <int kg, int m, int s> class DimensionType
{
    /* ... */
};
```

Next, we'll write a constructor which accepts a `double` representing the quantity to store. We'll mark the constructor `explicit` so that we can't implicitly convert from `doubles` to `DimensionTypes`. This is primarily to prevent clients of `DimensionType` from accidentally bypassing the unit-checking system. For instance, if the constructor was not marked `explicit`, then given the following function prototype:


```
void DoSomething(const DimensionType<1, 1, -2>& newtons);
```

We could call the function by writing

```
DoSomething(137);
```

Since C++ would perform the implicit conversion to a `DimensionType<1, 1, -2>`. This means that we've just used a scalar value (137) where a value in newtons was expected. Marking the constructor `explicit` prevents this from accidentally occurring.

The constructor and the associated data members are shown here:

```
template <int kg, int m, int s> class DimensionType
{
public:
    explicit DimensionType(double amount) : quantity(amount) {}

private:
    double quantity;
};
```

We'll also provide an accessor member function `getQuantity` so that clients can access the amount of the quantity available:

```
template <int kg, int m, int s> class DimensionType
{
public:
    explicit DimensionType(double amount) : quantity(amount) {}

    double getQuantity() const
    {
        return quantity;
    }

private:
    double quantity;
};
```

This is the final working version of the `DimensionType` class. It's remarkably simple and looks surprisingly like the `kilogramT` type that we defined earlier in this chapter. Most of the interesting work will show up when we begin defining mathematical operators.

If you'll notice, we haven't defined a copy constructor or assignment operator for `DimensionType`. In our case, C++'s default implementation of these functions is perfectly fine for our purposes. Remember the Rule of Three: since we don't have a destructor, we probably don't need to implement either of the copy functions.

Implementing the Mathematical Operators

Now that we have the body of the `DimensionType` class ready and written, let's implement the mathematical operators. We'll begin by implementing `operator +` and `operator -`. Recall that it's only legal to add or subtract dimensioned values if the units agree. We'll therefore define template implementations of `operator+` and `operator-` that accept two `DimensionTypes` with the same units and produce new `DimensionTypes` appropriately. This is shown here:

```
template <int kg, int m, int s>
const DimensionType<kg, m, s> operator+ (const DimensionType<kg, m, s>& one,
                                         const DimensionType<kg, m, s>& two)
{
    return DimensionType<kg, m, s>(one.getQuantity() + two.getQuantity());
}
```

This code is dense, but most of what we've written is type declarations. The overall `operator+` function is templated over the units in the `DimensionType`, accepts as input two `DimensionTypes` of those particular units (by reference-to-const, of course) and returns a `const DimensionType` of the same units. The body of the function simply returns a `DimensionType` of the proper units initialized to the sum of the quantities of the parameters. To better see what's going on here, here's the same code, but with `DimensionType<kg, m, s>` short-handed to `DT`. This is *not* legal C++, but is illustrative of what's going on:

```
template <int kg, int m, int s>
const DT operator+ (const DT& one, const DT& two)
{
    return DT(one.getQuantity() + two.getQuantity());
}
```

Because C++ can automatically infer the arguments to template functions, we will never need to explicitly write out the types of the arguments to `operator+`. For example, here's some code for adding several different masses:

```
DimensionType<1, 0, 0> mass1(5.0);
DimensionType<1, 0, 0> mass2(10.0);
DimensionType<1, 0, 0> mass3 = mass1 + mass2; // Calls operator + <1, 0, 0>
```

Now, what happens if we try to add dimensioned types with different units? For example, what will happen if we compile the following code?

```
DimensionType<1, 0, -1> rate(5.0); // 5 kg/s
DimensionType<0, 1, -1> velocity(10.0); // 10 m/s
DimensionType<1, 0, -1> undefined = rate + velocity;
```

Fortunately, this code doesn't compile and we'll get an error directing us to the spot where we tried to add the `rate` and `velocity` variables. Because the `+` operator is declared such that both the arguments must have the same type, if we try adding `DimensionTypes` of different units, C++ won't be able to find an implementation of `operator+` and will raise a compiler error. Moreover, this error exists independently of the fact that we tried to assign the result to a variable of type `DimensionType<1, 0, -1>`. Simply trying to add the two variables is enough to cause the compiler to issue an error. We're moving toward a system for automatically checking units!

Addition and subtraction of `DimensionTypes` are almost identical. Here's an implementation of `operator-`, where the only change from `operator +` is that the function body performs a subtraction rather than an addition:

```
template <int kg, int m, int s>
const DimensionType<kg, m, s> operator- (const DimensionType<kg, m, s>& one,
                                         const DimensionType<kg, m, s>& two)
{
    return DimensionType<kg, m, s>(one.getQuantity() - two.getQuantity());
}
```

Now, let's move on to multiplication and division of `DimensionTypes`. Recall that any two dimensioned types can be multiplied (or divided), and the resulting type has units equal to the sum (or difference) of the dimensions of the input values.

When implementing `operator*` or `operator/`, we'll need to templatize the function over the units of both the `DimensionType` arguments. This, unfortunately, means that the function will be templatized over *six different values*: three integers for the units of the first parameter and three integers for the units of the second parameter. Fortunately, thanks to template function argument inference, we'll never need to explicitly write out these arguments as a client. As an implementer, though, we'll have to be a bit pedantic.

To define `operator*`, we'll take in two parameters – a `DimensionType<kg1, m1, s1>` and a `DimensionType<kg2, m2, s2>` and return a `DimensionType<kg1 + kg2, m1 + m2, s1 + s2>` with the appropriate quantity. Note that we've named the template arguments `kg1` and `kg2`, etc., so that the two `DimensionType` arguments can be of any units. Moreover, notice that the return type is templatized over an arithmetic expression of the template arguments. This is perfectly legal C++ and is one of the more powerful features of the template system.

Given the above description, the implementation of `operator*` is shown here:

```
template <int kg1, int m1, int s1, int kg2, int m2, int s2>
    const DimensionType<kg1 + kg2, m1 + m2, s1 + s2>
operator* (const DimensionType<kg1, m1, s1>& one,
          const DimensionType<kg2, m2, s2>& two)
{
    return DimensionType<kg1 + kg2, m1 + m2, s1 + s2>
        (one.getQuantity() * two.getQuantity());
}
```

Wow – that's quite a function signature! Again for simplicity, if we let **DT1** stand for the type of the first argument, **DT2** stand for the type of the second, and **DT3** stand for the return type, this reduces to the much simpler

```
/* Again, this is not legal C++ code but is illustrative of what's going on: */
template <int kg1, int m1, int s1, int kg2, int m2, int s2>
const DT3 operator* (const DT1& one, const DT2 two)
{
    return DT3(one.getQuantity() * two.getQuantity());
}
```

The implementation of `operator/` is similar to that of `operator*`, except that we subtract units instead of adding them and divide the quantities instead of multiplying them:

```
template <int kg1, int m1, int s1, int kg2, int m2, int s2>
    const DimensionType<kg1 - kg2, m1 - m2, s1 - s2>
operator/ (const DimensionType<kg1, m1, s1>& one,
          const DimensionType<kg2, m2, s2>& two)
{
    return DimensionType<kg1 - kg2, m1 - m2, s1 - s2>
        (one.getQuantity() / two.getQuantity());
}
```

Now that we've defined all four of the basic mathematical operators, we can write code using `DimensionType`. C++'s ability to automatically infer template arguments is astounding here and the compiler will automatically figure out the types of all of the expressions in this code and will notice that there is no unit error:

```

/* Physics question: We have a 5kg block sitting on Earth where gravity accelerates
 * objects at 9.8m/s2. We wish to move the block vertically at a constant rate of
 * 10m/s. How much power, in watts (kg·m2/s3) does this require?
 *
 * The answer is the product of the weight of the block and the velocity at which
 * we're lifting. In code:
 */

DimensionType<1, 0, 0> mass(5);          // 5kg
DimensionType<0, 1, -2> gravity(9.8);    // 9.8m/s2
DimensionType<0, 1, -1> velocity(10);    // 10m/s
DimensionType<1, 2, -3> power = mass * gravity * velocity;

```

If this doesn't strike you as remarkable, pause for a minute and think about what's going on here. If we change even the slightest detail of this code, the compiler will recognize the unit error and give us an error. For example, suppose that we accidentally use kilograms per second instead of meters per second for the velocity at which we want to raise the block. Then we'd get an error trying to assign `mass * gravity * velocity` to `power` since the type of `mass * gravity * power` would be `DimensionType<2, 1, -3>` instead of the expected `DimensionType<1, 2, -3>`. Similarly, if we accidentally divided by the velocity, we'd get a result of type `DimensionType<1, 0, -1>`, which couldn't be stored in the `power` variable.

typedef to the Rescue

The above system works remarkably well, but is a bit clunky at times. For example, to work with variables in units of kilograms, we have to use a `DimensionType<1, 0, 0>` and to work with newtons must type out `DimensionType<1, 1, -2>`.

We can greatly simplify use of the `DimensionType` class by using `typedef` to create shorthand names for the longer types. For example, if we do the following:

```
typedef DimensionType<1, 0, 0> kilogramT;
```

We can then use `kilogramT` to refer to a quantity in kilograms. We can do a similar trick for variables in newtons. In fact, we might want to use all of the following `typedefs` to simplify our work:

```

typedef DimensionType<1, 0, 0>   kilogramT;
typedef DimensionType<0, 1, 0>   meterT;
typedef DimensionType<0, 0, 1>   secondT;
typedef DimensionType<1, 1, -2>  newtonT;
typedef DimensionType<1, 2, -2>  jouleT;
typedef DimensionType<1, 2, -3>  wattT;
typedef DimesnionType<0, 0, 0>   scalarT;

```

Earlier in this chapter we had to define all of these classes manually, but thanks to templates the compiler can now automatically generate them for us. Isn't that simpler?

More to Explore

The `DimensionType` class we've constructed here is marvelously useful and effectively solves the problem we set out to solve. However, it is far from complete and there are many other operations we might want to define on `DimensionType`. These are left as an exercise to the reader:

1. **Compound assignment operators.** The current implementation of `DimensionType` does not have support for `+=`, `-=`, `*=`, or `/=`. Add these operators to `DimensionType`, making sure that the operation is always well-defined. For example, does it make sense to write `mass /= otherMass`? How about `mass *= 15`?
2. **Scalar operations.** It is always legal to multiply a dimensioned type by a scalar, as in `10·5kg`. With the current implementation of `DimensionType`, it is illegal to write code such as `10 * mass` since `operator*` requires both parameters to be `DimensionTypes` and `10` is not a `DimensionType`. Implement versions of `operator*` and `operator/` such that it is legal to multiply or divide `DimensionTypes` by scalars.
3. **Stream insertion operators.** Implement a stream insertion operator for `DimensionType` that prints out the quantity and a human-readable representation of the units. For example, printing a `DimensionType<1, 0, 0>` with value `137` might print `137kg`.
4. **Template quantity types.** The `DimensionType` class we implemented in this example always stores its quantities as `doubles`. However, we may want to store quantities in other forms, such as complex or rational numbers. Modify the above code so that `DimensionType` is parameterized over an additional argument representing what type to use as a quantity.
5. **Advanced unit checking.** The `DimensionType` we've developed uses metric units for all of its values, but some programs might want to use imperial units (pounds, feet, etc.) instead. Mixing and matching these units without multiplying by the correct conversion factor can result in spectacular failures. One prominent example was the failed Mars Climate Orbiter mission, which was supposed to enter Martian orbit to study the planet's atmosphere. Unfortunately, the mission failed, and the official report concluded that a mismatch between metric and imperial units was to blame:

[I]t was discovered that the small forces ΔV 's reported by the spacecraft engineers for use in orbit determination solutions was low by a factor of 4.45 (1 pound force=4.45 Newtons) because the impulse bit data contained in the AMD file was delivered in lb-sec instead of the specified and expected units of Newton-sec. [MCO]

Devise a way of representing multiple systems of units in a `DimensionType` and safely converting between quantities expressed in two unit systems.

Chapter 25: Extended Example: `grid`

The STL encompasses a wide selection of associative and sequence containers. However, one useful data type that did not find its way into the STL is a multidimensional array class akin to the CS106B/X `Grid`. In this extended example, we will implement an STL-friendly version of the CS106B/X `Grid` class, which we'll call `grid`, that will support STL-compatible iterators, intuitive element-access syntax, and relational operators. Once we're done, we'll have an industrial-strength container class we will use later in this book to implement more complex examples.

Implementing a fully-functional `grid` may seem daunting at first, but fortunately it's easy to break the work up into several smaller steps that culminate in a working class.

Step 0: Implement the Basic `grid` Class.

Before diving into some of the `grid`'s more advanced features, we'll begin by implementing the `grid` basics. Below is a partial specification of the `grid` class that provides core functionality:

Figure 0: Basic (incomplete) interface for the `grid` class

```
template <typename ElemType> class grid {
public:
    /* Constructors, destructors. */
    grid(); // Create empty grid
    grid(int rows, int cols); // Construct to specified size

    /* Resizing operations. */
    void clear(); // Empty the grid
    void resize(int rows, int cols); // Resize the grid

    /* Query operations. */
    int numRows() const; // Returns number of rows in the grid
    int numCols() const; // Returns number of columns in the grid
    bool empty() const; // Returns whether the grid is empty
    int size() const; // Returns the number of elements

    /* Element access. */
    ElemType& getAt(int row, int col); // Access individual elements
    const ElemType& getAt(int row, int col) const; // Same, but const
};
```

These functions are defined in greater detail here:

<code>grid();</code>	Constructs a new, empty <code>grid</code> .
<code>grid(int rows, int cols);</code>	Constructs a new <code>grid</code> with the specified number of rows and columns. Each element in the <code>grid</code> is initialized to its default value. In true STL style, we won't do any error checking to see if the dimensions are negative.
<code>void clear();</code>	Resizes the <code>grid</code> to 0x0.

Basic grid functionality, contd.

<code>void resize(int rows, int cols);</code>	Discards the current contents of the <code>grid</code> and resizes the <code>grid</code> to the specified size. Each element in the <code>grid</code> is initialized to its default value. As with the constructor, we won't worry about the case where the dimensions are negative.
<code>int numRows() const;</code> <code>int numCols() const;</code>	Returns the number of rows and columns in the <code>grid</code> .
<code>bool empty() const;</code>	Returns whether the <code>grid</code> contains no elements. This is true if either the number of rows or columns is zero.
<code>int size() const;</code>	Returns the number of total elements in the <code>grid</code> .
<code>ElemType& getAt(int row, int col);</code> <code>const ElemType& getAt(int row, int col) const;</code>	Returns a reference to the element at the specified position. This function is <code>const</code> -overloaded. We won't worry about the case where the indices are out of bounds.

Because `grid`s can be dynamically resized, we will need to back `grid` with some sort of dynamic memory management. Because the `grid` represents a two-dimensional entity, you might think that we need to use a dynamically-allocated multidimensional array to store `grid` elements. However, working with dynamically-allocated multidimensional arrays is tricky and greatly complicates the implementation. Fortunately, we can sidestep this problem by implementing the two-dimensional `grid` object using a single-dimensional array. To see how this works, consider the following 3x3 `grid`:

0	1	2
3	4	5
6	7	8

We can represent all of the elements in this `grid` using a one-dimensional array by laying out all of the elements sequentially, as seen here:

0	1	2	3	4	5	6	7	8

If you'll notice, in this ordering, the three elements of the first row appear in order as the first three elements, then the three elements of the second row in order, and finally the three elements of the final row in order. Because this one-dimensional representation of a two-dimensional object preserves the ordering of individual rows, it is sometimes referred to as *row-major order*.

To represent a `grid` in row-major order, we need to be able to convert between `grid` coordinates and array indices. Given a coordinate (*row*, *col*) in a `grid` of dimensions (*nrows*, *ncols*), the corresponding position in the row-major order representation of that `grid` is given by $index = col + row * ncols$. The intuition behind this formula is that because the ordering within any row is preserved, each horizontal step in the `grid` translates into a single step forward or backward in the row-major order representation of the `grid`. However, each vertical step in the `grid` requires us to advance forward to the next row in the linearized `grid`, skipping over *ncols* elements.

Using row-major order, we can back the `grid` class with a regular STL `vector`, as shown here:


```

template <typename ElemType> class grid {
public:
    grid();
    grid(int rows, int cols);

    void clear();
    void resize(int rows, int cols);

    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;
private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

```

Serendipitously, implementing the `grid` with a `vector` allows us to use C++'s automatically-generated copy constructor and assignment operator for `grid`. Since `vector` already manages its own memory, we don't need to handle it manually.

Note that we explicitly keep track of the number of rows and columns in the `grid` even though the `vector` stores the total number of elements. This is necessary so that we can compute indices in the row-major ordering for points in two-dimensional space.

The above functions have relatively straightforward implementations that are given below:

```

template <typename ElemType> grid::grid() : rows(0), cols(0)
{
}

template <typename ElemType>
grid::grid(int rows, int cols) : elems(rows * cols), rows(rows), cols(cols)
{
}

template <typename ElemType> void grid::clear()
{
    elems.clear();
}

template <typename ElemType> void grid::resize(int rows, int cols)
{
    /* See below for assign */
    elems.assign(rows * cols, ElemType());

    /* Explicit this-> required since parameters have same name as members. */
    this->rows = rows;
    this->cols = cols;
}

```

```

template <typename ElemType> int grid::numRows() const
{
    return rows;
}

template <typename ElemType> int grid::numCols() const
{
    return cols;
}

template <typename ElemType> bool grid::empty() const
{
    return size() == 0;
}

template <typename ElemType> int grid::size() const
{
    return rows * cols;
}

/* Use row-major ordering to access the proper element of the vector. */
template <typename ElemType> ElemType& grid::getAt(int row, int col)
{
    return elems[col + row * cols];
}
template <typename ElemType> const ElemType& grid::getAt(int row, int col) const
{
    return elems[col + row * cols];
}

```

Most of these functions are one-liners and are explained in the comments. The only function that you may find interesting is `resize`, which uses the vector's `assign` member function. `assign` is similar to `resize` in that it changes the size of the vector, but unlike `resize` `assign` discards all of the current vector contents and replaces them with the specified number of copies of the specified element. The use of `ElemType()` as the second parameter to `assign` means that we will fill the vector with copies of the default value of the type being stored (since `ElemType()` uses the temporary object syntax to create a new `ElemType`).

Step 1: Add Support for Iterators

Now that we have the basics of a `grid` class, it's time to add iterator support. This will allow us to plug the `grid` directly into the STL algorithms and will be invaluable in a later chapter.

Like the `map` and `set`, the `grid` does not naturally lend itself to a linear traversal – after all, `grid` is two-dimensional – and so we must arbitrarily choose an order in which to visit elements. Since we've implemented the `grid` in row-major order, we'll have `grid` iterators traverse the grid row-by-row, top to bottom, from left to right. Thus, given a 3x4 `grid`, the order of the traversal would be

0	1	2
3	4	5
6	7	8
9	10	11

This order of iteration maps naturally onto the row-major ordering we've chosen for the `grid`. If we consider how the above grid would be laid out in row-major order, the resulting array would look like this:

0	1	2	3	4	5	6	7	8	9	10	11

Thus this iteration scheme maps to a simple linear traversal of the underlying representation of the `grid`. Because we've chosen to represent the elements of the `grid` using a `vector`, we can iterate over the elements of the `grid` using `vector` iterators. We thus add the following definitions to the `grid` class:

```
template <typename ElemType> class grid {
public:
    grid();
    grid(int rows, int cols);

    void clear();
    void resize(int rows, int cols);

    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;

    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;
private:
    vector<ElemType> elems;
    int rows;
    int cols;
};
```

Now, clients of `grid` can create `grid<int>::iterators` rather than `vector<int>::iterators`. This makes the interface more intuitive and increases encapsulation; since `iterator` is a `typedef`, if we later decide to replace the underlying representation with a dynamically-allocated array, we can change the `typedefs` to

```
typedef ElemType* iterator;
typedef const ElemType* const_iterator;
```

And clients of the `grid` will not notice any difference.

Notice that in the above `typedefs` we had to use the `typename` keyword to name the type `vector<ElemType>::iterator`. This is the pesky edge case mentioned in the chapter on templates and somehow manages to creep into more than its fair share of code. Since `iterator` is a nested type inside the template type `vector<ElemType>`, we have to use the `typename` keyword to indicate that `iterator` is the name of a type rather than a class constant.

We've now defined an `iterator` type for our `grid`, so what functions should we export to the `grid` clients? We'll at least want to provide support for `begin` and `end`, as shown here:

```

template <typename ElemType> class grid {
public:
    grid();
    grid(int rows, int cols);

    void clear();
    void resize(int rows, int cols);

    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;

    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

```

We've provided two versions of each function so that clients of a `const` grid can still use iterators. These functions are easily implemented by returning the value of the underlying vector's `begin` and `end` functions, as shown here:

```

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::begin()
{
    return elems.begin();
}

```

Notice that the return type of this function is `typename grid<ElemType>::iterator` rather than just `iterator`. Because `iterator` is a nested type inside `grid`, we need to use `grid<ElemType>::iterator` to specify which iterator we want, and since `grid` is a template type we have to use the `typename` keyword to indicate that `iterator` is a nested type. Otherwise, this function should be straightforward.

The rest of the functions are implemented here:

```

template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::begin() const
{
    return elems.begin();
}

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::end()
{
    return elems.end();
}

```

```
template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::end() const
{
    return elems.end();
}
```

Because the `grid` is implemented in row-major order, elements of a single row occupy consecutive locations in the vector. It's therefore possible to return iterators delineating the start and end of each row in the `grid`. This is useful functionality, so we'll provide it to clients of the `grid` through a pair of member functions `row_begin` and `row_end` (plus `const` overloads). These functions are declared here:

```
template <typename ElemType> class grid {
public:
    grid();
    grid(int rows, int cols);

    void clear();
    void resize(int rows, int cols);

    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;

    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    iterator row_begin(int row);
const_iterator row_begin(int row) const;
iterator row_end(int row);
const_iterator row_end(int row) const;

private:
    vector<ElemType> elems;
    int rows;
    int cols;
};
```

Before implementing these functions, let's take a minute to figure out exactly where the iterations we return should point to. Recall that the element at position $(row, 0)$ in a grid of size $(rows, cols)$ can be found at position $row * cols$. We should therefore have `row_begin(row)` return an iterator to the $row * cols$ element of the vector. Since there are $cols$ elements in each row and `row_end` should return an iterator to one position past the end of the row, this function should return an iterator to the position $cols$ past the location returned by `row_begin`. Given this information, we can implement these functions as shown here:

```

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::row_begin(int row)
{
    return begin() + cols * row;
}

template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::row_begin(int row) const
{
    return begin() + cols * row;
}

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::row_end(int row)
{
    return row_begin(row) + cols;
}

template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::row_end(int row) const
{
    return row_begin(row) + cols;
}

```

We now have an elegant `iterator` interface for the `grid` class. We can iterate over the entire container as a whole, just one row at a time, or some combination thereof. This enables us to interface the `grid` with the STL algorithms. For example, to zero out a `grid<int>`, we can use the `fill` algorithm, as shown here:

```
fill(myGrid.begin(), myGrid.end(), 0);
```

We can also sort the elements of a row using `sort`:

```
sort(myGrid.row_begin(0), myGrid.row_end(0));
```

With only a handful of functions we're now capable of plugging directly into the full power of the algorithms. This is part of the beauty of the STL – had the algorithms been designed to work on containers rather than iterators or ranges, this would not have been possible.

Step 2: Add Support for the Element Selection Operator

When using regular C++ multidimensional arrays, we can write code that looks like this:

```

int myArray[137][42];
myArray[2][4] = 271828;
myArray[9][0] = 314159;

```

However, with the current specification of the `grid` class, the above code would be illegal if we replaced the multidimensional array with a `grid<int>`, since we haven't provided an implementation of `operator []`.

Adding support for element selection to linear classes like the `vector` is simple – we simply have the brackets operator return a reference to the proper element. Unfortunately, it is much trickier to design `grid` such that the bracket syntax works correctly. The reason is that if we write code that looks like this:

```

grid<int> myGrid(137, 42);
int value = myGrid[2][4];

```

By replacing the bracket syntax with calls to `operator []`, we see that this code expands out to

```
grid<int> myGrid(137, 42);
int value = (myGrid.operator[] (2)).operator[] (4);
```

Here, there are *two* calls to `operator []`, one invoked on `myGrid` and the other on the value returned by `myGrid.operator[] (2)`. To make the above code compile, the object returned by the grid's `operator[]` must *itself* define an `operator []` function. It is this returned object, rather than the grid itself, which is responsible for retrieving the requested element from the grid. Since this temporary object is used to perform a task normally reserved for the grid, it is sometimes known as a *proxy object*.

How can we implement the grid's `operator []` so that it works as described above? First, we will need to define a new class representing the object returned by the grid's `operator []`. In this discussion, we'll call it `MutableReference`, since it represents an object that can call back into the grid and mutate it. For simplicity and to maximize encapsulation, we'll define `MutableReference` inside of `grid`. This results in the following interface for `grid`:

```
template <typename ElemType> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference
    {
    public:
        friend class grid;
        ElemType& operator[] (int col);
    private:
        MutableReference(grid* owner, int row);

        grid* const owner;
        const int row;
    };

private:
    vector<ElemType> elems;
    int rows;
    int cols;
};
```

The `MutableReference` object stores some a pointer to the grid that created it, along with the index passed in to the grid's `operator []` function when the `MutableReference` was created. That way, when we invoke the `MutableReference`'s `operator []` function specifying the *col* coordinate of the grid, we can pair it with the stored *row* coordinate, then query the grid for the element at (*row*, *col*). We have also made `grid` a friend of `MutableReference` so that the grid can call the private constructor necessary to initialize a `MutableReference`.

We can implement `MutableReference` as follows:

```

template <typename ElemType>
grid<ElemType>::MutableReference::MutableReference(grid* owner, int row) :
    owner(owner), row(row)
{
}

template <typename ElemType>
ElemType& grid<ElemType>::MutableReference::operator[] (int col)
{
    return owner->getAt(row, col);
}

```

Notice that because `MutableReference` is a nested class inside `grid`, the implementation of the `MutableReference` functions is prefaced with `grid<ElemType>::MutableReference` instead of just `MutableReference`. However, in this particular case the pesky `typename` keyword is not necessary because we are prototyping a function inside `MutableReference` rather than using the type `MutableReference` in an expression.

Now that we've implemented `MutableReference`, we'll define an `operator []` function for the `grid` class that constructs and returns a properly-initialized `MutableReference`. This function accepts an *row* coordinate, and returns a `MutableReference` storing that row number and a pointer back to the `grid`. That way, if we write

```
int value = myGrid[1][2];
```

The following sequences of actions occurs:

1. `myGrid.operator[]` is invoked with the parameter 1.
2. `myGrid.operator[]` creates a `MutableReference` storing the *row* coordinate 1 and a means for communicating back with the `myGrid` object.
3. `myGrid.operator[]` returns this `MutableReference`.
4. The returned `MutableReference` then has its `operator[]` function called with parameter 2.
5. The returned `MutableReference` then calls back to the `myGrid` object and asks for the element at position (1, 2).

This sequence of actions is admittedly complex, but is transparent to the client of the `grid` class and runs efficiently.

`operator[]` is defined and implemented as follows:


```

template <typename ElemType> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference
    {
    public:
        friend class grid;
        ElemType& operator[] (int col);
    private:
        MutableReference(grid* owner, int row);

        grid* const owner;
        const int row;
    };
    MutableReference operator[] (int row);

private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

template <typename ElemType>
typename grid<ElemType>::MutableReference grid<ElemType>::operator[] (int row)
{
    return MutableReference(this, row);
}

```

Notice that we've only provided an implementation of the non-const version of `operator[]`. But what if we want to use `operator[]` on a `const grid`? We would similarly need to return a proxy object, but that object would need to guarantee that `grid` clients could not write code like this:

```

const grid<int> myGrid(137, 42);
myGrid[0][0] = 2718; // Oops! Modified const object!

```

To prevent this sort of problem, we'll have the `const` version of `operator[]` return a proxy object of a different type, called `ImmutableReference` which behaves similarly to `MutableReference` but which returns `const` references to the elements in the `grid`. This results in the following interface for `grid`:

```

template <typename ElemType> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference
    {
    public:
        friend class grid;
        ElemType& operator[] (int col);
    private:
        MutableReference(grid* owner, int row);

        grid* const owner;
        const int row;
    };
    MutableReference operator[] (int row);

    class ImmutableReference
    {
    public:
        friend class grid;
        const ElemType& operator[] (int col) const;
    private:
        MutableReference(const grid* owner, int row);

        const grid* const owner;
        const int row;
    };
    ImmutableReference operator[] (int row) const;

private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

```

ImmutableReference and the const version of operator[] are similar to MutableReference and the non-const version of operator[], and to save space we won't write it here. The complete listing of the grid class at the end of this chapter contains the implementation if you're interested.

Step 3: Define Relational Operators

Now that our grid has full support for iterators and a nice bracket syntax that lets us access individual elements, it's time to put on the finishing touches. As a final step in the project, we'll provide implementations of the relational operators for our grid class. We begin by updating the grid interface to include the following functions:

```

template <typename ElemType> class grid {
public:
    /* ... previously-defined functions ... */

    bool operator < (const grid& other) const;
    bool operator <= (const grid& other) const;
    bool operator == (const grid& other) const;
    bool operator != (const grid& other) const;
    bool operator >= (const grid& other) const;
    bool operator > (const grid& other) const;

private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

```

Note that of the six operators listed above, only the `==` and `!=` operators have intuitive meanings when applied to grids. However, it also makes sense to define a `<` operator over grids so that we can store them in STL `map` and `set` containers, and to ensure consistency, we should define the other three operators as well.

Because there is no natural interpretation for what it means for one `grid` to be “less than” another, we are free to implement these functions in any way that we see fit, provided that the following invariants hold for any two grids `one` and `two`:

1. `one == two` if and only if the two grids have the same dimensions and each element compares identical by the `==` operator.
2. All of the mathematical properties of `<`, `<=`, `==`, `!=`, `>=`, and `>` hold. For example if `one < two`, `!(one >= two)` should be true and `one <= two` should also hold. Similarly, `one == one`, and if `one == two`, `two == one`.
3. For any `one` and `two`, exactly one of `one == two`, `one < two`, or `one > two` should be true.

As mentioned in the section on operator overloading, it is possible to implement all six of the relational operators in terms of the less-than operator. One strategy for implementing the relational operators is thus to implement just the less-than operator and then to define the other five as wrapped calls to `operator <`. But what is the best way to determine whether one `grid` compares less than another? One general approach is to define a *lexicographical ordering* over grids. We will compare each field one at a time, checking to see if the fields are equal. If so, we move on to the next field. Otherwise, we immediately return that one `grid` is less than another without looking at the remaining fields. If we go through every field and find that the grids are equal, then we can return that neither grid is less than the other. This is similar to the way that we might order words alphabetically – we find the first mismatched character, then return which word compares first. We can begin by implementing `operator <` as follows:

```

template <typename ElemType>
bool grid<ElemType>::operator < (const grid& other) const
{
    /* Compare the number of rows and return immediately if there's a mismatch. */
    if(rows != other.rows)
        return rows < other.rows;

    /* Next compare the number of columns the same way. */
    if(cols != other.cols)
        return cols < other.cols;

    /* ... */
}

```

Here, we compare the `rows` fields of the two objects and immediately return if they aren't equal. We can then check the `cols` fields in the same way. Finally, if the two `grid`s have the same number of rows and columns, we need to check how the elements of the `grid`s compare. Fortunately, this is straightforward thanks to the STL `lexicographical_compare` algorithm. `lexicographical_compare` accepts four iterators delineating two ranges, then lexicographically compares the elements in those ranges and returns if the first range compares lexicographically less than the second. Using `lexicographical_compare`, we can finish our implementation of `operator <` as follows:

```
template <typename ElemType>
bool grid<ElemType>::operator < (const grid& other) const
{
    /* Compare the number of rows and return immediately if there's a mismatch. */
    if(rows != other.rows)
        return rows < other.rows;

    /* Next compare the number of columns the same way. */
    if(cols != other.cols)
        return cols < other.cols;

    return lexicographical_compare(begin(), end(), other.begin(), other.end());
}
```

All that's left to do now is to implement the other five relational operators in terms of `operator <`. This is done below:

```
template <typename ElemType>
bool grid<ElemType>::operator >=(const grid& other) const
{
    return !(*this < other);
}

template <typename ElemType>
bool grid<ElemType>::operator ==(const grid& other) const
{
    return !(*this < other) && !(other < *this);
}

template <typename ElemType>
bool grid<ElemType>::operator !=(const grid& other) const
{
    return (*this < other) || (other < *this);
}

template <typename ElemType>
bool grid<ElemType>::operator >(const grid& other) const
{
    return other < *this;
}

template <typename ElemType>
bool grid<ElemType>::operator <=(const grid& other) const
{
    return !(other < *this);
}
```

At this point we're done! We now have a complete working implementation of the `grid` class that supports iteration, element access, and the relational operators. To boot, it's implemented on top of the `vector`, meaning that it's slick and efficient. This class should be your one-stop solution for applications that require a two-dimensional array.

More to Explore

While the `grid` class we've written is useful in a wide variety of circumstances, there are several extra features you may want to consider adding to `grid`. Here are some suggestions to get you started:

1. **Column iterators:** The iterators used in this `grid` class are useful for iterating across rows but not columns. Consider defining a custom class `column_iterator` that can iterate over `grid` and columns using iterator syntax. Make sure to provide `const` overloads!
2. **Smart resizing:** Right now, the only way to resize a `grid` is by discarding the old contents entirely. Wouldn't it be nice if there was a way to resize the `grid` by inserting or deleting rows and columns in the existing structure? This is not particularly difficult to implement, but is a nice feature to add.
3. **Subgrid functions:** Just as the C++ `string` exports a `substr` function to get a substring from that string, it might be useful to define a function that gets a “subgrid” out of the `grid`.
4. **Templatized Conversion Functions:** When working with two `grid`s of different types that are implicitly convertible to one another (say, a `grid<int>` and a `grid<double>`), it might be useful to allow expressions like `myDoubleGrid = myIntGrid`. Consider adding a templatized conversion constructor to the `grid` class to make these assignments legal.

If you're interested in some client-side applications of the `grid`, consider the following problems:

1. A *magic square* is a square array of numbers such that the sum of every row, column, and diagonal is the same. For example, the following 3x3 `grid` is a magic square where each row, column, and diagonal sums to 15:

4	9	2
3	5	7
8	1	6

This 5x5 `grid` is also a magic square where each row, column, and diagonal sums to 65:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Using the `next_permutation` algorithm and the `grid` class, write a function `ListAllMagicSquares` that accepts as input the size of a magic square, then prints out all magic squares of that size.

2. *Minesweeper* is a logic puzzle where the user is presented a two-dimensional grid containing a collection of mines. Initially every square in the grid is hidden, and the user may reveal squares to determine their contents. If the user reveals a mine, she loses the game. Otherwise, if the tile does not contain a mine, then the tile contains a number indicating the number of adjacent mines. The user wins if she reveals all tiles that do not contain mines. Write an implementation of Minesweeper using the grid to store the underlying board.

Complete grid Listing

```
template <typename ElemType> class grid
{
public:
    /* Constructors either create an empty grid or a grid of the
     * specified size.
     */
    grid();
    grid(int numRows, int numCols);

    /* Resizing functions. */
    void clear();
    void resize(int width, int height);

    /* Size queries. */
    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    /* Element access. */
    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;

    /* Iterator definitions. */
    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;

    /* Container iteration. */
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    /* Row iteration. */
    iterator row_begin(int row);
    iterator row_end(int row);
    const_iterator row_begin(int row) const;
    const_iterator row_end(int row) const;
```

```

/* Proxy objects for operator[] */
class MutableReference
{
public:
    friend class grid;
    ElemType& operator[](int col);
private:
    MutableReference(grid* source, int row);
    grid * ref;
    int row;
};

class ImmutableReference
{
public:
    friend class grid;
    const ElemType& operator[](int col) const;
private:
    ImmutableReference(const grid* source, int row);
    const grid * ref;
    int row;
};

/* Element selection functions. */
MutableReference operator[](int row);
ImmutableReference operator[](int row) const;

/* Relational operators. */
bool operator < (const grid& other) const;
bool operator <= (const grid& other) const;
bool operator == (const grid& other) const;
bool operator != (const grid& other) const;
bool operator >= (const grid& other) const;
bool operator > (const grid& other) const;
private:
    vector<ElemType> elems;
    int rows;
    int cols;
};

/* Constructs a new, empty grid. */
template <typename ElemType> grid<ElemType>::grid() : rows(0), cols(0)
{
}

/* Constructs a grid of the specified size. */
template <typename ElemType>
grid<ElemType>::grid(int rows, int cols) :
    elems(rows * cols), rows(rows), cols(cols)
{
}

/* Empties the grid. */
template <typename ElemType> void grid<ElemType>::clear()
{
    elems.clear();
}

```

```

/* Discards all existing content and redimensions the grid. */
template <typename ElemType> void grid<ElemType>::resize(int rows, int cols)
{
    elems.assign(rows * cols, ElemType());
    this->rows = rows;
    this->cols = cols;
}

/* Size query functions. */
template <typename ElemType> int grid<ElemType>::numRows() const
{
    return rows;
}
template <typename ElemType> int grid<ElemType>::numCols() const
{
    return cols;
}

/* The grid is empty if the size is zero. */
template <typename ElemType> bool grid<ElemType>::empty() const
{
    return size() == 0;
}

/* The size of the grid is the product of the number of rows and columns. */
template <typename ElemType> int grid<ElemType>::size() const
{
    return rows * cols;
}

/* Accessor member functions use the row-major order index conversions to access
 * elements.
 */
template <typename ElemType> ElemType& grid<ElemType>::getAt(int row, int col)
{
    return elems[col + row * cols];
}
template <typename ElemType>
const ElemType& grid<ElemType>::getAt(int row, int col) const
{
    return elems[col + row * cols];
}

/* Iterator support, including const overloads. */
template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::begin()
{
    return elems.begin();
}
template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::begin() const
{
    return elems.begin();
}

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::end()
{
    return elems.end();
}

```



```

template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::end() const
{
    return elems.end();
}

/* Row iteration uses the row-major order formula to select the proper elements. */
template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::row_begin(int row)
{
    return elems.begin() + row * cols;
}
template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::row_begin(int row) const
{
    return elems.begin() + row * cols;
}

template <typename ElemType>
typename grid<ElemType>::iterator grid<ElemType>::row_end(int row)
{
    return row_begin(row) + cols;
}
template <typename ElemType>
typename grid<ElemType>::const_iterator grid<ElemType>::row_end(int row) const
{
    return row_begin(row) + cols;
}

/* Implementation of the MutableReference and ImmutableReference classes. */
template <typename ElemType>
grid<ElemType>::MutableReference::MutableReference(grid* source, int row) :
    ref(source), row(row)
{
}

template <typename ElemType>
grid<ElemType>::ImmutableReference::ImmutableReference(const grid* source, int row)
    : ref(source), row(row)
{
}

/* operator[] calls back into the original grid. */
template <typename ElemType>
ElemType& grid<ElemType>::MutableReference::operator [] (int col)
{
    return ref->getAt(row, col);
}
template <typename ElemType>
const ElemType& grid<ElemType>::ImmutableReference::operator [] (int col) const
{
    return ref->getAt(row, col);
}

/* operator[] implementations create and return (Im)MutableReferences. */
template <typename ElemType>
typename grid<ElemType>::MutableReference grid<ElemType>::operator [] (int row)
{
    return MutableReference(this, row);
}

```

```

template <typename ElemType>
typename grid<ElemType>::ImmutableReference
    grid<ElemType>::operator[](int row) const
{
    return ImmutableReference(this, row);
}

/* operator< performs a lexicographical comparison of two grids. */
template <typename ElemType>
bool grid<ElemType>::operator <(const grid& other) const
{
    /* Lexicographically compare by dimensions. */
    if(rows != other.rows)
        return rows < other.rows;
    if(cols != other.cols)
        return cols < other.cols;

    /* Perform a lexicographical comparison. */
    return lexicographical_compare(begin(), end(), other.begin(), other.end());
}

/* Other relational operators defined in terms of operator<. */
template <typename ElemType>
bool grid<ElemType>::operator >=(const grid& other) const
{
    return !(*this < other);
}

template <typename ElemType>
bool grid<ElemType>::operator ==(const grid& other) const
{
    return !(*this < other) && !(other < *this);
}

template <typename ElemType>
bool grid<ElemType>::operator !=(const grid& other) const
{
    return (*this < other) || (other < *this);
}

template <typename ElemType>
bool grid<ElemType>::operator >(const grid& other) const
{
    return other < *this;
}

template <typename ElemType>
bool grid<ElemType>::operator <=(const grid& other) const
{
    return !(other < *this);
}

```

Chapter 26: Functors

Consider a simple task. Suppose you have a `vector<string>` you'd like to count the number of strings that have length less than five. You stumble upon the STL `count_if` algorithm, which accepts a range of iterators and a predicate function, then returns the number of elements in the range for which the function returns true. Since we want to count the number of strings with length less than five, we could write a function like this one:

```
bool LengthIsLessThanFive(const string& str)
{
    return str.length() < 5;
}
```

And then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanFive)` to get the number of short elements. Similarly, if you wanted to count the number of strings with length less than ten, we could write a `LengthIsLessThanTen` function, and so on and so forth.

The above approach is legal C++, but is not particularly elegant. Instead of writing multiple functions to compare string lengths against constants, good programming practice suggests that we should just write *one* function that looks like this:

```
bool LengthIsLessThan(const string& str, int length)
{
    return str.length() < length;
}
```

This way, we can specify the maximum length as the second parameter.

While this new function is more generic than the previous version, unfortunately we can't use it in conjunction with `count_if`. `count_if` requires a *unary* function (a function taking only one argument) as its final parameter, and the new `LengthIsLessThan` is a *binary* function. This new function, while more generally applicable than the original version, is actually less useful. There must be some way to compromise between the two approaches. We need a way to construct a function that takes in only one parameter (the string to test), but which can be customized to accept an arbitrary maximum length. How can we do this?

This problem boils down to a question of data flow. To construct this hybrid function, we need to somehow communicate the upper bound into the function so that it can perform the comparison. Now, a raw C++ function can access the following information:

- Its local variables.
- Its parameters.
- Global variables.

Is there some way that we can store the maximum length of the string in one of these locations? We can't store it in a local variable, since local variables don't persist between function calls. As mentioned above, we can't store it in a parameter, since `count_if` is hardcoded to accept a unary function. That leaves global variables. We *could* solve this problem using global variables: we store the maximum length in a global variable, then compare the string parameter length against the global. For example:

```
int gMaxLength; // Value to compare against

bool LengthIsLessThan(const string& str)
{
    return str.length() < gMaxLength;
}
```

This approach works: if our `vector<string>` is called `v`, then we can count the number of elements less than some value by writing

```
gMaxLength = /* ... some value ... */
int numShort = count_if(v.begin(), v.end(), LengthIsLessThan);
```

But just because this approach works does not mean that it is optimal. This approach is deeply flawed for several reasons, a handful of which are listed here:

- **It is error-prone.** Before we use `LengthIsLessThan`, we need to set `gMaxLength` to the correct value. If we forget, `LengthIsLessThan` will use the wrong value and we will get the wrong answer. Moreover, the compiler will not diagnose this sort of mistake.
- **It is not scalable.** If this problem arises again (i.e. we find another case where we need a unary function with access to more data), we need to introduce even more global variables. This leads to *namespace pollution*, where too many variables are in scope and it is easy to accidentally use one when another is expected.
- **It uses global variables.** Any use of global variables should send a shiver running down your spine. Global variables should be avoided at all costs, and the fact that we're using them here suggests that something is wrong with this setup.

None of the options we've considered are feasible or attractive. There has to be a better way to solve this, but how?

Functors to the Rescue

The problem is that a raw C++ function doesn't have access to enough information to return the correct answer. To solve this problem, we'll turn to a more powerful C++ entity: a *functor*. A functor (or *function object*) is an C++ class that acts like a function. Functors can be called using the familiar function call syntax, and can yield values and accept parameters just like regular functions. For example, suppose we create a functor class called `MyClass` imitating a function accepting an `int` and returning a `double`. Then we could “call” an object of type `MyClass` as follows:

```
MyClass myFunctor;
cout << myFunctor(137) << endl; // "Call" myFunctor with parameter 137
```

Notice that the syntax for calling a functor is the same for calling a function. This is no accident and is part of the appeal of functors.

To create a functor, we create an object that overloads the function call operator, `operator ()`. The name of this function is a bit misleading – it is a function called `operator ()`, not a function called `operator` that takes no parameters. Despite the fact that the name looks like “operator parentheses,” we're not redefining what it means to parenthesize the object. Instead, we're defining a function that gets called if we invoke the object like a function. Thus in the above code,

```
cout << myFunctor(137) << endl;
```

is equivalent to

```
cout << myFunctor.operator() (137) << endl;
```

Unlike other operators we've seen so far, when overloading the function call operator, you're free to return an object of any type (or even `void`) and can accept any number of parameters. For example, here's a sample functor that overloads the function call operator to print out a string:

```
class MyFunctor
{
public:
    void operator() (const string& str) const
    {
        cout << str << endl;
    }
};
```

Note that in the function definition there are two sets of parentheses. The first group is for the function name – `operator ()` – and the second for the parameters to `operator ()`. If we separated the implementation of `operator ()` from the class definition, it would look like this:

```
class MyFunctor
{
public:
    void operator() (const string& str) const;
};

void MyFunctor::operator() (const string& str) const
{
    cout << str << endl;
}
```

Now that we've written `MyFunctor`, we can use it as follows:

```
MyFunctor functor;
functor("Functor power!");
```

This code calls the functor and prints out “Functor power!”

At this point functors might seem like little more than a curiosity. “Sure,” you might say, “I can make an object that can be called like a function. But what does it buy me?” A lot more than you might initially suspect, it turns out. The key difference between a function and a functor is that a functor's function call operator is a *member function* whereas a raw C++ function is a *free function*. This means that a functor can access the following information when being called:

- Its local variables.
- Its parameters.
- Global variables.
- **Class data members.**

This last point is extremely important and is the key difference between a regular function and a functor. If a functor needs data beyond what can be communicated by its parameters, we can store that information as a data member inside the functor class. For example, consider the following functor class:

```

class StringAppender
{
public:
    /* Constructor takes and stores a string. */
    explicit StringAppender(const string& str) : toAppend(str) {}

    /* operator() prints out a string, plus the stored suffix. */
    void operator() (const string& str) const
    {
        cout << str << ' ' << toAppend << endl;
    }

private:
    const string toAppend;
};

```

This functor's constructor takes in a string and stores it for later use. Its `operator ()` function accepts a string, then prints it out suffixed with the string stored by the constructor. We use the `StringAppender` functor like this:

```

StringAppender myFunctor("is awesome");
myFunctor("C++");

```

This code will print out “C++ is awesome,” since the constructor stored the string “is awesome” and we passed “C++” as a parameter to the function. If you'll notice, though, in the actual function call we only passed in one piece of information. This is precisely why functors are so useful – they can mimic functions that take in a fixed amount of data, yet have access to as much information is necessary to solve the task at hand.

Let's return to the above example with `count_if`. Somehow we need to provide a unary function that can return whether a string is less than an arbitrary length. To solve this problem, instead of writing a unary function, we'll create a unary *functor* whose constructor stores the maximum length and whose `operator ()` accepts a string and returns whether it's of the correct length. Here's one possible implementation:

```

class ShorterThan
{
public:
    /* Accept and store an int parameter */
    explicit ShorterThan(int maxLength) : length(maxLength) {}

    /* Return whether the string length is less than the stored int. */
    bool operator() (const string& str) const
    {
        return str.length() < length;
    }

private:
    const int length;
};

```

Note that while `operator ()` takes in only a single parameter, it has access to the `length` field that was set up by the constructor. This is exactly what we want – a unary function that knows what value to compare the parameter's length against. To tie everything together, here's the code we'd use to count the number of strings in the vector that are shorter than the specified value:

```

ShorterThan st(length);
count_if(myVector.begin(), myVector.end(), st);

```

Functors are incredible when combined with STL algorithms for this very reason – they look and act like regular functions but have access to extra information. This is just your first taste of functors, as you continue your exploration of C++ you will recognize exactly how much they will influence your program design.

Look back to the above code with `count_if`. If you'll notice, we created a new `ShorterThan` object, then fed it to `count_if`. After the call to `count_if`, odds are we'll never use that particular `ShorterThan` object again. This is an excellent spot to use temporary objects, since we need a new `ShorterThan` for the function call but don't plan on using it afterwards. Thus, we can convert this code:

```
ShorterThan st(length)
count_if(myVector.begin(), myVector.end(), st);
```

Into this code:

```
count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

Here, `ShorterThan(length)` constructs a temporary `ShorterThan` functor with parameter `length`, then passes it to the `count_if` algorithm. Don't get tripped up by the syntax – `ShorterThan(length)` does *not* call the `ShorterThan`'s operator `()` function. Instead, it invokes the `ShorterThan` constructor with the parameter `length` to create a temporary object. Even if we had written the `operator()` function to take in an `int`, C++ would realize that the parentheses here means “construct an object” instead of “invoke `operator()`” from context.

Writing Functor-Compatible Code

In CS106B/X, you've seen how to write code that accepts a function pointer as a parameter. For example, the following code accepts a function that takes and returns a `double`, then prints a table of some values of that function:

```
const double LOWER_BOUND = 0.0;
const double UPPER_BOUND = 1.0;
const int    NUM_STEPS    = 25;
const double STEP         = (UPPER_BOUND - LOWER_BOUND) / NUM_STEPS;

void TabulateFunctionValues(double function(double))
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

For any function accepting and returning a `double`, we can call `TabulateFunctionValues` with that function as an argument. But what about functors? Can we pass them to `TabulateFunctionValues` as well? As an example, suppose we have some unary functor class `Reciprocal` that accepts a `double` and returns its reciprocal. Then is the following code legal?

```
TabulateFunctionValues(Reciprocal());
```

(Recall that `Reciprocal()` constructs a temporary `Reciprocal` object for use as the parameter to `TabulateFunctionValues`.)

At a high level, this code seems perfectly fine. After all, `Reciprocal` objects can be called as though they were functions taking and returning `doubles`, so it seems perfectly reasonable to pass a `Reciprocal` into `TabulateFunctionValues`. But despite the similarities, `Reciprocal` is *not* a function – it's a functor – and so the above code will not compile.

The problem is that C++'s static type system prevents function pointers from pointing to functors, even if the functor has the same parameter and return type as the function pointer. This is not without reason – the machine code for calling a function is very different from machine code for calling a functor, and if C++ were to conflate the two it would result either in slower function calls or undefined runtime behavior.

We know that this code won't compile, but how can we fix it so that it will? Let's begin with some observations, then generalize to the optimal solution. The above code will not compile because we're trying to provide a `Reciprocal` object to a function expecting a function pointer. This suggests one option – could we rewrite the `TabulateFunctionValues` function such that it accepts a `Reciprocal` as a parameter instead of a function pointer? For example, we could write the following:

```
void TabulateFunctionValues(Reciprocal function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, the above call will work correctly.

But what if we have another functor we want to use in `TabulateFunctionValues`? For example, we might write a functor called `Arccos` that computes the inverse cosine of its parameter. We could then provide an implementation of `TabulateFunctionValues` that looks like this:

```
void TabulateFunctionValues(Arccos function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

We could continue with this approach, rewriting the function once for every functor type we'd like to provide, but this method does not scale well. But is there some lesson we can take away from this method? Let's reprint all three versions of `TabulateFunctionValues`:

```
void TabulateFunctionValues(double function(double))
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Reciprocal function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Arccos function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Notice that the only difference between the three implementations of `TabulateFunctionValues` is the type of the parameter to the function. The rest of the code is identical. This suggests a rather elegant solution using templates. Instead of providing multiple different versions of `TabulateFunctionValues`, each specialized for a particular type of function or functors, we'll write a single template version of `TabulateFunctionValues` parameterized over the type of the argument. This is shown here:


```

template <typename UnaryFunction>
void TabulateFunctionValues(UnaryFunction function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}

```

Now, we can pass any type of object to `TabulateFunctionValues` that we want, provided that the argument can be called with a single `double` as a parameter to produce a value. This hearkens back to our discussion of implicit interfaces in the chapter on templates. Using a template function, we can accept any compatible type without worrying about how exactly it's implemented.

When writing functions that require a user-specified callback, you may want to consider parameterizing the function over the type of the callback instead of using function pointers. The resulting code will be more flexible and future generations of programmers will be much the better for your extra effort.

Storing Objects in STL maps, Part II

In the chapter on operator overloading, we demonstrated how to store custom objects as keys in an STL `map` by overloading the `<` operator. However, what if you want to store elements in a `map` or `set`, but not using the default comparison operator? For example, consider a `set<char *>` of C strings. Normally, the `<` operator will compare two `char *`s by seeing if one references memory with a lower address than the other. This isn't at all the behavior we want. First, it would mean that the `set` would store its elements in a seemingly random order, since the comparison is independent of the contents of the C strings. Second, if we tried to call `find` or `count` to determine membership in the `set`, since the `set` compares the *pointers* to the C strings, not the C strings themselves, `find` and `count` would return whether the given *pointer*, not the string pointed at, was contained in the `set`.

We need to tell the `set` that it should not use the `<` operator to compare C strings, but we can't simply provide an alternative `<` operator and expect the `set` to use it. Instead, we'll define a functor class whose `operator ()` compares two C strings lexicographically and returns whether one string compares less than the other. Here's one possible implementation:

```

class CStringCompare
{
public:
    bool operator() (const char* one, const char* two) const
    {
        return strcmp(one, two) < 0; // Use strcmp to do the comparison
    }
};

```

Then, to signal to the `set` that it should compare elements using `CStringCompare` instead of the default `<` operator, we'll parameterize the `set` over both `char*` and `CStringCompare`, as shown here:

```
set<char*, CStringCompare> mySet;
```

Because `CStringCompare` is a template argument, C++ treats `set<char *>` and `set<char *, CStringCompare>` as two different types. This means that you can only iterate over a `set<char *, CStringCompare>` with a `set<char *, CStringCompare>::iterator`. You will probably find `typedef` handy here to save some tedious typing.

This above discussion concerned the STL `set`, but you can use a similar trick on the STL `map`. For example, if we want to create a `map` whose keys are C strings, we could write

```
map<char*, int, CStringCompare> myMap;
```

STL Algorithms Revisited

Now that you're armed with the full power of C++ functors, let's revisit some of the STL algorithms we've covered and discuss how to maximize their firepower.

The very first algorithm we covered was `accumulate`, defined in the `<numeric>` header. If you'll recall, `accumulate` sums up the elements in a range and returns the result. For example, given a `vector<int>`, the following code returns the sum of all of the `vector`'s elements:

```
accumulate(myVector.begin(), myVector.end(), 0);
```

The first two parameters should be self-explanatory, and the third parameter (zero) represents the initial value of the sum.

However, this view of `accumulate` is limited, and to treat `accumulate` as simply a way to sum container elements would be an error. Rather, *accumulate is a general-purpose function for transforming a collection of elements into a single value.*

There is a second version of the `accumulate` algorithm that takes a binary function as a fourth parameter. This version of `accumulate` is implemented like this:

```
template <typename InputIterator, typename Type, typename BinaryFn>
inline Type accumulate(InputIterator start,
                      InputIterator stop,
                      Type accumulator,
                      BinaryFn fn)
{
    while(start != stop)
    {
        accumulator = fn(accumulator, *start);
        ++start;
    }
    return initial;
}
```

This `accumulate` iterates over the elements of a container, calling the binary function on the accumulator and the current element of the container and storing the result back in the accumulator. In other words, `accumulate` continuously updates the value of the accumulator based on its initial value and the values contained in the input range. Finally, `accumulate` returns the accumulator. Note that the version of `accumulate` we encountered earlier is actually a special case of the above version where the provided callback function computes the sum of its parameters.

To see `accumulate` in action, let's consider an example. Recall that the STL algorithm `lower_bound` returns an iterator to the first element in a range that compares greater than or equal to some value. However, `lower_bound` requires the elements in the iterator range to be in sorted order, so if you have an unsorted `vector`, you cannot use `lower_bound`. Let's write a function `UnsortedLowerBound` that accepts a range of iterators and a lower bound, then returns the value of the least element in the range greater than or equal to the lower bound. For simplicity, let's assume we're working with a `vector<int>` so that we don't get bogged down in template syntax, though this approach can easily be generalized.

Although this function can be implemented using loops, we can leverage off of `accumulate` to come up with a considerably more concise solution. Thus, we'll define a functor class to pass to `accumulate`, then write `UnsortedLowerBound` as a wrapper call to `accumulate` with the proper parameters.

Consider the following functor:

```
class LowerBoundHelper
{
public:
    explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
    int operator() (int bestSoFar, int current)
    {
        if(current >= lowestValue && current < bestSoFar) return current;
        return bestSoFar;
    }
private:
    const int lowestValue;
};
```

This functor's constructor accepts the value that we want to lower-bound. Its `operator ()` function accepts two `ints`, the first representing the lowest known value greater than `lowestValue` and the second the current value. If the value of the current element is greater than or equal to the lower bound and also less than the best value so far, `operator ()` returns the value of the current element. Otherwise, it simply returns the best value we've found so far. Thus if we call this functor on every element in the `vector` and keep track of the return value, we should end up with the lowest value in the `vector` greater than or equal to the lower bound. We can now write the `UnsortedLowerBound` function like this:

```
int UnsortedLowerBound(const vector<int>& input, int lowerBound)
{
    return accumulate(input.begin(), input.end(),
        numeric_limits<int>::max(),
        LowerBoundHelper(lowerBound));
}
```

Our entire function is simply a wrapped call to `accumulate`, passing a specially-constructed `LowerBoundHelper` object as a parameter. Note that we've used the value `numeric_limits<int>::max()` as the initial value for the accumulator. `numeric_limits`, defined in the `<limits>` header, is a traits class that exports useful information about the bounds and behavior of numeric types, and its `max` static member function returns the maximum possible value for an element of the specified type. We use this value as the initial value for the accumulator since any integer is less than it, so if the range contains no elements greater than the lower bound we will get `numeric_limits<int>::max()` back as a sentinel.

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed `transform` briefly in the chapter on STL algorithms in the context of `ConvertToUpperCase` and `ConvertToLowerCase`, but such examples are just the tip of the iceberg. `transform` is nothing short of a miracle function, and it arises a whole host of circumstances.*

Higher-order Programming

This discussion of functors was initially motivated by counting the number of short strings inside of an STL `vector`. We demonstrated that by using `count_if` with a custom functor as the final parameter, we were able to write code that counted the number of elements in a `vector<string>` whose length was less than a certain

* Those of you familiar with functional programming might recognize `accumulate` and `transform` as the classic higher-order functions Map and Reduce.

value. But while this code solved the problem efficiently, we ended up writing so much code that any potential benefits of the STL algorithms were dwarfed by the time spent writing the functor. For reference, here was the code we used:

```
class ShorterThan
{
public:
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string& str) const
    {
        return str.length() < length;
    }
private:
    int length;
};

const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));
```

Consider the following code which also solves the problem, but by using a simple `for` loop:

```
const int myValue = GetInteger();
int total = 0;
for(int i = 0; i < myVector.size(); ++i)
    if(myVector[i].length() < myValue) ++total;
```

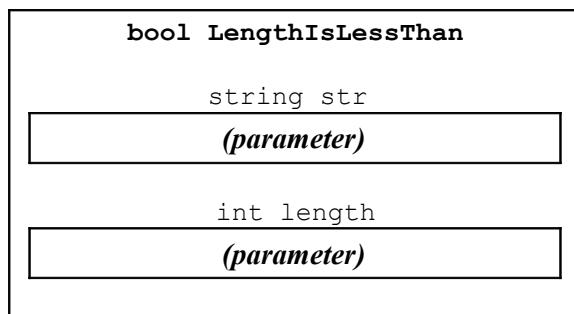
This code is considerably more readable than the functor version and is approximately a third as long. By almost any metric, this code would be considered superior to the earlier version.

If you'll recall, we were motivated to write this `ShorterThan` functor because we were unable to use `count_if` in conjunction with a traditional C++ function. Because `count_if` accepts as a parameter a unary function, we could not write a C++ function that could accept both the current container element and the value to compare its length against. However, we did note that were `count_if` to accept a binary function and extra client data, then we could have written a simple C++ function like this one:

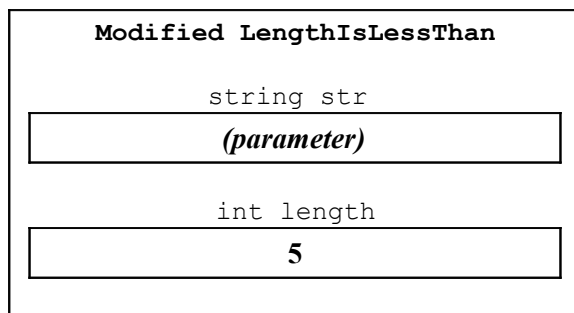
```
bool LengthIsLessThan(const string& str, int threshold)
{
    return str.length() < threshold;
}
```

And then passed it in, along with the cutoff length, to the `count_if` function.

The fundamental problem is that the STL `count_if` algorithm requires a single-parameter function, but the function we want to use requires two pieces of data. We want the STL algorithms to use our two-parameter function `LengthIsLessThan`, but with the second parameter always having the same value. What if somehow we could modify `LengthIsLessThan` by “locking in” the second parameter? In other words, we'd like to take a function that looks like this:



And transform it into another function that looks like this:



Now, if we call this special version of `LengthIsLessThan` with a single parameter (call it `str`), it would be as though we had called the initial version of `LengthIsLessThan`, passing as parameters the value of `str` and the stored value 5. This then returns whether the length of the `str` string is less than 5. Essentially, by binding the second parameter of the two-parameter `LengthIsLessThan` function, we end up with a one-parameter function that describes exactly the predicate function we want to provide to `count_if`. Thus, at a high level, the code we want to be able to write should look like this:

```
count_if(v.begin(), v.end(),
         the function formed by locking 5 as the second parameter of LengthIsLessThan);
```

This sort of programming, where functions can be created and modified just like regular objects, is known as *higher-order programming*. While by default C++ does not support higher-order programming, using functors and the STL functional programming libraries, in many cases it is possible to write higher-order code in C++. In the remainder of this chapter, we'll explore the STL functional programming libraries and see how to use higher-order programming to supercharge STL algorithms.

Adaptable Functions

To provide higher-order programming support, standard C++ provides the `<functional>` library. `<functional>` exports several useful functions that can transform and modify functions on-the-fly to yield new functions more suitable to the task at hand. However, because of several language limitations, the `<functional>` library can only modify specially constructed functions called “adaptable functions,” *functors* (not regular C++ functions) that export information about their parameter and return types. Fortunately, any one- or two-parameter function can easily be converted into an equivalent adaptable function. For example, suppose you want to make an adaptable function called `MyFunction` that takes a `string` by reference-to-`const` as a parameter and returns a `bool`, as shown below:

```

class MyFunction
{
public:
    bool operator() (const string& str) const
    {
        /* Function that manipulates a string */
    }
};

```

Now, to make this function an adaptable function, we need to specify some additional information about the parameter and return types of this functor's `operator ()` function. To assist in this process, the functional library defines a helper template class called `unary_function`, which is prototyped below:

```

template <typename ParameterType, typename ReturnType>
class unary_function;

```

The first template argument represents the type of the parameter to the function; the second, the function's return type.

Unlike the other classes you have seen before, the `unary_function` class contains no data members and no member functions. Instead, it performs some behind-the-scenes magic with the `typedef` keyword to export the information expressed in the template types to the rest of the functional programming library. Since we want our above functor to also export this information, we'll use a technique called *inheritance* to import all of the information from `unary_function` into our `MyFunction` functor. Because `MyFunction` accepts as a parameter an object of type `string` and returns a variable of type `bool`, we will have `MyFunction` inherit from the type `unary_function<string, bool>`. The syntax to accomplish this is shown below:

```

class MyFunction : public unary_function<string, bool>
{
public:
    bool operator() (const string& str) const
    {
        /* Function that manipulates a string */
    }
};

```

We'll explore inheritance in more detail later, but for now just think of it as a way for importing information from class into another. Note that although the function accepts as its parameter a `const string&`, we chose to use a `unary_function` specialized for the type `string`. The reason is somewhat technical and has to do with how `unary_function` interacts with other functional library components, so for now just remember that you should not specify reference-to-const types inside the `unary_function` template parametrization.

The syntax for converting a binary functor into an adaptable binary function works similarly to the above code for unary functions. Suppose that we'd like to make an adaptable binary function that accepts a `string` and an `int` and returns a `bool`. We begin by writing the basic functor code, as shown here:

```

class MyOtherFunction
{
public:
    bool operator() (const string& str, int val) const
    {
        /* Do something, return a bool. */
    }
};

```

To convert this functor into an adaptable function, we'll have it inherit from `binary_function`. Like `unary_function`, `binary_function` is a template class that's defined as

```
template <typename Param1Type, typename Param2Type, typename ResultType>
    class binary_function;
```

Thus the adaptable version of `MyOtherFunction` would be

```
class MyOtherFunction: public binary_function<string, int, bool>
{
public:
    bool operator() (const string& str, int val) const
    {
        /* Do something, return a bool. */
    }
};
```

While the above approach for generating adaptable functions is perfectly legal, it's a bit clunky and we still have a high ratio of boilerplate code to actual logic. Fortunately, the STL functional library provides the powerful but cryptically named `ptr_fun*` function that transforms a regular C++ function into an adaptable function. `ptr_fun` can convert both unary and binary C++ functions into adaptable functions with the correct parameter types, meaning that you can skip the hassle of the above code by simply writing normal functions and then using `ptr_fun` to transform them into adaptable functions. For example, given the following C++ function:

```
bool LengthIsLessThan(string myStr, int threshold)
{
    return myStr.length() < threshold;
}
```

If we need to get an adaptable version of that function, we can write `ptr_fun(LengthIsLessThan)` in the spot where the adaptable function is needed.

`ptr_fun` is a useful but imperfect tool. Most notably, you cannot use `ptr_fun` on functions that accept parameters as reference-to-const. `ptr_fun` returns a `unary_function` object, and as mentioned above, you cannot specify reference-to-const as template arguments to `unary_function`. Also, because of the way that the C++ compiler generates code for functors, code that uses `ptr_fun` can be a bit slower than code using functors.

For situations where you'd like to convert a member function into an adaptable function, you can use the `mem_fun` or `mem_fun_ref` functions. These functions convert member functions into unary functions that accept as input a receiver object, then invoke that member function on the receiver. The difference between `mem_fun` and `mem_fun_ref` is how they accept their parameters – `mem_fun` accepts a *pointer* to the receiver object, while `mem_fun_ref` accepts a *reference* to the receiver. For example, given a `vector<string>`, the following code will print out the lengths of all of the strings in the vector:

```
transform(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"),
    mem_fun_ref(&string::length));
```

Let's dissect this call to `transform`, since there's a lot going on. The first two parameters delineate the input range, in this case the full contents of `myVector`. The third parameter specifies where to put the output, and since here it's an `ostream_iterator` the output will be written to `cout`. The final parameter is `mem_fun_ref(&string::length)`, a function that accepts as input a `string` and then returns the value of the `length` member function called on that `string`.

* `ptr_fun` is short for “pointer function”, since you're providing as a parameter a function pointer. It should not be confused with Nick Parlante's excellent video “Binky's Pointer Fun.”

`mem_fun_ref` can also be used to convert unary (one-parameter) member functions into adaptable binary functions that take as a first parameter the object to apply the function to and as a second parameter the parameter to the function. When we cover binders in the next section, you should get a better feel for exactly how useful this is.

Binding Parameters

Now that we've covered how the STL functional library handles adaptable functions, let's consider how we can use them in practice.

At the beginning of this chapter, we introduced the notion of *parameter binding*, converting a two-parameter function into a one-parameter function by locking in the value of one of its parameters. To allow you to bind parameters to functions, the STL functional programming library exports two functions, `bind1st` and `bind2nd`, which accept as parameters an adaptable function and a value to bind and return new functions that are equal to the old functions with the specified values bound in place. For example, given the following implementation of `LengthIsLessThan`:

```
bool LengthIsLessThan(string str, int threshold)
{
    return str.length() < threshold;
}
```

We could use the following syntax to construct a function that's `LengthIsLessThan` with the value five bound to the second parameter:

```
bind2nd(ptr_fun(LengthIsLessThan), 5);
```

The line `bind2nd(ptr_fun(LengthIsLessThan), 5)` first uses `ptr_fun` to generate an adaptable version of the `LengthIsLessThan` function, then uses `bind2nd` to lock the parameter 5 in place. The result is a new unary function that accepts a `string` parameter and returns if that string's length is less than 5, the value we bound to the second parameter. Since `bind2nd` is a function that accepts a function as a parameter and returns a function as a result, `bind2nd` is a function that is sometimes referred to as a *higher-order function*.

Because the result of the above call to `bind2nd` is a unary function that determines if a string has length less than five, we can use the `count_if` algorithm to count the number of values less than five by using the following code:

```
count_if(container.begin(), container.end(),
         bind2nd(ptr_fun(LengthIsLessThan), 5));
```

Compare this code to the functor-based approach illustrated at the start of this chapter. This version of the code is much, *much* shorter than the previous version. If you aren't beginning to appreciate exactly how much power and flexibility the `<functional>` library provides, skip ahead and take a look at the practice problems.

The `bind1st` function acts similarly to `bind2nd`, except that (as its name suggests) it binds the first parameter of a function. Returning to the above example, given a `vector<int>`, we could count the number of elements in that `vector` smaller than the length of string "C++!" by writing

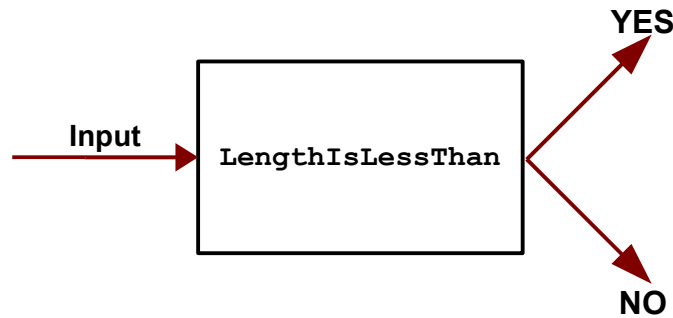
```
count_if(myVector.begin(), myVector.end(),
         bind1st(ptr_fun(LengthIsLessThan), "C++!"));
```

In the STL functional programming library, parameter binding is restricted only to binary functions. Thus you cannot bind a parameter in a three-parameter function to yield a new binary function, nor can you bind the para-

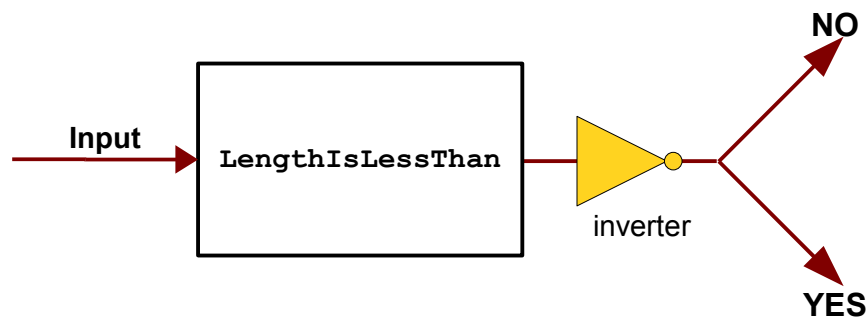
meter of a unary function to yield a zero-parameter (“nullary”) function. For these operations, you'll need to create your own custom functors, as shown in the practice problems at the end of this chapter.

Negating Results

Suppose that given a function `LengthIsLessThan`, we want to find the number of strings in a container that are *not* less than a certain length. While we could simply write another function `LengthIsNotLessThan`, it would be much more convenient if we could somehow tell C++ to take whatever value `LengthIsLessThan` returns and to use the opposite result. That is, given a function that looks like this:



We'd like to change it into a function that looks like this:



This operation is *negation* – constructing a new function whose return value has the opposite value of the input function. There are two STL negator functions – `not1` and `not2` – that return the negated result of a unary or binary predicate function, respectively. Thus, the above function that's a negation of `LengthIsLessThan` could be written as `not2(ptr_fun(LengthIsLessThan))`. Since `not2` returns an adaptable function, we can then pass the result of this function to `bind2nd` to generate a unary function that returns whether a string's length is at least a certain threshold value. For example, here's code that returns the number of strings in a container with length at least 5:

```
count_if(container.begin(), container.end(),
         bind2nd(not2(ptr_fun(LengthIsLessThan)), 5));
```

While this line is dense, it elegantly solves the problem at hand by combining and modifying existing code to create entirely different functions. Such is the beauty and simplicity of higher-order programming – why rewrite code from scratch when you already have all the pieces individually assembled?

Operator Functions

Let's suppose that you have a container of `ints` and you'd like to add 137 to each of them. Recall that you can use the STL `transform` algorithm to apply a function to each element in a container and then store the result. Because we're adding 137 to each element, we might consider writing a function like this one:

```
int Add137(int param)
{
    return param + 137;
}
```

And then writing

```
transform(container.begin(), container.end(), container.begin(), Add137);
```

While this code works correctly, this approach is not particularly robust. What if later on we needed to increment all elements in a container by 42, or perhaps by an arbitrary value? Thus, we might want to consider replacing `Add137` by a function like this one:

```
int AddTwoInts(int one, int two)
{
    return one + two;
}
```

And then using binders to lock the second parameter in place. For example, here's code that's equivalent to what we've written above:

```
transform(container.begin(), container.end(), container.begin(),
          bind2nd(ptr_fun(AddTwoInts), 137));
```

At this point, our code is correct, but it can get a bit annoying to have to write a function `AddTwoInts` that simply adds two integers. Moreover, if we then need code to increment all `doubles` in a container by 1.37, we would need to write another function `AddTwoDoubles` to avoid problems from typecasts and truncations. Fortunately, because this is a common use case, the STL functional library provides a large number of template adaptable function classes that simply apply the basic C++ operators to two values. For example, in the above code, we can use the adaptable function class `plus<int>` instead of our `AddTwoInts` function, resulting in code that looks like this:

```
transform(container.begin(), container.end(), container.begin(),
          bind2nd(plus<int>(), 137));
```

Note that we need to write `plus<int>()` instead of simply `plus<int>`, since we're using the temporary object syntax to construct a `plus<int>` object for `bind2nd`. Forgetting the parentheses can cause a major compiler error headache that can take a while to track down. Also notice that we don't need to use `ptr_fun` here, since `plus<int>` is already an adaptable function.

For reference, here's a list of the common “operator functions” exported by `<functional>`:

<code>plus</code>	<code>multiplies</code>	<code>divides</code>	<code>minus</code>	<code>modulus</code>	<code>equal_to</code>
<code>less</code>	<code>less_equal</code>	<code>greater</code>	<code>greater_equal</code>	<code>not_equal_to</code>	

To see an example that combines the techniques from the previous few sections, let's consider a function that accepts a `vector<double>` and converts each element in the `vector` to its reciprocal (one divided by the value). Because we want to convert each element with value x to the value $1/x$, we can use a combination of binders and

operator functions to solve this problem by binding the value 1.0 to the first parameter of the `divides<double>` functor. The result is a unary function that accepts a parameter of type `double` and returns the element's reciprocal. The resulting code looks like this:

```
transform(v.begin(), v.end(), v.begin(), bind1st(divides<double>(), 1.0));
```

This code is concise and elegant, solving the problem in a small space and making explicit what operations are being performed on the data.

Implementing the `<functional>` Library

Now that we've seen how the `<functional>` library works from a client perspective, let's discuss how the library is put together. What's so special about adaptable functions? How does `ptr_fun` convert a regular function into an adaptable one? How do functions like `bind2nd` and `not1` work?

Let's begin by looking at exactly what an adaptable function is. Recall that adaptable functions are functors that inherit from either `unary_function` or `binary_function`. Neither of these template classes are particularly complicated; here's the complete definition of `unary_function`.*

```
template <typename ArgType, typename RetType> class unary_function
{
public:
    typedef ArgType argument_type;
    typedef RetType result_type;
};
```

This class contains no data members and no member functions. Instead, it exports two typedefs – one renaming `ArgType` to `argument_type` and one renaming `RetType` to `result_type`. When you create an adaptable function that inherits from `unary_function`, your class acquires these typedefs. For example, if we write the following adaptable function:

```
class IsPositive: public unary_function<double, bool>
{
public:
    bool operator() (double value) const
    {
        return value > 0.0;
    }
};
```

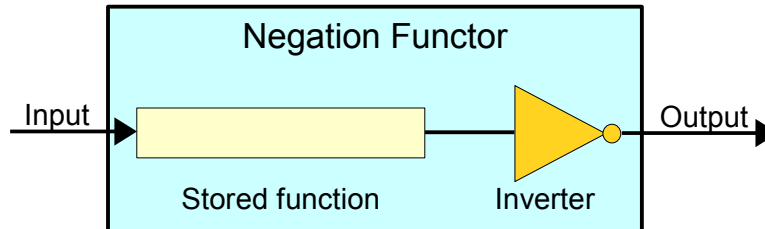
The statement `public unary_function<double, double>` imports two typedefs into `IsPositive`: `argument_type` and `return_type`, equal to `double` and `bool`, respectively. Right now it might not be apparent how these types are useful, but as we begin implementing the other pieces of the `<functional>` library it will become more apparent.

Implementing `not1`

To begin our backstage tour of the `<functional>` library, let's see how to implement the `not1` function. Recall that `not1` accepts as a parameter a unary adaptable predicate function, then returns a new adaptable function that yields opposite values as the original function. For example, `not1(IsPositive())` would return a function that returns whether a value is *not* positive.

* Technically speaking `unary_function` and `binary_function` are structs, but this is irrelevant here.

Implementing `not1` requires two steps. First, we'll create a template functor class parameterized over the type of the adaptable function to negate. This functor's constructor will take as a parameter an adaptable function of the proper type and store it for later use. We'll then implement its `operator()` function such that it calls the stored function and returns the negation of the result. Graphically, this is shown here:



Once we have designed this functor, we'll have `not1` accept an adaptable function, wrap it in our negating functor, then return the resulting object to the caller. This means that the return value of `not1` is an adaptable unary predicate function that returns the opposite value of its parameter, which is exactly what we want.

Let's begin by writing the template functor class, which we'll call `unary_negate` (this is the name of the functor class generated by the `<functional>` library's `not1` function). We know that this functor should be parameterized over the type of the adaptable function it negates, so we can begin by writing the following:

```

template <typename UnaryPredicate> class unary_negate
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    /* ... */
private:
    UnaryPredicate p;
};

```

Here, the constructor accepts an object of type `UnaryPredicate`, then stores it in the data member `p`.

Now, let's implement the `operator()` function, which, as you'll recall, should take in a parameter, feed it into the stored function `p`, then return the inverse result. The code for this function looks like this:

```

template <typename UnaryPredicate> class unary_negate
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool operator() (const /* what goes here? */& param) const
    {
        return !p(param);
    }
private:
    UnaryPredicate p;
};

```

We've almost finished writing our `unary_negate` class, but we have a slight problem – what is the type of the parameter to `operator()`? This is where adaptable functions come in. Because `UnaryPredicate` is adaptable, it must export a type called `argument_type` corresponding to the type of its argument. We can thus define our `operator()` function to accept a parameter of type `typename UnaryPredicate::argument_type` to guarantee that it has the same parameter type as the `UnaryPredicate` class.* The updated code for `unary_negate` looks like this:

```
template <typename UnaryPredicate> class unary_negate
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool operator() (const typename UnaryPredicate::argument_type& param) const
    {
        return !p(param);
    }
private:
    UnaryPredicate p;
};
```

That's quite a mouthful, but it's exactly the solution we're looking for.

There's one step left to finalize this functor class, and that's to make the functor into an adaptable function by having it inherit from the proper `unary_function`. Since the functor's argument type is `typename UnaryPredicate::argument_type` and its return type is `bool`, we'll inherit from `unary_function<typename UnaryPredicate::argument_type, bool>`. The final code for `unary_negate` is shown here:

```
template <typename UnaryPredicate>
class unary_negate:
    public unary_function<typename UnaryPredicate::argument_type, bool>
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool operator() (const typename UnaryPredicate::argument_type& param) const
    {
        return !p(param);
    }
private:
    UnaryPredicate p;
};
```

We've now finished writing our functor class to perform the negation, and all that's left to do is write `not1`. `not1` is much simpler than `unary_negate`, since it simply has to take in a parameter and wrap it in a `unary_negate` functor. This is shown here:

```
template <typename UnaryPredicate>
unary_negate<UnaryPredicate> not1(const UnaryPredicate& pred)
{
    return unary_negate<UnaryPredicate>(pred);
}
```

That's all there is to it – we've successfully implemented `not1`!

* Remember that the type is `typename UnaryPredicate::argument_type`, not `UnaryPredicate::argument_type`. `argument_type` is nested inside `UnaryPredicate`, and since `UnaryPredicate` is a template argument we have to explicitly use `typename` to indicate that `argument_type` is a type.

You might be wondering why there are two steps involved in writing `not1`. After all, once we have the functor that performs negation, why do we need to write an additional function to create it? The answer is *simplicity*. We don't need `not1`, but having it available reduces complexity. For example, using the `IsPositive` adaptable function from above, let's suppose that we want to write code to find the first nonpositive element in a `vector`. Using the `find_if` algorithm and `not1`, we'd write this as follows:

```
vector<double>::iterator itr = find_if(v.begin(), v.end(), not1(IsPositive()));
```

If instead of using `not1` we were to explicitly create a `unary_negate` object, the code would look like this:

```
vector<double>::iterator itr = find_if(v.begin(), v.end(),
                                     unary_negate<IsPositive>(IsPositive()));
```

That's quite a mouthful. When calling the template function `not1`, the compiler automatically infers the type of the argument and constructs an appropriately parameterized `unary_negate` object. If we directly use `unary_negate`, C++ will not perform type inference and we'll have to spell out the template arguments ourselves. The pattern illustrated here – having a template class and a template function to create it – is common in library code because it lets library clients use complex classes without ever having to know how they're implemented behind-the-scenes.

Implementing `ptr_fun`

Now that we've seen how `not1` works, let's see if we can construct the `ptr_fun` function. At a high level `ptr_fun` and `not1` work the same way – they each accept a parameter, construct a special functor class based on the parameter, then return it to the caller. The difference between `not1` and `ptr_fun`, however, is that there are two different versions of `ptr_fun` – one for unary functions and one for binary functions. The two versions work almost identically and we'll see how to implement them both, but for simplicity we'll begin with the unary case.

To convert a raw C++ unary function into an adaptable unary function, we need to wrap it in a functor that inherits from the proper `unary_function` base class. We'll make this functor's `operator()` function simply call the stored function and return its value. To be consistent with the naming convention of the `<functional>` library, we'll call the functor `pointer_to_unary_function` and will parameterize it over the argument and return types of the function. This is shown here:

```
template <typename ArgType, typename RetType>
class pointer_to_unary_function: public unary_function<ArgType, RetType>
{
public:
    explicit pointer_to_unary_function(ArgType fn(RetType)) : function(fn) {}

    RetType operator() (const ArgType& param) const
    {
        return function(param);
    }
private:
    ArgType (*function)(RetType);
};
```

There isn't that much code here, but it's fairly dense. Notice that we inherit from `unary_function<ArgType, RetType>` so that the resulting functor is adaptable. Also note that the argument and return types of `operator()` are considerably easier to determine than in the `unary_negate` case because they're specified as template arguments.

Now, how can we implement `ptr_fun` to return a correctly-constructed `pointer_to_unary_function`? Simple – we just write a template function parameterized over argument and return types, accept a function pointer of the appropriate type, then wrap it in a `pointer_to_unary_function` object. This is shown here:

```
template <typename ArgType, typename RetType>
pointer_to_unary_function<ArgType, RetType> ptr_fun(RetType function(ArgType))
{
    return pointer_to_unary_function<ArgType, RetType>(function);
}
```

This code is fairly dense, but gets the job done.

The implementation of `ptr_fun` for binary functions is similar to the implementation for unary functions. We'll create a template functor called `pointer_to_binary_function` parameterized over its argument and return types, then provide an implementation of `ptr_fun` that constructs and returns an object of this type. This is shown here:

```
template <typename Arg1, typename Arg2, typename Ret>
class pointer_to_binary_function: public binary_function<Arg1, Arg2, Ret>
{
public:
    explicit pointer_to_binary_function(Ret fn(Arg1, Arg2)) : function(fn) {}

    Ret operator() (const Arg1& arg1, const Arg2& arg2) const
    {
        return function(arg1, arg2);
    }
private:
    Ret (*function) (Arg1, Arg2);
};

template <typename Arg1, typename Arg2, typename Ret>
pointer_to_binary_function<Arg1, Arg2, Ret> ptr_fun(Ret function(Arg1, Arg2))
{
    return pointer_to_binary_function<Arg1, Arg2, Ret>(function);
}
```

Note that we now have *two* versions of `ptr_fun` – one that takes in a unary function and one that takes in a binary function. This isn't a problem, since the two template functions have different signatures.

Implementing `bind1st`

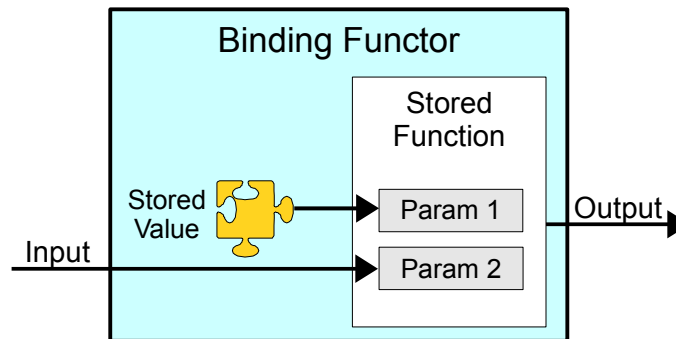
To wrap up our tour of the `<functional>` library, let's see how to implement `bind1st`. If you'll recall, `bind1st` takes in a binary adaptable function and a value, then returns a new unary function equal to the input function with the first parameter locked in place. We'll follow the pattern of `not1` and `ptr_fun` by writing a template functor class called `binder1st` that actually does the binding, then having `bind1st` construct and return an object of the proper type.

Before proceeding with our implementation of `binder1st`, we need to take a quick detour into the inner workings of the `binary_function` class. Like `unary_function`, `binary_function` exports typedefs so that other parts of the `<functional>` library can recover the argument and return types of adaptable functions. However, since a binary function has two arguments, the names of the exported types are slightly different. `binary_function` provides the following three typedefs:

- `first_argument_type`, the type of the first argument,
- `second_argument_type`, the type of the second argument, and
- `result_type`, the function's return type.

We will need to reference each of these type names when writing `binder1st`.

Now, how do we implement the `binder1st` functor? Here is one possible implementation. The `binder1st` constructor will accept and store an adaptable binary function and the value for its first argument. `binder1st` then provides an implementation of `operator()` that takes a single parameter, then invokes the stored function passing in the function parameter and the saved value. This is shown here:



Let's begin implementing `binder1st`. The functor has to be a template, since we'll be storing an arbitrary adaptable function and value. However, we only need to parameterize the functor over the type of the binary adaptable function, since we can determine the type of the first argument from the adaptable function's `first_argument_type`. We'll thus begin with the following implementation:

```
template <typename BinaryFunction> class binder1st
{
    /* ... */
};
```

Now, let's implement the constructor. It should take in two parameters – one representing the binary function and the other the value to lock into place. The first will have type `BinaryFunction`; the second, `typename BinaryFunction::first_argument_type`. This is shown here:

```
template <typename BinaryFunction> class binder1st
{
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
        function(fn), first(arg) {}

    /* ... */
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

Phew! That's quite a mouthful, but is the reality of much library template code. Look at the declaration of the `first` data member. Though it may seem strange, this is the correct way to declare a data member whose type is a type nested inside a template argument.

We now have the constructor written and all that's left to take care of is `operator()`. Conceptually, this function isn't very difficult, and if we ignore the parameter and return types have the following implementation:

```
template <typename BinaryFunction> class binder1st
{
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
        function(fn), first(arg) {}

    /* ret */ operator() (const /* arg */& param) const
    {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

What are the argument and return types for this function? Well, the function returns whatever object is produced by the stored function, which has type `typename BinaryFunction::result_type`. The function accepts a value for use as the second parameter to the stored function, so it must have type `typename BinaryFunction::second_argument_type`. This results in the following code:

```
template <typename BinaryFunction> class binder1st
{
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
        function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator() (const typename BinaryFunction::second_argument_type& param) const
    {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

We're almost finished, and all that's left for `binder1st` is to make it adaptable. Using the logic from above, we'll have it inherit from the proper instantiation of `unary_function`, as shown here:

```

template <typename BinaryFunction>
class binder1st:
    public unary_function<typename BinaryFunction::second_argument_type,
                        typename BinaryFunction::result_type>
{
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
        function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator() (const typename BinaryFunction::second_argument_type& param) const
    {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};

```

That's it for the `binder1st` class. As you can see, the code is dense and does a lot of magic with `typename` and nested types. Without adaptable functions, code like this would not be possible.

To finish up our discussion, let's implement `bind1st`. This function isn't particularly tricky, though we do need to do a bit of work to extract the type of the value to lock in place:

```

template <typename BinaryFunction>
binder1st<BinaryFunction>
bind1st(const BinaryFunction& fn,
        const typename BinaryFunction::first_argument_type& arg)
{
    return binder1st<BinaryFunction>(fn, arg);
}

```

We now have a complete working implementation of `bind1st`. If you actually open up the `<functional>` header and peek around inside, the code you'll find will probably bear a strong resemblance to what we've written here.

Limitations of the Functional Library

While the STL functional library is useful in a wide number of cases, the library is unfortunately quite limited. `<functional>` only provides support for adaptable unary and binary functions, but commonly you'll encounter situations where you will need to bind and negate functions with more than two parameters. In these cases, one of your only options is to construct functor classes that accept the extra parameters in their constructors. Similarly, there is no support for function composition, so we could not create a function that computes $2x + 1$ by calling the appropriate combination of the `plus` and `multiplies` functors. However, the next version of C++, nicknamed “C++0x,” promises to have more support for functional programming of this sort. For example, it will provide a general function called `bind` that lets you bind as many values as you'd like to a function of arbitrary arity. Keep your eyes peeled for the next release of C++ – it will be far more functional than the current version!

More to Explore

Although this is a C++ text, if you're interested in functional programming, you might want to consider learning other programming languages like Scheme or Haskell. Functional programming is an entirely different way of thinking about computation, and if you're interested you should definitely consider expanding your horizons with other languages. The ever-popular class CS242 (Programming Languages) is a great place to look.

Practice Problems

We've covered a lot of programming techniques in this chapter and there are no shortage of applications for the material. Here are some problems to get you thinking about how functors and adaptable functions can influence your programming style:

1. The STL algorithm `for_each` accepts as parameters a range of iterators and a unary function, then calls the function on each argument. Unusually, the return value of `for_each` is the unary function passed in as a parameter. Why might this be? ♦
2. Using the fact that `for_each` returns the unary function passed as a parameter, write a function `MyAccumulate` that accepts as parameters a range of `vector<int>::iterator`s and an initial value, then returns the sum of all of the values in the range, starting at the specified value. Do not use any loops – instead, use `for_each` and a custom functor class that performs the addition.
3. Write a function `AdvancedBiasedSort` that accepts as parameters a `vector<string>` and a string “winner” value, then sorts the range, except that all strings equal to the winner are at the front of the vector. Do not use any loops. (*Hint: Use the STL sort algorithm and functor that stores the “winner” parameter.*) ♦
4. Modify the above implementation of `AdvancedBiasedSort` so that it works over an arbitrary range of iterators over strings, not just a `vector<string>`. Then modify it once more so that the iterators can iterate over any type of value.
5. The STL `generate` algorithm is defined as `void generate(ForwardIterator start, ForwardIterator end, NullaryFunction fn)` and iterates over the specified range storing the return value of the zero-parameter function `fn` as it goes. For example, calling `generate(v.begin(), v.end(), rand)` would fill the range `[v.begin() to v.end())` with random values. Write a function `FillAscending` that accepts an iterator range, then sets the first element in the range to zero, the second to one, etc. Do not use any loops.
6. Write a function `ExpungeLetter` that accepts four parameters – two iterators delineating an input range of strings, one iterator delineating the start of an output range, and a character – then copies the strings in the input range that do not contain the specified character into the output range. The function should then return an iterator one past the last location written. Do not use loops. (*Hint: Use the `remove_copy_if` algorithm and a custom functor.*)
7. The *standard deviation* of a set of data is a measure of how much the data varies from its average value. Data with a small standard deviation tends to cluster around a point, while data with large standard deviation will be more spread out.

The formula for the standard deviation of a set of data $\{x_1, x_2, \dots, x_n\}$ is

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$$

Here, $\bar{x}$ is the average of the data points.

To give a feeling for this formula, given the data points 1, 2, 3, the average of the data points is 2, so the standard deviation is given by

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2} = \sqrt{\frac{1}{3} \sum_{i=1}^3 (x_i - 2)^2} = \sqrt{\frac{1}{3} ((1-2)^2 + (2-2)^2 + (3-2)^2)} = \sqrt{\frac{1}{3} (1+0+1)} = \sqrt{\frac{2}{3}}$$

Write a function `StandardDeviation` that accepts an input range of iterators over doubles (or values implicitly convertible to doubles) and returns its standard deviation. Do not use any loops – instead use the `accumulate` function to compute the average, then use `accumulate` once more to compute the sum. (*Hint: To get the number of elements in the range, you can use the `distance` function*) ♦

8. Write a function `ClearAllStrings` that accepts as input a range of iterators over C strings that sets each string to be the empty string. If you harness the `<functional>` library correctly here, the function body will be only a single line of code. ♦
9. Rewrite the `ClearAllStrings` function so that it works on a range of C++ strings. As with the above exercise, the function body can be written in a single line of code. ♦
10. The ROT128 cipher is a weak encryption cipher that works by adding 128 to the value of each character in a string to produce a garbled string. Since `char` can only hold 256 different values, two successive applications of ROT128 will produce the original string. Write a function `ApplyROT128` that accepts a string and returns the string's ROT128 cipher equivalent.
11. Write a template function `CapAtValue` that accepts a range of iterators and a value by reference-to-`const` and replaces all elements in the range that compare greater than the parameter with a copy of the parameter. (*Hint: use the `replace_if` algorithm*) ♦
12. One piece of functionality missing from the `<functional>` library is the ability to bind the first parameter of a unary function to form a nullary function. In this practice problem, we'll implement a function called `BindOnly` that transforms a unary adaptable function into a nullary function.
 - a. Write a template functor class `BinderOnly` parameterized whose constructor accepts an adaptable function and a value to bind and whose `operator()` function calls the stored function passing in the stored value as a parameter. Your class should have this interface:

```
template <typename UnaryFunction> class BinderOnly
{
public:
    BinderOnly(const UnaryFunction& fn,
               const typename UnaryFunction::argument_type& value);
    RetType operator() () const;
};
```

- b. Write a template function `BindOnly` that accepts the same parameters as the `BinderOnly` constructor and returns a `BinderOnly` of the proper type. The signature for this function should be

```
template <typename UnaryFunction>
    BinderOnly<UnaryFunction>
    BindOnly(const UnaryFunction &fn,
             const typename UnaryFunction::argument_type& value);
```

13. Another operation not supported by the `<functional>` library is *function composition*. For example, given two functions `f` and `g`, the composition `g ∘ f` is a function such that `g ∘ f(x) = g(f(x))`. In this example, we'll write a function `Compose` that lets us compose two unary functions of compatible types.
 - a. Write a template functor `UnaryCompose` parameterized over two adaptable function types whose constructor accepts and stores two unary adaptable functions and whose `operator()` accepts a single parameter and returns the composition of the two functions applied to that argument. Make sure that `UnaryCompose` is an adaptable unary function.
 - b. Write a wrapper function `Compose` that takes in the same parameters as `UnaryCompose` and returns a properly-constructed `UnaryCompose` object.
 - c. Explain how to implement `not1` using `Compose` and `logical_not`, a unary adaptable function exported by `<functional>` that returns the logical inverse of its argument.

Chapter 27: Introduction to Exception Handling

Forty years ago, goto-laden code was considered perfectly good practice. Now we strive to write structured control flows. Twenty years ago, globally accessible data was considered perfectly good practice. Now we strive to encapsulate data. Ten years ago, writing functions without thinking about the impact of exceptions was considered good practice. Now we strive to write exception-safe code.

Time goes on. We live. We learn.

– Scott Meyers, author of *Effective C++* and one of the leading experts on C++. [Mey05]

In an ideal world, network connections would never fail, files would always exist and be properly formatted, users would never type in malformed input, and computers would never run out of memory. Realistically, though, all of the above can and will occur and your programs will have to be able to respond to them gracefully. In these scenarios, the normal function-call-and-return mechanism is not robust enough to signal and report errors and you will have to rely on *exception handling*, a C++ language feature that redirects program control in case of emergencies.

Exception handling is a complex topic and will have far-reaching effects on your C++ code. This chapter introduces the motivation underlying exception handling, basic exception-handling syntax, and some advanced techniques that can keep your code operating smoothly in an exception-filled environment.

A Simple Problem

Up to this point, all of the programs you've written have proceeded linearly – they begin inside a special function called `main`, then proceed through a chain of function calls and returns until (hopefully) hitting the end of `main`. While this is perfectly acceptable, it rests on the fact that each function, given its parameters, can perform a meaningful task and return a meaningful value. However, in some cases this simply isn't possible.

Suppose, for example, that we'd like to write our own version of the CS106B/X `StringToInteger` function, which converts a `string` representation of a number into an `int` equivalent. One possible (partial) implementation of `StringToInteger` might look like this:*

```
int StringToInteger(const string &input)
{
    stringstream converter(input);
    int result; // Try reading an int, fail if we're unable to do so.
    converter >> result;
    if(converter.fail())
        // What should we do here?

    char leftover; // See if anything's left over. If so, fail.
    converter >> leftover;
    if(!converter.fail())
        return result;
    else
        // What should we do here?
}
```

* This is based off of the `GetInteger` function we covered in the chapter on streams. Instead of looping and reprompting the user for input at each step, however, it simply reports errors on failure.

If the parameter input is a string with a valid integer representation, then this function simply needs to perform the conversion. But what should our function do if the parameter doesn't represent an integer? One possible option, and the one used by the CS106B/X implementation of `StringToInteger`, is to call a function like `Error` that prints an error and terminates the program. This response seems a bit drastic and is a decidedly sub-optimal solution for several reasons. First, calling `Error` doesn't give the program a chance to recover from the problem. `StringToInteger` is a simple utility function, not a critical infrastructure component, and if it fails chances are that there's an elegant way to deal with the problem. For example, if we're using `StringToInteger` to convert user input in a text box into an integer for further processing, it makes far more sense to re-prompt the user than to terminate the program. Second, in a very large or complicated software system, it seems silly to terminate the program over a simple string error. For example, if this `StringToInteger` function were used in an email client to convert a string representation of a time to an integer format (parsing the hours and minutes separately), it would be disastrous if the program crashed whenever receiving malformed emails. In essence, while using a function like `Error` will prevent the program from continuing with garbage values, it is simply too drastic a move to use in serious code.

This approach suggests a second option, one common in pure C – *sentinel values*. The idea is to have functions return special values meaning “this value indicates that the function failed to execute correctly.” In our case, we might want to have `StringToInteger` return -1 to indicate an error, for example. Compared with the “drop everything” approach of `Error` this may seem like a good option – it reports the error and gives the calling function a chance to respond. However, there are several major problems with this method. First, in many cases it is not possible to set aside a value to indicate failure. For example, suppose that we choose to reserve -1 as an error code for `StringToInteger`. In this case, we'd make all of our calls to `StringToInteger` as

```
if(StringToInteger(myParam) == -1) { /* ... handle error ... */ }
```

But what happens if the input to `StringToInteger` is the string “-1”? In this case, whether or not the `StringToInteger` function completes successfully, it will still return -1 and our code might confuse it with an error case.

Another serious problem with this approach is that if each function that might possibly return an error has to reserve sentinel values for errors, we might accidentally check the return value of one function against the error code of another function. Imagine if there were several constants floating around named `ERROR`, `STATUS_ERROR`, `INVALID_RESULT`, etc., and whenever you called a function you needed to check the return value against the correct one of these choices. If you chose incorrectly, even with the best of intentions your error-checking would be invalid.

Yet another shortcoming of this approach is that in some cases it will be impossible to reserve a value for use as a sentinel. For example, suppose that a function returns a `vector<double>`. What special `vector<double>` should we choose to use as a sentinel?

However, the most serious problem with the above approach is that you as a programmer can ignore the return value without encountering any warnings. Even if `StringToInteger` returns a sentinel value indicating an error, there are no compile-time or runtime warnings if you choose not to check for a return value. In the case of `StringToInteger` this may not be that much of a problem – after all, holding a sentinel value instead of a meaningful value will not immediately crash the program – but this can lead to problems down the line that can snowball into fully-fledged crashes. Worse, since the crash will probably be caused by errors from far earlier in the code, these sorts of problems can be nightmarish to debug. Surprisingly, experience shows that many programmers – either out of negligence or laziness – forget to check return values for error codes and snowball effects are rather common.

We seem to have reached an unsolvable problem. We'd like an error-handling system that, like `Error`, prevents the program from continuing normally when an error occurs. At the same time, however, we'd like the elegance

of sentinel values so that we can appropriately process an error. How can we combine the strengths of both of these approaches into a single system?

Exception Handling

The reason the above example is such a problem is that the normal C++ function-call-and-return system simply isn't robust enough to communicate errors back to the calling function. To resolve this problem, C++ provides language support for an error-messaging system called *exception handling* that completely bypasses function-call-and-return. If an error occurs inside a function, rather than returning a value, you can report the problem to the exception handling system to jump to the proper error-handling code.

The C++ exception handling system is broken into three parts – `try` blocks, `catch` blocks, and `throw` statements. `try` blocks are simply regions of code designated as areas that runtime errors might occur. To declare a `try` block, you simply write the keyword `try`, then surround the appropriate code in curly braces. For example, the following code shows off a `try` block:

```
try
{
    cout << "I'm in a try block!" << endl;
}
```

Inside of a `try` block, code executes as normal and jumps to the code directly following the `try` block once finished. However, at some point inside a `try` block your program might run into a situation from which it cannot normally recover – for example, a call to `StringToInteger` with an invalid argument. When this occurs, you can report the error by using the `throw` keyword to “throw” the exception into the nearest matching `catch` clause. Like `return`, `throw` accepts a single parameter that indicates an object to throw so that when handling the exception your code has access to extra information about the error. For example, here are three statements that each throw objects of different types:

```
throw 0;                // Throw an int
throw new vector<double>; // Throw a vector<double> *
throw 3.14159;          // Throw a double
```

When you throw an exception, it can be caught by a `catch` clause specialized to catch that error. `catch` clauses are defined like this:

```
catch(ParameterType param)
{
    /* Error-handling code */
}
```

Here, `ParameterType` represents the type of variable this `catch` clause is capable of catching. `catch` blocks must directly follow `try` blocks, and it's illegal to declare one without the other. Since `catch` clauses are specialized for a single type, it's perfectly legal to have cascading `catch` clauses, each designed to pick up a different type of exception. For example, here's code that catches exceptions of type `int`, `vector<int>`, and `string`:

```

try
{
    // Do something
}
catch(int myInt)
{
    // If the code throws an int, execution continues here.
}
catch(const vector<int>& myVector)
{
    // Otherwise, if the code throws a vector<int>, execution resumes here.
}
catch(const string& myString)
{
    // Same for string
}

```

Now, if the code inside the `try` block throws an exception, control will pass to the correct `catch` block. You can visualize exception handling as a room of people and a ball. The code inside the `try` block begins with the ball and continues talking as long as possible. If an error occurs, the `try` block throws the ball to the appropriate `catch` handler, which begins executing.

Let's return to our earlier example with `StringToInteger`. We want to signal an error in case the user enters an invalid parameter, and to do so we'd like to use exception handling. The question, though, is what type of object we should throw. While we can choose whatever type of object we'd like, C++ provides a header file, `<stdexcept>`, that defines several classes that let us specify what error triggered the exception. One of these, `invalid_argument`, is ideal for the situation. `invalid_argument` accepts in its constructor a `string` parameter containing a message representing what type of error occurred, and has a member function called `what` that returns what the error was.* We can thus rewrite the code for `StringToInteger` as

```

int StringToInteger(const string& input)
{
    stringstream converter(input);

    int result; // Try reading an int, fail if we're unable to do so.
    converter >> result;
    if(converter.fail())
        throw invalid_argument("Cannot parse " + input + " as an integer.");

    char leftover; // See if anything's left over. If so, fail.
    converter >> leftover;
    if(!converter.fail())
        return result;
    else
        throw invalid_argument(string("Unexpected character: ") + leftover);
}

```

Notice that while the function itself does not contain a `try/catch` pair, it nonetheless has a `throw` statement. If this statement is executed, then C++ will step backwards through all calling functions until it finds an appropriate `catch` statement. If it doesn't find one, then the program will halt with a runtime error. Now, we can write code using `StringToInteger` that looks like this:

* `what` is a poor choice of a name for a member function. Please make sure to use more descriptive names in your code!


```

try
{
    int result = StringToInteger(myString);
    cout << "The result was: " << result;
}
catch(const invalid_argument& problem)
{
    cout << problem.what() << endl; // Prints out the error message.
}
cout << "Yay! We're done." << endl;

```

Here, if `StringToInteger` encounters an error and throws an exception, control will jump out of the `try` block into the `catch` clause specialized to catch objects of type `invalid_argument`. Otherwise, code continues as normal in the `try` block, then skips over the `catch` clause to print “Yay! We're done.”

There are several things to note here. First, if `StringToInteger` throws an exception, control *immediately* breaks out of the `try` block and jumps to the `catch` clause. Unlike the problems we had with our earlier approach to error handling, here, if there is a problem in the `try` block, we're guaranteed that the rest of the code in the `try` block will not execute, preventing runtime errors stemming from malformed objects. Second, if there is an exception and control resumes in the `catch` clause, once the `catch` block finishes running, control does not resume back inside the `try` block. Instead, control resumes directly following the `try/catch` pair, so the program above will print out “Yay! We're done.” once the `catch` block finishes executing. While this might seem unusual, remember that the reason for exception handling in the first place is to halt code execution in spots where no meaningful operation can be defined. Thus if control leaves a `try` block, chances are that the rest of the code in the `try` could not complete without errors, so C++ does not provide a mechanism for resuming program control. Third, note that we caught the `invalid_argument` exception by reference (`const invalid_argument&` instead of `invalid_argument`). As with parameter-passing, exception-catching can take values either by value or by reference, and by accepting the parameter by reference you can avoid making an unnecessary copy of the thrown object.

A Word on Scope

Exception handling is an essential part of the C++ programming language because it provides a system for recovering from serious errors. As its name implies, exception handling should be used only for *exceptional* circumstances – errors out of the ordinary that necessitate a major change in the flow of control. While you can use exception handling as a fancy form of function call and return, it is highly recommended that you avoid doing so. Throwing an exception is *much* slower than returning a value because of the extra bookkeeping required, so be sure that you're only using exception handling for serious program errors.

Also, the exception handling system will only respond when manually triggered. Unless a code snippet explicitly `throws` a value, a `catch` block cannot respond to it. This means that you cannot use exception handling to prevent your program from crashing from segmentation faults or other pointer-based errors, since pointer errors result in operating-system level process termination, not C++-level exception handling.*

Programming with Exception Handling

While exception handling is a robust and elegant system, it has several sweeping implications for C++ code. Most notably, when using exception handling, unless you are absolutely certain that the classes and functions you use never throw exceptions, you must treat your code as though it might throw an exception at any point. In other words, you can never assume that an entire code block will be completed on its own, and should be pre-

* If you use Microsoft's Visual Studio development environment, you might notice that various errors like null-pointer dereferences and stack overflows result in errors that mention “unhandled exception” in their description. This is a Microsoft-specific feature and is different from C++'s exception-handling system.

pared to handle cases where control breaks out of your functions at inopportune times. For example, consider the following function:

```
void SimpleFunction()
{
    char* myCString = new char[128];
    DoSomething(myCString);
    delete [] myCString;
}
```

Here, we allocate space for a C string, pass it to a function, then deallocate the memory. While this code seems totally safe, when you introduce exceptions into the mix, this code can be very dangerous. What happens, for example, if `DoSomething` throws an exception? In this case, control would jump to the nearest `catch` block and the line `delete [] myCString` would never execute. As a result, our program will leak 128 bytes of memory. If this program runs over a sufficiently long period of time, eventually we will run out of memory and our program will crash.

There are three main ways that we can avoid these problems. First, it's completely acceptable to just avoid exception-handling all together. This approach might seem like a cop-out, but it is a completely valid option that many C++ developers choose. Several major software projects written in C++ do not use exception handling, partially because of the extra difficulties encountered when using exceptions. However, this approach results in code that runs into the same problems discussed earlier in this chapter with `StringToInteger` – functions can only communicate errors through return values and programmers must be extra vigilant to avoid ignoring return values.

The second approach to writing exception-safe code uses a technique called “catch-and-rethrow.” Let's return to the above code example with a dynamically-allocated character buffer. We'd like to guarantee that the C string we've allocated gets deallocated, but as our code is currently written, it's difficult to do so because the `DoSomething` function might throw an exception and interrupt our code flow. If there is an exception, what if we were able to somehow intercept that exception, clean up the buffer, and then propagate the exception outside of the `SimpleFunction` function? From an outside perspective, it would look as if the exception had come from inside the `DoSomething` function, but in reality it would have taken a quick stop inside `SimpleFunction` before proceeding onwards.

The reason this method works is that *it is legal to throw an exception from inside a catch block*. Although `catch` blocks are usually reserved for error handling, there is nothing preventing us from throwing the exception we catch. For example, this code is completely legal:

```
try
{
    try
    {
        DoSomething();
    }
    catch(const invalid_argument& error)
    {
        cout << "Inner block: Error: " << error.what() << endl;
        throw error; // Propagate the error outward
    }
}
catch(const invalid_argument& error)
{
    cout << "Outer block: Error: " << error.what() << endl;
}
```

Here, if the `DoSomething` function throws an exception, it will first be caught by the innermost `try` block, which prints it to the screen. This `catch` handler then throws `error` again, and this time it is caught by the outermost `catch` block.

With this technique, we can almost rewrite our `SimpleFunction` function to look something like this:

```
void SimpleFunction()
{
    char* myCString = new char[128];

    /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
    try
    {
        DoSomething(myCString);
    }
    catch (/* What to catch? */)
    {
        delete [] myCString;
        throw /* What to throw? */;
    }

    /* Note that if there is no exception, we still need to clean things up. */
    delete [] myCString;
}
```

There's a bit of a problem here – what sort of exceptions should we catch? Suppose that we know every sort of exception `DoSomething` might throw. Would it be a good idea to write a `catch` block for each one of these types? At first this may seem like a good idea, but it can actually cause more problems than it solves. First, in each of the `catch` blocks, we'd need to write the same `delete []` statement. If we were to make changes to the `SimpleFunction` function that necessitated more cleanup code, we'd need to make progressively more changes to the `SimpleFunction` `catch` cascade, increasing the potential for errors. Also, if we forget to catch a specific type of error, or if `DoSomething` later changes to throw more types of errors, then we might miss an opportunity to catch the thrown exception and will leak resources. Plus, if we don't know what sorts of exceptions `DoSomething` might throw, this entire approach will not work.

The problem is that in this case, we want to tell C++ to catch *anything* that's thrown as an exception. We don't care about what the type of the exception is, and need to intercept the exception simply to ensure that our resource gets cleaned up. Fortunately, C++ provides a mechanism specifically for this purpose. To catch an exception of any type, you can use the special syntax `catch(...)`, which catches any exception. Thus we'll have the `catch` clause inside `DoSomething` be a `catch(...)` clause, so that we can catch any type of exception that `DoSomething` might throw. But this causes another problem: we'd like to rethrow the exception, but since we've used a `catch(...)` clause, we don't have a name for the specific exception that's been caught. Fortunately, C++ has a special use of the `throw` statement that lets you throw the current exception that's being processed. The syntax is

`throw;`

That is, a lone `throw` statement with no parameters. Be careful when using `throw;`, however, since if you're not inside of a `catch` block the program will crash!

The final version of `SimpleFunction` thus looks like this:

```

void SimpleFunction()
{
    char* myCString = new char[128];

    /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
    try
    {
        DoSomething(myCString);
    }
    catch (...)
    {
        delete [] myCString;
        throw;
    }

    /* Note that if there is no exception, we still need to clean things up. */
    delete [] myCString;
}

```

As you can tell, the “catch-and-throw” approach to exception handling results in code that can be rather complicated. While in some circumstances catch-and-throw is the best option, in many cases there's a much better alternative that results in concise, readable, and thoroughly exception-safe code – object memory management.

Object Memory Management and RAII

C++'s memory model is best described as “dangerously efficient.” Unlike other languages like Java, C++ does not have a garbage collector and consequently you must manually allocate and deallocate memory. At first, this might seem like a simple task – just `delete` anything you allocate with `new`, and make sure not to `delete` something twice. However, it can be quite difficult to keep track of all of the memory you've allocated in a program. After all, you probably won't notice any symptoms of memory leaks unless you run your programs for hours on end, and in all likelihood will have to use a special tool to check memory usage. You can also run into trouble where two objects each point to a shared object. If one of the objects isn't careful and accidentally `delete`s the memory while the other one is still accessing it, you can get some particularly nasty runtime errors where seemingly valid data has been corrupted. The situation gets all the more complicated when you introduce exception-handling into the mix, where the code to `delete` allocated memory might not be reached because of an exception.

In some cases having a high degree of control over memory management can be quite a boon to your programming, but much of the time it's simply a hassle. What if we could somehow get C++ to manage our memory for us? While building a fully-functional garbage collection system in C++ would be just short of impossible, using only basic C++ concepts it's possible to construct an excellent approximation of automatic memory management. The trick is to build *smart pointers*, objects that acquire a resource when created and that clean up the resource when destroyed. That is, when the objects are constructed, they wrap a newly-allocated pointer inside an object shell that cleans up the mess when the object goes out of scope. Combined with features like operator overloading, it's possible to create slick smart pointers that look almost exactly like true C++ pointers, but that know when to free unused memory.

The C++ header file `<memory>` exports the `auto_ptr` type, a smart pointer that accepts in its constructor a pointer to dynamically-allocated memory and whose constructor calls `delete` on the resource.* `auto_ptr` is a template class whose template parameter indicates what type of object the `auto_ptr` will “point” at. For example, an `auto_ptr<string>` is a smart pointer that points to a `string`. Be careful – if you write `auto_ptr<string *>`, you'll end up with an `auto_ptr` that points to a `string *`, which is similar to a `string **`. Through the magic of operator overloading, you can use the regular dereference and arrow operat-

* Note that `auto_ptr` calls `delete`, not `delete []`, so you cannot store dynamically-allocated arrays in `auto_ptr`. If you want the functionality of an array with automatic memory management, use a `vector`.

ors on an `auto_ptr` as though it were a regular pointer. For example, here's some code that dynamically allocates a `vector<int>`, stores it in an `auto_ptr`, and then adds an element into the vector:

```
/* Have the auto_ptr point to a newly-allocated vector<int>. The constructor
 * is explicit, so we must use parentheses.
 */
auto_ptr<vector<int> > managedVector(new vector<int>);
managedVector->push_back(137); // Add 137 to the end of the vector.
(*managedVector)[0] = 42; // Set element 0 by dereferencing the pointer.
```

While in many aspects `auto_ptr` acts like a regular pointer with automatic deallocation, `auto_ptr` is fundamentally different from regular pointers in assignment and initialization. Unlike objects you've encountered up to this point, assigning or initializing an `auto_ptr` to hold the contents of another destructively modifies the source `auto_ptr`. Consider the following code snippet:

```
auto_ptr<int> one(new int);
auto_ptr<int> two;
two = one;
```

After the final line executes, `two` will hold the resource originally owned by `one`, and `one` will be empty. During the assignment, `one` relinquished ownership of the resource and cleared out its state. Consequently, if you use `one` from this point forward, you'll run into trouble because it's not actually holding a pointer to anything. While this is highly counterintuitive, it has several advantages. First, it ensures that there can be at most one `auto_ptr` to a resource, which means that you don't have to worry about the contents of an `auto_ptr` being cleaned up out from underneath you by another `auto_ptr` to that resource. Second, it means that it's safe to return `auto_ptr`s from functions without the resource getting cleaned up. When returning an `auto_ptr` from a function, the original copy of the `auto_ptr` will transfer ownership to the new `auto_ptr` during return-value initialization, and the resource will be transferred safely.* Finally, because each `auto_ptr` can assume that it has sole ownership of the resource, `auto_ptr` can be implemented extremely efficiently and has almost zero overhead.

As a consequence of the “`auto_ptr` assignment is transference” policy, you must be careful when passing an `auto_ptr` by value to a function. Since the parameter will be initialized to the original object, it will empty the original `auto_ptr`. Similarly, you should not store `auto_ptr`s in STL containers, since when the containers reallocate or balance themselves behind the scenes they might assign `auto_ptr`s around in a way that will trigger the object destructors.

For reference, here's a list of the member functions of the `auto_ptr` template class:

* For those of you interested in programming language design, C++ uses what's known as *copy semantics* for most of its operations, where assigning objects to one another creates copies of the original objects. `auto_ptr` seems strange because it uses *move semantics*, where assigning `auto_ptr`s to one another transfers ownership of some resource. Move semantics are not easily expressed in C++ and the code to correctly implement `auto_ptr` is surprisingly complex and requires an intricate understanding of the C++ language. The next revision of C++, C++0x, will add several new features to the language to formalize and simply move semantics and will replace `auto_ptr` with `unique_ptr`, which formalizes the move semantics.

<code>explicit auto_ptr (Type* resource)</code>	<code>auto_ptr<int> ptr(new int);</code> Constructs a new <code>auto_ptr</code> wrapping the specified pointer, which must be from dynamically-allocated memory.
<code>auto_ptr(auto_ptr& other)</code>	<code>auto_ptr<int> one(new int);</code> <code>auto_ptr<int> two = one;</code> Constructs a new <code>auto_ptr</code> that acquires resource ownership from the <code>auto_ptr</code> used in the initialization. Afterwards, the old <code>auto_ptr</code> will not encapsulate any dynamically-allocated memory.
<code>T& operator *() const</code>	<code>*myAutoPtr = 137;</code> Dereferences the stored pointer and returns a reference to the memory it's pointing at.
<code>T* operator-> () const</code>	<code>myStringAutoPtr->append("C++!");</code> References member functions of the stored pointer.
<code>T* release()</code>	<code>int *regularPtr = myPtr.release();</code> Relinquishes control of the stored resource and returns it so it can be stored in another location. The <code>auto_ptr</code> will then contain a <code>NULL</code> pointer and will not manage the memory any more.
<code>void reset(T* ptr = NULL)</code>	<code>myPtr.reset();</code> <code>myPtr.reset(new int);</code> Releases any stored resources and optionally stores a new resource inside the <code>auto_ptr</code> .
<code>T* get() const</code>	<code>SomeFunction(myPtr.get()); // Retrieve stored resource</code> Returns the stored pointer. Useful for passing the managed resource to other functions.

Of course, dynamically-allocated memory isn't the only C++ resource that can benefit from object memory management. For example, when working with OS-specific libraries like Microsoft's Win32 library, you will commonly have to manually manage handles to system resources. In spots like these, writing wrapper classes that act like `auto_ptr` but that do cleanup using methods other than a plain `delete` can be quite beneficial. In fact, the system of having objects manage resources through their constructors and destructors is commonly referred to as *resource acquisition is initialization*, or simply *RAII*.

Exceptions and Smart Pointers

Up to this point, smart pointers might seem like a curiosity, or perhaps a useful construct in a limited number of circumstances. However, when you introduce exception handling to the mix, smart pointers will be invaluable. In fact, in professional code where exceptions can be thrown at almost any point, smart pointers are almost as ubiquitous as regular C++ pointers.

Let's suppose you're given the following linked list cell struct:

```
struct nodeT
{
    int data;
    nodeT *next;
};
```

Now, consider this function:

```
nodeT* GetNewCell()
{
    nodeT* newCell = new nodeT;
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell;
}
```

This function allocates a new `nodeT` cell, then tells it to hold on to the value returned by `SomeComplicatedFunction`. If we ignore exception handling, this code is totally fine, provided of course that the calling function correctly holds on to the `nodeT *` pointer we return. However, when we add exception handling to the mix, this function is a recipe for disaster. What happens if `SomeComplicatedFunction` throws an exception? Since `GetNewCell` doesn't have an associated `try` block, the program will abort `GetNewCell` and search for the nearest `catch` clause. Once the `catch` finishes executing, we have a problem – we allocated a `nodeT` object, but we didn't clean it up. Worse, since `GetNewCell` is no longer running, we've lost track of the `nodeT` entirely, and the memory is orphaned.

Enter `auto_ptr` to save the day. Suppose we change the declaration `nodeT* newCell` to `auto_ptr<nodeT> newCell`. Now, if `SomeComplicatedFunction` throws an exception, we won't leak any memory since when the `auto_ptr` goes out of scope, it will reclaim the memory for us. Wonderful! Of course, we also need to change the last line from `return newCell` to `return newCell.release()`, since we promised to return a `nodeT *`, not an `auto_ptr<nodeT>`. The new code is printed below:

```
nodeT* GetNewCell()
{
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell.release(); // Tell the auto_ptr to stop managing memory.
}
```

This function is now wonderfully exception-safe thanks to `auto_ptr`. Even if we prematurely exit the function from an exception in `SomeComplicatedFunction`, the `auto_ptr` destructor will ensure that our resources are cleaned up. However, we can make this code even safer by using the `auto_ptr` in yet another spot. What happens if we call `GetNewCell` but don't store the return value anywhere? For example, suppose we have a function like this:

```
void SillyFunction()
{
    GetNewCell(); // Oh dear, there goes the return value.
}
```

When we wrote `GetNewCell`, we tacitly assumed that the calling function would hold on to the return value and clean the memory up at some later point. However, it's totally legal to write code like `SillyFunction` that calls `GetNewCell` and entirely discards the return value. This leads to memory leaks, the very problem we were trying to solve earlier. Fortunately, through some creative use of `auto_ptr`, we can eliminate this problem. Consider this modified version of `GetNewCell`:

```

auto_ptr<nodeT> GetNewCell()
{
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell; // See below
}

```

Here, the function returns an `auto_ptr`, which means that the returned value is itself managed. Now, if we call `SillyFunction`, even though we didn't grab the return value of `GetNewCell`, because `GetNewCell` returns an `auto_ptr`, the memory will still get cleaned up.

More to Explore

Exception-handling and RAII are complex topics that have impressive ramifications for the way that you write C++ code. However, we simply don't have time to cover every facet of exception handling. In case you're interested in exploring more advanced topics in exception handling and RAII, consider looking into the following:

1. **The Standard Exception Classes:** In this chapter we discussed `invalid_argument`, one of the many exception classes available in the C++ standard library. However, there are several more exception classes that form an elaborate hierarchy. Consider reading into some of the other classes – some of them even show up in the STL!
2. **Exception Specifications.** Because functions can throw exceptions at any time, it can be difficult to determine which pieces of code can and cannot throw exceptions. Fortunately, C++ has a feature called an *exception specification* which indicates what sorts of exceptions a function is allowed to throw. When an exception leaves a function with an exception specification, the program will abort unless the type of the exception is one of the types mentioned in the specification.
3. **Function `try` Blocks.** There is a variant of a regular try block that lets you put the entire contents of a function into a try/catch handler pair. However, it is a relatively new feature in C++ and is not supported by several popular compilers. Check a reference for more information.
4. **`new` and Exceptions.** If your program runs out of available memory, the `new` operator will indicate a failure by throwing an exception of type `bad_alloc`. When designing custom container classes, it might be worth checking against this case and acting accordingly.
5. **The Boost Smart Pointers:** While `auto_ptr` is useful in a wide variety of circumstances, in many aspects it is limited. Only one `auto_ptr` can point to a resource at a time, and `auto_ptr`s cannot be stored inside of STL containers. The Boost C++ libraries consequently provide a huge number of smart pointers, many of which employ considerably more complicated resource-management systems than `auto_ptr`. Since many of these smart pointers are likely to be included in the next revision of the C++ standard, you should be sure to read into them.

Bjarne Stroustrup (the inventor of C++) wrote an excellent introduction to exception safety, focusing mostly on implementations of the C++ Standard Library. If you want to read into exception-safe code, you can read it online at http://www.research.att.com/~bs/3rd_safe.pdf. Additionally, there is a most excellent reference on `auto_ptr` available at http://www.gotw.ca/publications/using_auto_ptr_effectively.htm that is a great resource on the subject.

Practice Problems

1. What happens if you put a `catch(...)` handler at the top of a catch cascade? ♦
2. Explain why the `auto_ptr` constructor is marked `explicit`. (*Hint: Give an example of an error you can make if the constructor is not marked explicit*).
3. The `SimpleFunction` function from earlier in this chapter ran into difficulty with exception-safety because it relied on a manually-managed C string. Explain why this would not be a problem if it instead used a C++ string.
4. Consider the following C++ function:

```
void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    /* This might throw an exception! */
    DoSomething(myStack);

    myStack.push(topElem);
}
```

This function accepts as input a C++ `stack<string>`, pops off the top element, calls the `DoSomething` function, then pushes the element back on top. Provided that the `DoSomething` function doesn't throw an exception, this code will guarantee that the top element of the `stack` does not change before and after the function executes. Suppose, however, that we wanted to absolutely guarantee that the top element of the stack never changes, even if the function throws an exception. Using the catch-and-rethrow strategy, explain how to make this the case. ♦

5. Write a class called `AutomaticStackManager` whose constructor accepts a `stack<string>` and pops off the top element (if one exists) and whose destructor pushes the element back onto the `stack`. Using this class, rewrite the code in Problem 4 so that it's exception safe. How does this version of the code compare to the approach using catch-and-rethrow? ♦
6. Write a template class `EnsureCalled` parameterized over an arbitrary nullary function type (i.e. zero-parameter function) whose constructor accepts a nullary function and whose destructor calls this function. This is a generalization of `AutomaticStackManager` class you just wrote. Should `EnsureCalled` have copy semantics, or should it be uncopyable? Why?

Chapter 28: Extended Example: Gauss-Jordan Elimination

Linear equations and matrices are at the heart of many techniques in computer science. Markov processes, which underlie Google's PageRank algorithm, can be described and manipulated using matrices, and modern 3D graphics use matrices extensively to transform points in space into pixels on a screen.

In this extended example, we'll explore systems of linear equations and write a program to solve them using *Gauss-Jordan elimination*. We'll also see how to use this technique to efficiently invert arbitrary square matrices. In the process, we'll get the chance to play around with the STL `<functional>` library, our homegrown `grid` class, and even a bit of exception handling.

Systems of Linear Equations

A *linear equation* is a sum or difference of terms where every term is either a scalar or the product of a variable and a scalar. For example, the formula for a line in two-dimensional space, $y = mx + b$, is a linear equation with variables x and y and constants m and b ; this is the reason for the terminology “linear equation.” The formula $x^2 + y^2 = r^2$ is nonlinear because x^2 and y^2 are variables raised to the second power, and the formula $xy = k$ is nonlinear because xy is a product of two variables. A *solution* to a linear equation is a set of values for each of the variables in the equation that make the equation hold. For example, given the linear equation $3x + 2y + z = 0$, one solution is $x = -1, y = 2, z = -1$ since $3(-1) + 2(2) + (-1) = -3 + 4 - 1 = 0$. When the variables in the system refer to coordinate axes, solutions are sometimes written as points in space. Thus one solution to the above equation is $(-1, 2, -1)$ and another is $(0, 0, 0)$.

A *system of linear equations* is a collection of linear equations over the same variables. A solution to a system of linear equations is a set of values for each of the variables that is a solution to each of the equations in the system. For example, consider the following system of equations:

$$\begin{aligned}y &= 3x + 4 \\ y &= 7x - 12\end{aligned}$$

The point $(2, 10)$ is a solution to the first equation in this system but is not a solution to the second, so it is not a solution to the system. Similarly, $(0, 0)$ is not a solution to the system because it is not a solution to either of the equations. However, $(4, 16)$ is a solution to the system, since it solves both the first and the second equation.

The *solution set* to a system of equations is the set of all solutions to that system. In the above example, the solution set only contains one element: $(4, 16)$. Some systems of equations have no solution at all, such as this one:

$$\begin{aligned}y &= x + 3 \\ y &= x + 4\end{aligned}$$

In this case the solution set is the empty set, $\emptyset$.

Other systems of equations have infinitely many solutions. For example, any point on the x axis is a solution to these equations:

$$\begin{aligned}x + y + 2z &= 0 \\ -2y - 4z &= 0\end{aligned}$$

Solving Systems of Linear Equations

There are infinitely many combinations of values we could assign to variables in a linear equation, so how can efficiently compute which combinations are solutions? Fortunately, there are many techniques for solving systems of linear equations, most of which work by manipulating the equations to derive a minimal set of constraints on valid solutions. Before describing and implementing one particular technique called *Gauss-Jordan elimination*, let us first examine how we can manipulate systems of linear equations. To motivate this example, let's suppose that we have the following system of linear equations:

$$\begin{aligned} 3x + y - 7z &= 11 \\ x + y - 3z &= 5 \end{aligned}$$

There are many ways that we can modify this system of equations without changing the solution set. For instance, we can multiply both sides of one of the equations by a nonzero constant without affecting whether a particular point is a solution. As an example, the above system is equivalent to the system given below since we have simply multiplied both sides of the bottom equation by -3 :

$$\begin{aligned} 3x + y - 7z &= 11 \\ -3x - 3y + 9z &= -15 \end{aligned}$$

Similarly, we can add one equation to another without changing the solution set. Thus

$$\begin{aligned} 3x + y - 7z &= 11 \\ -3x - 3y + 9z &= -15 \end{aligned}$$

Is equivalent to

$$\begin{aligned} -2y + 2z &= -4 \\ -3x - 3y + 9z &= -15 \end{aligned}$$

Because we have added the second equation to the first.

Using these two rules, we can greatly simplify a system until we reach a state where the solution can be derived easily. In our particular example, we can simplify as follows:

$$\begin{aligned} -2y - 2z &= -4 \\ -3x - 3y - 9z &= -15 \end{aligned}$$

$$\begin{aligned} -2y + 2z &= -4 \\ x + y - 3z &= 5 \end{aligned} \quad \text{(Multiplying the second equation by } -1/3 \text{)}$$

$$\begin{aligned} -y + z &= -2 \\ x + y - 3z &= 3 \end{aligned} \quad \text{(Multiplying the first equation by } 1/2 \text{)}$$

$$\begin{aligned} -y + z &= -2 \\ x - 2z &= 3 \end{aligned} \quad \text{(Adding the first and second equations)}$$

$$\begin{aligned} y - z &= 2 \\ x - 2z &= 3 \end{aligned} \quad \text{(Multiplying the first equation by } -1 \text{)}$$

At this point we cannot eliminate the x from the second equation or y from the first and thus cannot proceed any further. We could eliminate z from one of the equations, but only if we reintroduce x or y to an equation from which it has been eliminated. If we now solve for x and y , we have the following:

$$\begin{aligned}y &= 2 + z \\x &= 3 + 2z\end{aligned}$$

Now, consider any point (x, y, z) that is a solution to the system of linear equations. By the above equations, we have that $(x, y, z) = (3 + 2z, 2 + z, z) = (3, 2, 0) + (2z, z, z) = (3, 2, 0) + z(2, 1, 1)$. This first term is the point $(3, 2, 0)$ in three-dimensional space and the second the vector $(2, 1, 1)$ scaled by some value z . As we vary z , we can obtain more and more solutions to this system of equations. In fact, the set of solutions to this system is given by $\{ (3, 2, 0) + z(2, 1, 1) \mid z \text{ is a real number} \}$.

Matrix Representations

Any linear equation containing variables $x_0, x_1, x_2, \dots, x_n$ can be rearranged to an equivalent linear equation of the form

$$a_0x_0 + a_1x_1 + a_2x_2 + \dots + a_nx_n = c$$

A system of m linear equations over variables $x_0, x_1, x_2, \dots, x_n$ can thus be written as

$$\begin{aligned}a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \dots + a_{0n}x_n &= c_0 \\a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= c_1 \\&\dots \\a_{m0}x_0 + a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= c_m\end{aligned}$$

If we try solving a large system of equations using the above techniques, it can be tedious to write out every equation at each step. Mathematicians and computer scientists are inherently lazy, and rather than writing out these large systems instead use *matrices*, two-dimensional grids of numbers corresponding to the coefficients and constants in a system of equations. For example, for the above system of equations, we would write out the following matrix:

$$\left(\begin{array}{cccc|c} a_{00} & a_{01} & \cdots & a_{0n} & c_0 \\ a_{10} & a_{11} & \cdots & a_{1n} & c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m0} & a_{m1} & \cdots & a_{mn} & c_m \end{array} \right)$$

This is technically called an *augmented matrix* because of the coefficients in the final column. We will refer to this column as the *augmented column*.

To give a concrete example of a matrix representation, the system of equations defined by

$$\begin{aligned}x + 2y + 3z &= 4 \\-x + y &= -3 \\4x + 4y - 5z &= 1\end{aligned}$$

would be represented as the augmented matrix

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right)$$

Because a matrix is simply a more condensed notation for a system of linear equations, we can solve the linear system represented in matrix form using the same set of transformations we could apply directly to equations. For example, we can multiply any row in a matrix by a nonzero constant, as shown here:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right) \Rightarrow \left(\begin{array}{ccc|c} \mathbf{2} & \mathbf{4} & \mathbf{6} & \mathbf{8} \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right)$$

We can similarly add one row to another, as shown here (adding the second row to the first)

$$\left(\begin{array}{ccc|c} 2 & 4 & 6 & 8 \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right) \Rightarrow \left(\begin{array}{ccc|c} \mathbf{1} & \mathbf{5} & \mathbf{6} & \mathbf{5} \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right)$$

Finally, we can swap any two rows. This corresponds to writing the equations in a different order than before.

$$\left(\begin{array}{ccc|c} 1 & 5 & 6 & 5 \\ -1 & 1 & 0 & -3 \\ 4 & 4 & -5 & 1 \end{array} \right) \Rightarrow \left(\begin{array}{ccc|c} -1 & \mathbf{1} & \mathbf{0} & -3 \\ \mathbf{1} & \mathbf{5} & \mathbf{6} & \mathbf{5} \\ 4 & 4 & -5 & 1 \end{array} \right)$$

Reduced Row Echelon Form

Before introducing Gauss-Jordan elimination, we need to cover two more pieces of terminology – *pivots* and *reduced row echelon form*. An element in a matrix is called a *pivot element* if it is the first nonzero element in its row and the only nonzero value in its column. By convention, values in the augmented column of the matrix are not considered pivot elements. To give a concrete example, consider the following matrix:

$$\left(\begin{array}{cccc|c} 3 & 0 & 2 & 8 & 0 \\ 0 & 5 & 0 & 5 & 9 \\ 0 & 0 & 0 & 3 & 0 \end{array} \right)$$

The three in the first column is a pivot since it is the first nonzero element in its row and is the only nonzero element in its column. Similarly, the first five in the second row is a pivot. The two in the first row is *not* a pivot because it is not the first element in its row, and the three in the bottom row is not a pivot because it is not the only nonzero element in its column.

An important point to note is that there cannot be any more pivots in a matrix than there are rows or columns. Thus a 4x3 matrix can only have three pivots, while a 7x2 matrix can only have two. This is because no row or column can contain more than one pivot, since pivots must come first in their row and must be the only nonzero entry in a column.

A matrix is in *reduced row echelon form* (rref) if the following two properties hold:

- The first nonzero element of each row is a pivot with value 1.
- The first nonzero element of each row is to the right of the first nonzero element of the row above it.

For example, the following matrix is in rref:

$$\left(\begin{array}{cccc|c} 1 & 3 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

This matrix, however, is not because the pivot on the second row is to the left of the pivot on the first row:

$$\left(\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 1 \\ 1 & 3 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

Swapping the first and second rows of this matrix will convert it to rref, however. Similarly, the matrix below is not in rref because the second row does not begin with a pivot:

$$\left(\begin{array}{cccc|c} 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

If we subtract the second row from the first, though, we get this matrix:

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & -2 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

Which is in rref. Finally, the following matrix is not in rref because the pivot on the last row is not equal to 1:

$$\left(\begin{array}{cccc|c} 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 6 \end{array}\right)$$

Dividing that row by 3 will put the matrix into rref:

$$\left(\begin{array}{cccc|c} 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 \end{array}\right)$$

We will not prove it here, but every matrix can be reduced to a unique matrix in rref using the three manipulations listed above. The procedure for performing this conversion is Gauss-Jordan elimination, discussed later.

What's so special about reduced row echelon form? Intuitively, the rref version of a matrix is the most simplified version of the matrix possible. We cannot learn any more about the system by performing additional manipulations, and the relations between variables are as simple as possible. This means that we can easily extract the solutions (if any) to the system of equations from the matrix without much effort. Recall that there are three possible outcomes to a system of equations – there is either *no solution*, *one solution*, or *infinitely many solutions*. These correspond to rref matrix forms as follows:

- *No solution.* Then the rref form of the matrix will have a row where the body of the matrix consists solely of zeros with a nonzero coefficient. If we convert the matrix back into a system of equations, this corresponds to the equation $0 = k$ for some nonzero k , which is clearly impossible. For example, this rref matrix corresponds to a system with no solutions:

$$\left(\begin{array}{cccc|c} 1 & 0 & 2 & -7 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{array} \right)$$

- *One solution.* Then every column in the rref matrix contains a pivot. These correspond to equations of the form $x_0 = c_0, x_1 = c_1, \dots, x_n = c_n$. The solution to the system can then be read off from the final column. For example:

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right)$$

Note that if the coefficient in the bottom row of this matrix were nonzero, the system would have no solution.

- *Infinitely many solutions.* Then some column does not contain a pivot. This corresponds to a variable whose value is either unconstrained or defined in terms of some other variable in the system. For example:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 1 \end{array} \right)$$

Gauss-Jordan Elimination

Gauss-Jordan elimination is a modified form of Gaussian elimination* that converts a matrix into its reduced row echelon form equivalent. The technique hinges on the fact that the first nonzero element of each row must be a pivot, which must have value 1 and be the only nonzero value in its column. The algorithm walks across the columns of the matrix, trying to clear the column out so that the only remaining element is a pivot. After clearing all of the columns or creating the maximum possible number of pivots, the algorithm terminates. During its execution, Gauss-Jordan elimination partitions matrices into two sections – an upper section consisting of rows that begin with a pivot and a lower section that has not yet been analyzed.

To see how Gauss-Jordan elimination works, we'll use the algorithm to convert the following matrix to rref:

$$\left(\begin{array}{cccc|c} 0 & 5 & 10 & -1 & -7 \\ 2 & 6 & 12 & 10 & 20 \\ 2 & 3 & 6 & 11 & 25 \end{array} \right)$$

We begin by trying to put a pivot into the first column of the matrix. This requires us to find a row with a nonzero value in the first column. Let's choose the second row. For an element to be a pivot, it has to have value 1, so we'll normalize the second row by dividing it by two. This yields

* Gaussian Elimination converts a matrix to *row echelon form* rather than *reduced row echelon form*.

$$\left(\begin{array}{cccc|c} 0 & 5 & 10 & -1 & -7 \\ \mathbf{1} & \mathbf{3} & \mathbf{6} & \mathbf{5} & \mathbf{10} \\ 2 & 3 & 6 & 11 & 25 \end{array}\right)$$

Now, for the this 1 to be a pivot, every other element in the column must be zero. Since the final row of this matrix contains a two in this column, we will need to use our matrix operations to clear this value. We can accomplish this by normalizing the row so that the row has a 1 in the current column, then subtracting the second row out from the third. Since the second row has a 1 in the first column, this will result with the first column of the third row containing a zero. To see how this works, let's begin by dividing the third row by two, as seen here:

$$\left(\begin{array}{cccc|c} 0 & 5 & 10 & -1 & -7 \\ 1 & 3 & 6 & 5 & 10 \\ \mathbf{1} & \mathbf{3/2} & \mathbf{3} & \mathbf{11/2} & \mathbf{25/2} \end{array}\right)$$

Now that the row has a one in the first position, we subtract the second row from the third. This puts a zero into the first column, making our 1 a pivot:

$$\left(\begin{array}{cccc|c} 0 & 5 & 10 & -1 & -7 \\ 1 & 3 & 6 & 5 & 10 \\ \mathbf{0} & \mathbf{-3/2} & \mathbf{-3} & \mathbf{1/2} & \mathbf{5/2} \end{array}\right)$$

The first column now contains a pivot, so we're one step closer to having the matrix in rref. Recall that the definition of rref requires the pivot in each row to be further to the right than the pivot in the row above it. To maintain this property, we'll thus move this row up to the top by swapping it with the first row. This results in the following matrix:

$$\left(\begin{array}{cccc|c} \mathbf{1} & \mathbf{3} & \mathbf{6} & \mathbf{5} & \mathbf{10} \\ 0 & 5 & 10 & -1 & -7 \\ 0 & -3/2 & -3 & 1/2 & 5/2 \end{array}\right)$$

As mentioned earlier, Gauss-Jordan elimination partitions the matrix into an upper group where the pivots are in place and a lower group that has not yet been analyzed. These groups are shown above, where the bold rows have been processed and the unbolded rows have not.

Now that we've finished the first column, let's move on to the second. We want to place a pivot into this column, so we need to find a row that can contain this pivot. Since each row can contain at most one pivot, this means that we'll only look at the bottom two rows. Both the second and third rows have a nonzero value in the second column and could be chosen as the row to contain the pivot, so we'll arbitrarily choose the second row to contain the pivot. As with the first row, we'll normalize this row by dividing by five, as shown here:

$$\left(\begin{array}{cccc|c} 1 & 3 & 6 & 5 & 10 \\ 0 & \mathbf{1} & \mathbf{2} & \mathbf{-1/5} & \mathbf{-7/5} \\ 0 & -3/2 & -3 & 1/2 & 5/2 \end{array}\right)$$

Now, we need to clear out the rest of the column. The top row has a three in this position that we need to eliminate. We begin by dividing the first row by three, as shown here:

$$\left(\begin{array}{cccc|c} \mathbf{1/3} & \mathbf{1} & \mathbf{2} & \mathbf{5/3} & \mathbf{10/3} \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & -3/2 & -3 & 1/2 & 5/2 \end{array} \right)$$

Next, we subtract out the second row, as shown here:

$$\left(\begin{array}{cccc|c} \mathbf{1/3} & \mathbf{0} & \mathbf{0} & \mathbf{28/15} & \mathbf{71/15} \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & -3/2 & -3 & 1/2 & 5/2 \end{array} \right)$$

We've cleared out the element in the second position, but in doing so made the first row no longer begin with a 1. Thus we'll multiply the top row by 3 so that its first entry is a 1. This is shown here:

$$\left(\begin{array}{cccc|c} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{28/5} & \mathbf{71/5} \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & -3/2 & -3 & 1/2 & 5/2 \end{array} \right)$$

We've restored the first row and now need to repeat this process on the bottom row. We multiply every element in the row by $^{-2/3}$ to normalize it:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 28/5 & 71/5 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & \mathbf{1} & \mathbf{-2} & \mathbf{-1/3} & \mathbf{-5/3} \end{array} \right)$$

We then subtract out the second row from the third:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 28/5 & 71/5 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{-2/15} & \mathbf{-4/15} \end{array} \right)$$

The second column now contains a pivot. The matrix below again shows what rows have been processed, with rows in bold corresponding to those that have been considered and unbolded rows those that have not.

$$\left(\begin{array}{cccc|c} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{28/5} & \mathbf{71/5} \\ \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{-1/5} & \mathbf{-7/5} \\ 0 & 0 & 0 & -2/15 & -4/15 \end{array} \right)$$

Let's now try to put a pivot into the third column. Since the first two rows already contain pivots, they cannot store any more pivots, so if we are going to have a pivot in the third column the pivot will have to be in the last row. However, the last row has a zero in the second position, meaning that we can't convert it into a pivot. We're therefore done with the third column and can move on to the last column. Again, we want to put a pivot into this column, and again the first two rows are used up. We thus try to convert the third row so that it contains a pivot in this column. We normalize the row by multiplying by $^{-15/2}$ to yield

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 28/5 & 71/5 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{2} \end{array} \right)$$

To clear this column in the first row, we normalize it by dividing by $^{28/5}$:

$$\left(\begin{array}{cccc|c} 5/28 & 0 & 0 & 1 & 71/28 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

Then subtract by the last row:

$$\left(\begin{array}{cccc|c} 5/28 & 0 & 0 & 0 & 15/28 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

And then normalizing by dividing by $5/28$:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 2 & -1/5 & -7/5 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

Repeating this process again on the second row, we normalize the second row by dividing by $-1/5$:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & -5 & -10 & 1 & 7 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

Subtracting out the last row:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & -5 & -10 & 0 & 5 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

And normalizing:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 2 & 0 & -1 \\ 0 & 0 & 0 & 1 & 2 \end{array} \right)$$

This matrix is now in reduced row echelon form and the algorithm terminates. We've successfully reduced this matrix to rref.

Here's a quick summary of how the algorithm works:

Starting in the first column and continuing rightward:

- Look for a row that doesn't already contain a pivot with a nonzero value in this column.
- Normalize that row so that the value in the current column is one.
- For each other row with a nonzero value in this column:
 - Normalize the row so that the value in the current column is one.
 - Subtract out the row containing the new pivot.
 - Normalize the row to restore any pivots it contains.
- Move the current row above all other rows that don't already contain pivots.

Implementing Gauss-Jordan Elimination

Using the C++ techniques we've covered so far, we can write an elegant implementation of the Gauss-Jordan elimination algorithm. For our implementation we'll represent a matrix as a `grid<double>`, using the implementation of `grid` we wrote several chapters ago. To refresh your memory, the `grid` interface is as follows:

```
template <typename ElemType> class grid
{
public:
    /* Constructors create a grid of the specified size. */
    grid();
    grid(int numRows, int numCols);

    /* Resizing functions. */
    void clear();
    void resize(int width, int height);

    /* Size queries. */
    int numRows() const;
    int numCols() const;
    bool empty() const;
    int size() const;

    /* Element access. */
    ElemType& getAt(int row, int col);
    const ElemType& getAt(int row, int col) const;

    /* Iterator definitions. */
    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;

    /* Container iteration. */
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    /* Row iteration. */
    iterator row_begin(int row);
    iterator row_end(int row);
    const_iterator row_begin(int row) const;
    const_iterator row_end(int row) const;

    /* Proxy objects for operator[] */
    class MutableReference { /* ... */ };
    class ImmutableReference { /* ... */ };

    /* Element selection functions. */
    MutableReference operator[](int row);
    ImmutableReference operator[](int row) const;

    /* Relational operators. */
    bool operator < (const grid& other) const;
    bool operator <= (const grid& other) const;
    bool operator == (const grid& other) const;
    bool operator != (const grid& other) const;
    bool operator >= (const grid& other) const;
    bool operator > (const grid& other) const;
};
```

For our purposes, we'll assume that the last column of the `grid<double>` represents the augmented column. We'll implement Gauss-Jordan Elimination as a function called `GaussJordanEliminate`, as shown here:

```
void GaussJordanEliminate(grid<double>& matrix)
{
    /* ... */
}
```

The algorithm works by iterating across the columns of the matrix placing pivots into the proper position. We can thus start our implementation off as follows:

```
void GaussJordanEliminate(grid<double>& matrix)
{
    for(int col = 0; col < matrix.numCols() - 1; ++col)
    {
        /* ... */
    }
}
```

Notice that we iterate not up to `matrix.numCols()` but rather to `matrix.numCols() - 1`. This is deliberate: the last column of the matrix is the augmented column and is not technically part of the matrix body.

Recall that the number of pivots in a matrix cannot exceed the number of rows or columns. We will thus keep track of not only the current column, but also how many pivots we have placed. We will break out of the loop either when we've finished processing the final column or when we've placed the maximum number of pivots, as shown here:

```
void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        /* ... */
    }
}
```

Throughout this implementation, we'll maintain the invariant that the first `pivotsPlaced` rows of the matrix have already been processed and begin with pivots.

Now, we need to place a pivot in the current column of the matrix. We can't place the pivot into a row that already contains one, and since the first `pivotsPlaced` rows already contain pivots we need to look in rows numbered between `pivotsPlaced` and `matrix.numRows()` for a nonzero value to convert to a pivot. For simplicity, we'll decompose out the code for finding a row satisfying this property to a helper function called `FindPivotRow`, as shown here:

```
int FindPivotRow(const grid<double>& matrix, int col, int pivotsPlaced)
{
    for(int row = pivotsPlaced; row < matrix.numRows(); ++row)
        if(matrix[row][col] != 0.0) return row;
    return -1;
}
```

This function returns the index of a non-pivot row containing a nonzero value in the `col`-th column, or `-1` if no row could be found. We can use this function in `GaussJordanEliminate` as follows:

```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        /* ... */
    }
}

```

We now have the row that will house the new pivot, and now it's time to normalize the row so that the element at position `(row, col)` is 1. While we could do this using a simple `for` loop, this is a great chance to show off our skills with the STL algorithms and the `<functional>` library. Recall the `transform` algorithm, which has the following signature:

```

template <typename InputIterator,
          typename OutputIterator,
          typename UnaryFunction>
OutputIterator transform(InputIterator begin, InputIterator end,
                        OutputIterator out,
                        UnaryFunction fn);

```

`transform` applies the function `fn` to each element in the range `[begin, end)` and stores the resulting values in the range beginning with `out`. As we discussed in the chapter on algorithms, the output iterator `out` can point back to the input range, in which case `transform` will update all of the values in the range by applying `fn` to them. If we can construct a function that normalizes every element in the range, then we can use `transform` to update the entire row. Fortunately, we do have such a function thanks to the STL `<functional>` library. We want to normalize the row by dividing every element in the row by the element that will be the pivot. Since the pivot is stored in `matrix[row][col]`, we can create a function that divides by the pivot by calling

```
bind2nd(divides<double>(), matrix[row][col])
```

`divides<double>` is an adaptable binary functor exported by `<functional>` that takes two doubles and returns their quotient. Binding the second parameter with the value of `matrix[row][col]` thus constructs a unary function that divides its argument by `matrix[row][col]`.

Using this bit of STL power, we can normalize the row in a single line of code as follows:

```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        transform(matrix.row_begin(row), matrix.row_end(row),
                  matrix.row_begin(row), // Overwrite row
                  bind2nd(divides<double>(), matrix[row][col]));

        /* ... */
    }
}

```

Recall that the `row_begin` and `row_end` functions on the `grid` return iterators that define a single row of the `grid`. Now that this has been taken care of, let's move on to the next step – clearing out the rest of the column. We do this by iterating over each other row, checking if the row has a nonzero value in the current column. If so, we normalize the row so that the element in the pivot column is one, subtract out the current row, then renormalize the row to restore any pivots. We'll begin by looking for rows that need to be cleared:

```
void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        transform(matrix.row_begin(row), matrix.row_end(row),
                  matrix.row_begin(row), // Overwrite row
                  bind2nd(divides<double>(), matrix[row][col]));

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            /* ... */
        }
    }
}
```

Now that we've found a row to clear, we need to normalize the row by dividing every value in the row by the value in the current column. We've already figured out how to do this using `transform` and `bind2nd`, so to avoid code duplication we'll decompose out the logic to divide the row by a value into a helper function called `DivideRow`. This is shown here:

```
void DivideRow(grid<double>& matrix, int row, double by)
{
    transform(matrix.row_begin(row), matrix.row_end(row),
              matrix.row_begin(row),
              bind2nd(divides<double>(), by);
}
```

And can then update the existing code to use this function:

```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            /* ... */
        }
    }
}

```

Now, we need to subtract out the row containing the new pivot from this row. Again, we can do this using hand-written loops, but is there some way to harness the STL to solve the problem? The answer is yes, thanks to a second version of `transform`. Up to this point, the `transform` we've used transforms a single range with a unary function. However, another form of `transform` exists that accepts as input *two* ranges and a *binary* function, then produces a stream of values by applying the binary function pairwise to the values in range. For example, given two iterator ranges delineating the ranges `{0, 1, 2, 3, 4}` and `{5, 6, 7, 8, 9}`, using this second version of `transform` and the `multiplies<int>` function object would produce the values `{0, 6, 14, 24, 36}`.

The prototype for the two-range version of `transform` is

```

template <typename InItr1,
          typename InItr2,
          typename OutItr,
          typename BinaryFunction>
OutItr transform(InItr1 start1, InItr1 end1,
                InItr2 start2,
                OutItr out,
                BinaryFunction fn);

```

Notice that the first two arguments define a full range whereas the third argument only specifies the *start* of the second range. This version of `transform` assumes that the second iterator range has at least as many elements as the first range, which is valid for our current application.

To subtract the two rows, we can use this version of `transform` and the `minus<double>` function object to subtract the values in the new pivot row from the values in the current row. This is shown here:


```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                      matrix.row_begin(row), // Subtract values from new pivot row
                      matrix.row_begin(nextRow), // Overwrite existing values
                      minus<double>());

            /* ... */
        }
    }
}

```

We're almost done cleaning up this row and there's only one more step – normalizing the row to restore any pivots. Because the algorithm moves from left to right, any pivots in the row we just modified must be to the left of the current column. We thus need to check to see if there are any nonzero values to the left of the current column that might be pivots. As you might have guessed, this can easily be done using the STL, thanks to the `<functional>` library. In particular, we can use the `not_equal_to<double>` function object in conjunction with `bind2nd` to construct a unary function that returns true for any nonzero value, then can pass this function into the STL `find_if` algorithm to obtain an iterator to the first nonzero element in the row. If we find a value, we'll normalize the row, and otherwise just leave the row as-is. This is shown here:

```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                      matrix.row_begin(row), // Subtract values from new pivot row
                      matrix.row_begin(nextRow), // Overwrite existing values
                      minus<double>());

            grid<double>::iterator nonzero =
                find_if(matrix.row_begin(nextRow), matrix.row_begin(nextRow) + col,
                       bind2nd(not_equal_to<double>(), 0.0));
            if(nonzero != matrix.row_begin(nextRow) + col)
                DivideRow(matrix, nextRow, *nonzero);
        }

        /* ... */
    }
}

```

Notice that the arguments to `find_if` were

```
matrix.row_begin(nextRow)
```

and

```
matrix.row_begin(nextRow) + col
```

These are iterators defining all elements in the row to the left of the current column. Since `find_if` returns the end of the iterator range as a sentinel if no elements in the range pass the predicate, we check if the returned iterator is equal to `matrix.row_begin(nextRow) + col` before normalizing the row again.

At this point we've cleaned up the row and after this inner `for` loop terminates we will have made the element at position `(row, col)` a pivot. The last step is to move this row as high up in the matrix as possible so that we don't end up considering the row again in the future. We thus want to swap this row with the first row that doesn't already contain a pivot. But we know which row that is – it's the row at the position specified by `pivotsPlaced`. Using the `swap_ranges` algorithm, which exchanges the contents of two iterator ranges, this can be done in one line. We'll also increment `pivotsPlaced`, since we've just made another pivot. This yields the final version of `GaussJordanEliminate`, as shown here:

```

void GaussJordanEliminate(grid<double>& matrix)
{
    const int maxPivots = min(matrix.numRows(), matrix.numCols() - 1);
    int pivotsPlaced = 0;
    for(int col = 0; col < matrix.numCols() - 1 && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                      matrix.row_begin(row), // Subtract values from new pivot row
                      matrix.row_begin(nextRow), // Overwrite existing values
                      minus<double>());

            grid<double>::iterator nonzero =
                find_if(matrix.row_begin(nextRow), matrix.row_end(nextRow) + col,
                       bind2nd(not_equal_to<double>(), 0.0));
            if(nonzero != matrix.row_end(nextRow) + col)
                DivideRow(matrix, nextRow, *nonzero);
        }
        swap_ranges(matrix.row_begin(row), matrix.row_end(row),
                   matrix.row_begin(pivotsPlaced));
        ++pivotsPlaced;
    }
}

```

This function is dense, but correctly implements Gauss-Jordan elimination. We can now solve linear systems of equations!

Computing Inverses

Before concluding our treatment of Gauss-Jordan elimination, we'll consider one practical application: inverting a matrix. For this discussion, we will only consider square matrices.

When multiplying real numbers, the number 1 is the *identity* in that the product of any number and 1 is that original number. That is, $1 \cdot x = x \cdot 1 = x$. Similarly, the *inverse* of a number (denoted x^{-1}) is the unique number such that the product of the original number and the inverse is the identity: $x \cdot x^{-1} = x^{-1} \cdot x = 1$. When multiplying matrices,* the *identity matrix* (denoted I) plays the same role; given a matrix A , the identity matrix is the matrix such that $AI = IA = A$. Similarly, The *inverse* of a matrix is a matrix that, when multiplied with the original matrix, yields the identity: $AA^{-1} = A^{-1}A = I$. Not all numbers have an inverse – in particular, there is no number which multiplied by zero yields 1. Similarly, not all matrices have an inverse. A matrix with no inverse is said to be *singular*.

For square matrices, matrix multiplication is only defined when the two matrices have the same size. Therefore, there are multiple identity matrices, one for each size of square matrix. The identity matrix is a matrix that is

* We did not cover matrix multiplication here, though it's well-defined for matrices if the dimensions are correct. Consult a linear algebra textbook for more information on matrix multiplication.

zero everywhere except for on the main diagonal (from the upper-left corner to the lower-right corner), where it is all ones. Thus the 3x3 identity matrix is

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

When working with real numbers it is easy to compute the inverse – just take the reciprocal of the number. Inverting a matrix is considerably trickier because matrix multiplication is more complicated. Fortunately, we can use a modified version of Gauss-Jordan elimination to compute the inverse of a matrix. The trick is to augment the initial matrix with the identity matrix, then reduce the matrix to rref. If the matrix is nonsingular, the result of the row reduction should be the identity matrix augmented with the inverse of the original matrix. For example, if we want to invert the following matrix:

$$\begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix}$$

We would first augment it with the identity matrix, as shown here:

$$\left(\begin{array}{ccc|ccc} 3 & 1 & 4 & 1 & 0 & 0 \\ 1 & 5 & 9 & 0 & 1 & 0 \\ 2 & 6 & 5 & 0 & 0 & 1 \end{array} \right)$$

Then convert the matrix to rref using the Gauss-Jordan Elimination algorithm to yield

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 29/90 & -19/90 & 11/90 \\ 0 & 1 & 0 & -13/90 & -7/90 & 23/90 \\ 0 & 0 & 1 & 2/45 & 8/45 & -7/45 \end{array} \right)$$

The matrix is now the identity matrix augmented with the matrix inverse, so the inverse matrix is

$$\begin{pmatrix} 29/90 & -19/90 & 11/90 \\ -13/90 & -7/90 & 23/90 \\ 2/45 & 8/45 & -7/45 \end{pmatrix}$$

Proving this technique correct is beyond the scope of a mere C++ programming text, so we will take its correctness for granted.

Using our current implementation of Gauss-Jordan Elimination, we cannot compute inverse matrices because we have hardcoded the assumption that the input matrix has exactly one augmented column. This is easily remedied with an additional parameter to the function specifying how many columns in the matrix are part of the augmented matrix, as shown here:

```

void GaussJordanEliminate(grid<double>& matrix, int numAugmentedCols)
{
    const int numMatrixColumns = matrix.numCols() - numAugmentedCols;
    const int maxPivots = min(matrix.numRows(), numMatrixColumns);

    int pivotsPlaced = 0;
    for(int col = 0; col < numMatrixColumns && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                      matrix.row_begin(row), // Subtract values from new pivot row
                      matrix.row_begin(nextRow), // Overwrite existing values
                      minus<double>());

            grid<double>::iterator nonzero =
                find_if(matrix.row_begin(nextRow), matrix.row_end(nextRow) + col,
                       bind2nd(not_equal_to<double>(), 0.0));
            if(nonzero != matrix.row_end(nextRow) + col)
                DivideRow(matrix, nextRow, *nonzero);
        }
        swap_ranges(matrix.row_begin(row), matrix.row_end(row),
                    matrix.row_begin(pivotsPlaced));
        ++pivotsPlaced;
    }
}

```

We can now begin writing a function `InvertMatrix` which accepts a `grid<double>` as a parameter and updates it to hold its inverse. Our first step is to ensure that the matrix is square, which we can do here:

```

void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    /* ... */
}

```

Next, we need to construct a matrix that has the input matrix as its body and that is augmented with the identity matrix. Unfortunately, we cannot simply use the `grid`'s `resize` member function since this discards all internal data. Instead, we'll construct a second `grid` of the proper size and explicitly initialize it to hold the right values. We'll use the `copy` algorithm to do the actual copying, plus a simple for loop to initialize the augmented part of the matrix to the identity. This is shown here:

```

void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
            scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    /* ... */
}

```

Now that we've built our matrix, we can hand it off to the `GaussJordanEliminate` function to get row-reduced, as shown here:

```

void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
            scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    GaussJordanEliminate(scratchWork, matrix.numCols());

    /* ... */
}

```

At this point, one of two cases holds. If the input matrix was invertible, the reduced row echelon form of the matrix should be the identity matrix and the augmented portion of the matrix should be the matrix inverse. Otherwise, the reduced row echelon form of the matrix will *not* be the identity and we should report an error. We can check if the matrix is the identity matrix by iterating over every element and checking if there are nonzero values off the main diagonal or values other than one on the diagonal. This is shown here:

```

void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
            scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    GaussJordanEliminate(scratchWork, matrix.numCols());

    for(int row = 0; row < matrix.numRows(); ++row)
        for(int col = 0; col < matrix.numCols(); ++col)
            if((row == col && scratchWork[row][col] != 1.0) ||
                (row != col && scratchWork[row][col] != 0.0))
                throw invalid_argument("Singular matrix.");

    /* ... */
}

```

If the matrix is invertible, then we need to copy the augmented portion of the matrix back into the parameter. We'll again use `copy` to help with this one. Since we want to copy the second half of each row back into the source matrix, we'll start iterating at `row_begin(...) + matrix.numCols()`, which is the first element of the augmented matrix in each row. The final code for `InvertMatrix` is shown here:

```

void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
            scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    GaussJordanEliminate(scratchWork, matrix.numCols());

    for(int row = 0; row < matrix.numRows(); ++row)
        for(int col = 0; col < matrix.numCols(); ++col)
            if((row == col && scratchWork[row][col] != 1.0) ||
                (row != col && scratchWork[row][col] != 0.0))
                throw invalid_argument("Singular matrix.");

    for(int row = 0; row < matrix.numRows(); ++row)
        copy(scratchWork.row_begin(row) + matrix.numCols(),
            scratchWork.row_end(row),
            matrix.row_begin(row));
}

```

More to Explore

Whew! We've just finished a whirlwind tour of linear algebra and matrix algorithms and got a chance to see exactly how much mileage we can get out of the STL. But we've barely scratched the surface of matrix processing and numerical computing and there are far more advanced techniques out there. If you're interested in exploring what else you can do with this code, consider looking into the following:

1. **Templates.** The code we've written has assumed that the `grid` contains `doubles`, but there's nothing fundamentally wrong with using a `grid<float>` or even a `grid<complex<double> >` (the `complex` template type, defined in `<complex>`, represents a complex number and is one of the oldest headers in the standard library). Templatize the code we've written so that it works on `grids` of any type. If you're up for a real challenge, implement a `RationalNumber` class with full operator overloading support and use it in conjunction with `GaussJordanEliminate` to get mathematically precise values for the inverse of a matrix.
2. **Gaussian Elimination with Backsolving.** If you are only interested in solving linear systems of equations and don't necessarily want to get the fully row-reduced form of a matrix, you can use an alternative technique called *Gaussian elimination with backsolving*. This approach is much faster than Gauss-Jordan elimination and is used more frequently in practice. It also is not particularly difficult to implement and with a bit of work can give you impressive performance gains.
3. **Numerical Stability.** Computations involving `doubles` or `floats` are inherently suspect because the binary representation of these types is subject to rounding errors and other inaccuracies. Research how to improve the numerical stability of this algorithm so that you can use it to obtain increasingly concise values.

Complete Source Listing

```

void DivideRow(grid<double>& matrix, int row, double by)
{
    transform(matrix.row_begin(row),
               matrix.row_end(row),
               matrix.row_begin(row),
               bind2nd(divides<double>(), by));
}

int FindPivotRow(grid<double>& matrix, int col, int numFound)
{
    for(int row = numFound; row < matrix.numRows(); ++row)
        if(matrix[row][col] != 0)
            return row;
    return -1;
}

void GaussJordanEliminate(grid<double>& matrix, int numAugmentedCols)
{
    const int numMatrixColumns = matrix.numCols() - numAugmentedCols;
    const int maxPivots = min(matrix.numRows(), numMatrixColumns);

    int pivotsPlaced = 0;
    for(int col = 0; col < numMatrixColumns && pivotsPlaced < maxPivots; ++col)
    {
        const int row = FindPivotRow(matrix, col, pivotsPlaced);
        if(row == -1) continue;

        DivideRow(matrix, row, matrix[row][col]);

        for(nextRow = 0; nextRow < matrix.numRows(); ++nextRow)
        {
            /* Skip the current row and rows that have zero in this column. */
            if(nextRow == row || matrix[nextRow][col] != 0) continue;

            DivideRow(matrix, nextRow, matrix[nextRow][col]);

            transform(matrix.row_begin(nextRow), matrix.row_end(nextRow),
                       matrix.row_begin(row), // Subtract values from new pivot row
                       matrix.row_begin(nextRow), // Overwrite existing values
                       minus<double>());

            grid<double>::iterator nonzero =
                find_if(matrix.row_begin(nextRow), matrix.row_end(nextRow) + col,
                        bind2nd(not_equal_to<double>(), 0.0));
            if(nonzero != matrix.row_end(nextRow) + col)
                DivideRow(matrix, nextRow, *nonzero);
        }
        swap_ranges(matrix.row_begin(row), matrix.row_end(row),
                    matrix.row_begin(pivotsPlaced));
        ++pivotsPlaced;
    }
}

```

```
void InvertMatrix(grid<double>& matrix)
{
    if(matrix.numRows() != matrix.numCols())
        throw invalid_argument("Cannot invert a non-square matrix.");

    grid<double> scratchWork(matrix.numRows(), matrix.numCols() * 2);

    /* Copy existing data. */
    for(int row = 0; row < matrix.numRows(); ++row)
        copy(matrix.row_begin(row), matrix.row_end(row),
            scratchWork.row_begin(row));

    /* Write the identity matrix. */
    for(int row = 0; row < matrix.numRows(); ++row)
        scratchWork[row][row + matrix.numCols()] = 1.0;

    GaussJordanEliminate(scratchWork, matrix.numCols());

    for(int row = 0; row < matrix.numRows(); ++row)
        for(int col = 0; col < matrix.numCols(); ++col)
            if((row == col && scratchWork[row][col] != 1.0) ||
                (row != col && scratchWork[row][col] != 0.0))
                throw invalid_argument("Singular matrix.");

    for(int row = 0; row < matrix.numRows(); ++row)
        copy(scratchWork.row_begin(row) + matrix.numCols(),
            scratchWork.row_end(row),
            matrix.row_begin(row));
}
```

Chapter 29: Introduction to Inheritance

It's impossible to learn C++ or any other object-oriented language without encountering *inheritance*, a mechanism that lets different classes share implementation and interface design. However, inheritance has evolved greatly since it was first introduced to C++, and consequently C++ supports several different inheritance schemes. This chapter introduces and motivates inheritance, then discusses how inheritance interacts with other language features.

Inheritance of Implementation and Interface

C++ started off as a language called “C with Classes,” so named because it was essentially the C programming language with support for classes and object-oriented programming. C++ is the more modern incarnation of C with Classes, so most (but not all) of the features of C with Classes also appear in C++.

The inheritance introduced in C with Classes allows you to define new classes in terms of older ones. For example, suppose you are using a third-party library that exports a `Printer` class, as shown below:

```
class Printer
{
public:
    /* Constructor, destructor, etc. */

    void setFont(const string& fontName, int size);
    void setColor(const string& color);
    void printDocument(const string& document);
private:
    /* Implementation details */
};
```

This `Printer` class exports several formatting functions, plus `printDocument`, which accepts a string of the document to print. Let's assume that `printDocument` is implemented synchronously – that is, `printDocument` will not return until the document has finished printing. In some cases this behavior is fine, but in others it's simply not acceptable. For example, suppose you're writing database software for a large library and want to give users the option to print out call numbers. Chances are that people using your software will print call numbers for multiple books and will be irritated if they have to sit and wait for their documents to finish printing before continuing their search. To address this problem, you decide to add a new feature to the printer that lets the users enqueue several documents and print them in a single batch job. That way, users searching for books can enqueue call numbers without long pauses, then print them all in one operation. However, you don't want to force users to queue up documents and then do a batch print job at the end – after all, maybe they're just looking for one book – so you want to retain all of the original features of the `Printer` class. How can you elegantly solve this problem in software?

Let's consider the above problem from a programming perspective. The important points are:

- We are provided the `Printer` class from an external source, so we cannot modify the `Printer` interface.
- We want to preserve all of the existing functionality from the `Printer` class.
- We want to extend the `Printer` class to include extra functionality.

This is an ideal spot to use *inheritance*, a means for defining a new class in terms of an older one. We have an existing class that contains most of our needed functionality, but we'd like to add some extra features.

Let's define a `BatchPrinter` class that supports two new functions, `enqueueDocument` and `printAllDocuments`, on top of all of the regular `Printer` functionality. In C++, we write this as

```
class BatchPrinter: public Printer // Inherit from Printer
{
public:
    void enqueueDocument(const string& document);
    void printAllDocuments();
private:
    queue<string> documents; // Document queue
};
```

Here, the class declaration `class BatchPrinter: public Printer` indicates that the new class `BatchPrinter` inherits the functionality of the `Printer` class. Although we haven't explicitly provided the `printDocument` or `setFont` functions, since those functions are defined in `Printer`, they are also part of `BatchPrinter`. For example:

```
BatchPrinter myPrinter;
myPrinter.setColor("Red"); // Inherited from Printer
myPrinter.printDocument("This is a document!"); // Same
myPrinter.enqueueDocument("Print this one later."); // Defined in BatchPrinter
myPrinter.printAllDocuments(); // Same
```

While the `BatchPrinter` can do everything that a `Printer` can do, the converse is not true – a `Printer` cannot call `enqueueDocument` or `printAllDocuments`, since we did not modify the `Printer` class interface.

In the above setup, `Printer` is called a *base class* of `BatchPrinter`, which is a *derived class*. In C++ jargon, the relationship between a derived class and its base class is the *is-a* relationship. That is, a `BatchPrinter` *is-a* `Printer` because everything the `Printer` can do, the `BatchPrinter` can do as well. The converse is not true, though, since a `Printer` is not necessarily a `BatchPrinter`.

Because `BatchPrinter` *is-a* `Printer`, anywhere that a `Printer` is expected we can instead provide a `BatchPrinter`. For example, suppose we have a function that accepts a `Printer` object, perhaps to configure its font rendering, as shown here:

```
void InitializePrinter(Printer& p);
```

Then the following code is perfectly legal:

```
BatchPrinter batch;
InitializePrinter(batch);
```

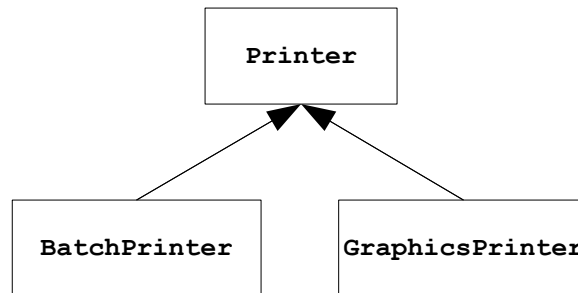
Although `InitializePrinter` expects an argument of type `Printer&`, we can instead provide it a `BatchPrinter`. This operation is well-defined and perfectly safe because the `BatchPrinter` contains all of the functionality of a regular `Printer`. If we temporarily forget about all of the extra functionality provided by the `BatchPrinter` class, we still have a good old-fashioned `Printer`. When working with inheritance, you can think of the types of arguments to functions as specifying the minimum requirements for the parameter. A function accepting a `Printer&` or a `const Printer&` can take in an object of any type, provided of course that it ultimately derives from `Printer`.

Note that it is completely legal to have several classes inherit from a single base class. Thus, if we wanted to develop another printer that supported graphics printing in addition to text, we could write the following class definition:

```
class GraphicsPrinter: public Printer
{
public:
    /* Constructor, destructor, etc. */
    void printPicture(const Picture& picture); // For some Picture class
private:
    /* Implementation details */
};
```

Now, `GraphicsPrinter` can do everything a regular `Printer` can do, but can also print `Picture` objects. Again, `GraphicsPrinter` *is-a* `Printer`, but not vice-versa. Similarly, `GraphicsPrinter` is not a `BatchPrinter`. Although they are both derived classes of `Printer`, they have nothing else in common.

It sometimes help to visualize the inheritance relations between classes as a tree. We adopt the convention that if one class derives from another, the first class is represented as a child of the second. We also label all edges in the tree with arrows pointing from derived classes to base classes. For example, `Printer`, `BatchPrinter`, and `GraphicsPrinter` are all related as follows:



Runtime Costs of Basic Inheritance

The inheritance scheme outlined above incurs no runtime penalties. Programs using this type of inheritance will be just as fast as programs not using inheritance.

In memory, a derived class is simply a base class object with its extra data members tacked on the end. For example, suppose you have the following classes:

```
class BaseClass
{
private:
    int baseX, baseY;
};

class DerivedClass: public BaseClass
{
private:
    int derX, derY;
};
```

Then, in memory, a `DerivedClass` object looks like this:

Address 1000	baseX	<- BaseClass members
1004	baseY	
1008	derX	<- DerivedClass -specific members
1012	derY	

Notice that the first eight bytes of this object are precisely the data members of a `BaseClass` object. This is in part the reason that you can treat instances of a derived class as instances of a base class. This in-memory representation of inheritance is extremely efficient and is one of the reasons that C++ was so popular in its infancy. Most competing object-oriented languages represented objects with considerably more complicated structures that required complex pointer lookups, so inheritance in those languages incurred a steep runtime penalty. C++, on the other hand, supported this simple form of inheritance with zero overhead.

Inheritance of Interface

The inheritance pattern outlined above uses inheritance to add *extensions* to existing classes. This is undoubtedly useful, but does not arise frequently in practice. A different version of inheritance, called *inheritance of interface*, is extraordinarily useful in modern programming.

Let's return to the `Printer` class. `Printer` exports a `printDocument` member function that accepts a string parameter, then sends the string to the printer. One of our other derived classes, `GraphicsPrinter`, has a `printPicture` member function that accepts some sort of `Picture` object, then sends the picture to the printer. What if we want to print a document containing a mix of text and pictures – for example, this course reader? We'd then need to introduce yet another subclass of `Printer`, perhaps a `MixedTextPrinter`, that supports a `printMixedText` member function that prints a combination of text and images. While we could continue to use the style of inheritance introduced above, it will quickly spiral out of control for several reasons. First, each printer can only print out one type of document. That is, a `MixedTextPrinter` cannot print out pictures, nor a `GraphicsPrinter` a mixed-text document. We could eliminate this problem by writing a single `MixedTextAndGraphicsPrinter` class, but this too has its problems if we then introduce another type of object to print (say, a high-resolution photo) that required its own special printing code. This leads to the second problem, a lack of extensibility. For any new type of object we want to print, we need to introduce another member function or class capable of handling that object. In our case this is inconvenient and does not scale well. We need to pick another plan of attack.

The problem is that everything that might get sent to the printer requires slightly different logic to print out. Text documents need to apply fonts and formatting transformations, graphics documents need to convert from some application-specific format into something readable by the printer, and mixed-text documents need to arrange their text and images into an appropriate layout before processing each piece individually. But each type of document has one piece of functionality in common. Most printers are programmed to accept as input a grid of dots representing the document to print. That is, whether you're printing text or a three-dimensional pie chart, the input to the printer is a grid of pixels representing the dots making up the image. A text document might end up producing different pixels than a high-resolution photo, but they both end up as pixels at some point. Provided that we can transform an object in memory into a mess of pixels, we can send it to the printer.

Consider the following class definition for a `GenericBatchPrinter` object:

```

class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument( /* What goes here? */ );
    void printAllDocuments();
private:
    /* Implementation details */
};

```

This `GenericBatchPrinter` object exports an `enqueueDocument` function that stores an arbitrary document in a print queue that can then be printed by calling `printAllDocuments`. There is one major question, though: what should the parameter type be? We can print any type of document we can think of, provided that we can convert it into a grid of pixels. We might be tempted to accept a grid of pixels as a parameter. This, however, has several drawbacks. First, pixel grids take up a huge amount of memory. Color printers usually store color information as quadruples of the cyan, magenta, yellow, and black (CYMK) color components, so a single pixel is usually a four-byte value. If you have a 200 DPI printer and want to print to an 8.5 x 11" page, you'd need to store around 75kb. That's a lot of memory, and if you wanted to enqueue a large number of documents this approach might strain or exhaust system resources. Plus, we don't actually need the pixels until we begin printing, and even then we only need pixel information for one document at a time. Second, what if later in design we realize that we need extra information about the print job? For example, suppose we want to implement a printing priority system where more urgent documents print before less important ones. In this case, we'd need to add an extra parameter to `enqueueDocument` representing that priority and all existing code using `enqueueDocument` would stop working. Finally, this approach exposes too much of the inner workings of `GenericBatchPrinter` to the client. By treating documents as masses of pixels instead of documents, the `GenericBatchPrinter` violates some of the fundamental rules of data abstraction.

Let's review these problems:

- The above approach is needlessly memory-intensive by catering to the lowest common denominator of all possible printable documents.
- The approach limits later extensions by fixing the parameter as an inflexible pixel array.
- The `GenericBatchPrinter` should work on documents, not pixels.

Is there a language feature that would let us solve all of these problems? The answer is yes. What if we simply create an object that looks like this:

```

class Document
{
public:
    /* Constructor, destructor, etc. */

    grid<pixelT> convertToPixelArray() const; // For some struct pixelT
    int getPriority() const;
private:
    /* Implementation details */
};

```

This `Document` class exports two functions – `convertToPixelArray()`, which converts the document from its current format into a `grid<pixelT>` of the pixels in the image, and `getPriority()`, which returns the relative priority of the document.

We can now have the `enqueueDocument` function from `GenericBatchPrinter` accept a `Document` as a parameter. That way, when the `GenericBatchPrinter` needs to get an array of pixels, it can simply call `convertToPixelArray` on any stored `Document`. Similarly, if the `GenericBatchPrinter` decides to implement a priority system, it can use the information provided by the `Document`'s `getPriority()` function. Moreover, if later on during implementation we realize that `GenericBatchPrinter` needs access to additional information, we can simply add extra member functions to the `Document` class. While this still requires us to rewrite code to add these member functions, the actual calls to `enqueueDocument` will still work correctly, and the only people who need to modify any code is the `Document` class implementer, not the `Document` class clients.

While this solution might seem elegant, it still has a major problem – how can we write a `Document` class that encompasses all of the possible documents we can try to print? The answer is simple: we can't. Using the language features we've covered so far, it simply isn't possible to solve this problem.

Consider for a minute what form our problem looks like. We need to provide a `Document` object that represents a printable document, but we cannot write a single umbrella class representing every conceivable document. Instead of creating a single `Document` class, what if we could create *several different* `Document` classes, each of which provided a working implementation of the `convertToPixelArray` and `getDocumentName` functions? That is, we might have a `TextDocument` class that stores a string and whose `convertToPixelArray` converts the string into a grid of pixels representing that string. We could also have a `GraphicsDocument` object with member functions like `addCircle` or `addImage` whose `convertToPixelArray` function generates a graphical representation of the stored image. In other words, we want to make the `Document` class represent an *interface* rather than an *implementation*. `Document` should simply outline what member functions are common to other classes like `GraphicsDocument` or `TextDocument` without specifying how those functions should work. In C++ code, this means that we will rewrite the `Document` class to look like this:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual int* convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

If you'll notice, we tagged both of the member functions with the `virtual` keyword, and put an `= 0` after each function declaration. What does this strange syntax mean? The `= 0` syntax is an odd bit of C++ syntax that says “this function does not actually exist.” In other words, we've prototyped a function that we have no intention of ever writing. Why would we ever want to do this? The reason is simple. Because we've prototyped the function, other pieces of C++ code know what the parameter and return types are for the `convertToPixelArray` and `getDocumentName` functions. However, since there is no meaningful implementation for either of these functions, we add the `= 0` to tell C++ not to expect one.

To understand the `virtual` keyword, consider this `TextDocument` class outlined below:


```

class TextDocument: public Document // Inherit from Document
{
public:
    /* Constructor, destructor, etc. */
    virtual grid<pixelT> convertToPixelArray() const; // Has an implementation
    virtual int getPriority() const; // Has an actual implementation

    void setText(const string& text); // Text-specific formatting functions
    void setFont(const string& font);
    void setSize(int size);
private:
    /* Implementation details */
};

```

This `TextDocument` class inherits from `Document`, and although `Document` has a declaration of the `convertToPixelArray` and `getPriority` functions, `TextDocument` has specified that it too contains these functions. They are marked `virtual`, as in the `Document` class, but unlike `Document`'s versions of these functions the `TextDocument` functions do not have the `= 0` notation after them. This indicates that the functions actually exist and do have implementations. We won't cover how these functions are implemented since it's irrelevant to our discussion, but because they have actual implementations code like this is perfectly legal:

```

TextDocument myDocument;
grid<pixelT> array = myDocument.convertToPixelArray();

```

We've covered the `= 0` notation, but what does the `virtual` keyword mean? To understand how `virtual` works, consider the following code snippet:

```

TextDocument* myDocument = new TextDocument;
grid<pixelT> array = myDocument->convertToPixelArray();

```

This code should not be at all surprising – we've just rewritten the above code using a pointer to a `TextDocument` rather than a stack-based `TextDocument`. However, consider this code snippet below:

```

Document* myDocument = new TextDocument; // Note: pointer is a Document *
grid<pixelT> array = myDocument->convertToPixelArray();

```

This code looks similar to the above code but represents a fundamentally different operation. In the first line, we allocate a new `TextDocument` object, but store it in a pointer of type `Document *`. Initially, this might seem nonsensical – pointers of type `Document *` should only be able to point to objects of type `Document`. However, because `TextDocument` is a derived class of `Document`, *TextDocument is-a Document*. The *is-a* relation applies literally here – since `TextDocument is-a Document`, we can point to objects of type `TextDocument` using pointers of type `Document *`.

Even if we can point to objects of type `TextDocument` with objects of type `Document *`, why is the line `myDocument->convertToPixelArray()` legal? As mentioned earlier, the `Document` class definition says that `Document` does not have an implementation of `convertToPixelArray`, so it seems like this code should not compile. This is where the `virtual` keyword comes in. Since we marked `convertToPixelArray` `virtual`, when we call the `convertToPixelArray` function through a `Document *` object, C++ will call the function named `convertToPixelArray` for the class that's *actually being pointed at*, not the `convertToPixelArray` function defined for objects of the type of the pointer. In this case, since our `Document *` is actually pointing at a `TextDocument`, the call to `convertToPixelArray` will call the `TextDocument`'s version of `convertToPixelArray`.

The above approach to the problem is known as *polymorphism*. We define a base class (in this case `Document`) that exports several functions marked `virtual`. In our program, we pass around pointers to objects of this base class, which may in fact be pointing to a base class object or to some derived class. Whenever we make member function calls to the virtual functions of the base class, C++ figures out at runtime what type of object is being pointed at and calls its implementation of the virtual function.

Let's return to our `GenericBatchPrinter` class, which now in its final form looks something like this:

```
class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument(Document* doc);
    void printAllDocuments();
private:
    queue<Document *> documents;
};
```

Our `enqueueDocument` function now accepts a `Document *`, and its private data members include an STL queue of `Document *`s. We can now implement `enqueueDocument` and `printAllDocuments` using code like this:

```
void GenericBatchPrinter::enqueueDocument(Document* doc)
{
    documents.push(doc); // Recall STL queue uses push instead of enqueue
}

void GenericBatchPrinter::printAllDocuments()
{
    /* Print all enqueued documents */
    while(!documents.empty())
    {
        Document* nextDocument = documents.front();
        documents.pop(); // Recall STL queue requires explicit pop operation

        sendToPrinter(nextDocument->convertToPixelArray());
        delete nextDocument; // Assume it was allocated with new
    }
}
```

The `enqueueDocument` function accepts a `Document *` and enqueues it in the document queue, and the `printAllDocuments` function continuously dequeues documents, converts them to pixel arrays, then sends them to the printer. But while this above code might seem simple, it's actually working some wonders behind the scenes. Notice that when we call `nextDocument->convertToPixelArray()`, the object pointed at by `nextDocument` could be of *any type* derived from `Document`. That is, the above code will work whether we've enqueued `TextDocuments`, `GraphicsDocuments`, or even `MixedTextDocuments`. Moreover, the `GenericBatchPrinter` class does not even need to know of the existence of these types of documents; as long as `GenericBatchPrinter` knows the generic `Document` interface, C++ can determine which functions to call. This is the main strength of inheritance – we can write code that works with objects of arbitrary types by identifying the common functionality across those types and writing code solely in terms of these operations.

Virtual Functions, Pure Virtual Functions, and Abstract Classes

In the above example with the `Document` class, we defined `Document` as

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual grid<pixelT> convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

Here, all of the `Document` member functions are marked `virtual` and have the `= 0` syntax to indicate that the functions are not actually defined. Functions marked `virtual` with `= 0` are called *pure virtual functions* and represent functions that exist solely to define how other pieces of C++ code should interact with derived classes.*

Classes that contain pure virtual functions are called *abstract classes*. Because abstract classes contain code for which there is no implementation, it is illegal to directly instantiate abstract classes. In the case of our document example, this means that both of the following are illegal:

```
Document myDocument; // Error!
Document* myDocument = new Document; // Error!
```

Of course, it's still legal to declare `Document *` variables, since those are pointers to abstract classes rather than abstract classes themselves.

A derived class whose base class is abstract may or may not implement all of the pure virtual functions defined in the base class. If the derived class does implement each function, then the derived class is non-abstract (unless, of course, it introduces its own pure virtual functions). Otherwise, if there is at least one pure virtual function declared in the base class and not defined in the derived class, the derived class itself will be an abstract class.

There is no requirement that functions marked `virtual` be pure virtual functions. That is, you can provide virtual functions that have implementations. For example, consider the following class representing a rollerblader:

```
class RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    virtual void slowDown(); // Virtual, not pure virtual
private:
    /* Implementation details */
};
```

* Those of you familiar with inheritance in other languages like Java might wonder why C++ uses the awkward `= 0` syntax instead of a clearer keyword like `abstract` or `pure`. The reason was mostly political. Bjarne Stroustrup introduced pure virtual functions to the C++ language several weeks before the planned release of the next set of revisions to C++. Adding a new keyword would have delayed the next language release, so to ensure that C++ had support for pure virtual functions, he chose the `= 0` syntax.

```
void RollerBlader::slowDown() // Implementation doesn't have virtual keyword
{
    applyBrakes();
}
```

Here, `slowDown` is implemented as a virtual function that is not pure virtual. In the implementation of `slowDown`, you do not repeat the `virtual` keyword, and for all intents and purposes treat `slowDown` as a regular C++ function. Now, suppose we write an `InexperiencedRollerBlader` class, as shown here:

```
class InexperiencedRollerBlader: public RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    virtual void slowDown();
private:
    /* Implementation details */
};

void InexperiencedRollerBlader::slowDown()
{
    fallDown();
}
```

This `InexperiencedRollerBlader` class provides its own implementation of `slowDown` that calls some `fallDown` function.* Now, consider the following code snippet:

```
RollerBlader* blader = new RollerBlader;
blader->slowDown();

RollerBlader* blader2 = new InexperiencedRollerBlader;
blader2->slowDown();
```

In both cases, we call the `slowDown` function through a pointer of type `RollerBlader *`, so C++ will call the version of `slowDown` for the class that's actually pointed at. In the first case, this will call the `RollerBlader`'s version of `slowDown`, which calls `applyBrakes`. In the second, since `blader2` points to an `InexperiencedRollerBlader`, the `slowDown` call will call `InexperiencedRollerBlader`'s `slowDown` function, which then calls `fallDown`. In general, when calling a virtual function, C++ will invoke the version of the function that corresponds to the most derived implementation available in the object being pointed at. Because the `InexperiencedRollerBlader` implementation of `slowDown` replaces the base class version, `InexperiencedRollerBlader`'s implementation `slowDown` is said to *override* `RollerBlader`'s.

When inheriting from non-abstract classes that contain virtual functions, there is no requirement to provide your own implementation of the virtual functions. For example, a `StuntRollerBlader` might be able to do tricks a regular `RollerBlader` can't, but still slows down the same way. In code we could write this as

```
class StuntRollerBlader: public RollerBlader
{
public:
    /* Note: no mention of slowDown */
    void backflip();
    void tripleAxel();
};
```

* Of course, this is not based on personal experience. ☺

If we then were to write code that used `StuntRollerBlader`, calling `slowDown` would invoke `RollerBlader`'s version of `slowDown` since it is the most derived implementation of `slowDown` available to `StuntRollerBlader`. For example:

```
RollerBlader* blader = new StuntRollerBlader;
blader->slowDown(); // Calls RollerBlader::slowDown
```

Similarly, if we were to create a class `TerriblyInexperiencedRollerBlader` that exports a `panic` function but no `slowDown` function, as shown here:

```
class TerriblyInexperiencedRollerBlader: public InexperiencedRollerBlader
{
public:
    /* Note: no reference to slowDown */
    void panic();
};
```

Then the following code will invoke `InexperiencedRollerBlader::slowDown`, causing the roller blader to fall down:

```
RollerBlader* blader = new TerriblyInexperiencedRollerBlader;
blader->slowDown();
```

In this last example we wrote a class that derived from a class which itself was a derived class. This is perfectly legal and arises commonly in programming practice.

A Word of Warning

Consider the following two classes:

```
class NotVirtual
{
public:
    void notAVirtualFunction();
};

class NotVirtualDerived: public NotVirtual
{
public:
    void notAVirtualFunction();
};
```

Here, the base class `NotVirtual` exports a function called `notAVirtualFunction` and its derived class, `NotVirtualDerived`, also provides a `notAVirtualFunction` function. Although these functions have the same name, since `notAVirtualFunction` is not marked `virtual`, the derived class version does *not* replace the base class version. Consider this code snippet:

```
NotVirtual* nv = new NotVirtualDerived;
nv->notAVirtualFunction();
```

Here, since `NotVirtualDerived` is-a `NotVirtual`, the above code will compile. However, since `notAVirtualFunction` is (as its name suggests) not a virtual function, the above code will call the `NotVirtual` version of `notAVirtualFunction`, not `NotVirtualDerived`'s `notAVirtualFunction`.

If you want to let derived classes override functions in a base class, you *must* mark the base class's function `virtual`. Otherwise, C++ won't treat the function call virtually and will always call the version of the function associated with the type of the pointer. For example:

```
NotVirtual* nv = new NotVirtualDerived;
nv->notAVirtualFunction(); // Calls NotVirtual::notAVirtualFunction()

NotVirtualDerived *nv2 = new NotVirtualDerived;
nv2->notAVirtualFunction(); // Calls NotVirtualDerived::notAVirtualFunction();
```

In general, it is considered bad programming practice to have a derived class implement a member function with the same name as a non-virtual function in its base class. Doing so leads to the sorts of odd behavior shown above and is an easy source of errors.

The protected Access Specifier

Let's return to the `Document` class from earlier in the chapter. Suppose that while designing some of the `Document` subclasses, we note that every single subclass ends up having a `width` and `height` field. To minimize code duplication, we decide to move the `width` and `height` fields from the derived classes into the `Document` base class. Since we don't want `Document` class clients directly accessing these fields, we decide to mark them `private`, as shown here:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual grid<pixelT> convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    int width, height; // Warning: slight problem here
};
```

However, by moving `width` and `height` into the `Document` base class, we've accidentally introduced a problem into our code. Since `width` and `height` are `private`, even though `TextDocument` and the other subclasses inherit from `Document`, the subclasses will not be able to access the `width` and `height` fields. We want the `width` and `height` fields to be accessible only to the derived classes, but not to the outside world. Using only the C++ we've covered up to this point, this is impossible. However, there is a third access specifier beyond `public` and `private` called *protected* that does exactly what we want. Data members and functions marked `protected`, like `private` data members, cannot be accessed by class clients. However, unlike `private` variables, `protected` functions and data members *are* accessible by derived classes.

`protected` is a useful access specifier that in certain circumstances can make your code quite elegant. However, you should be very careful when granting derived classes `protected` access to data members. Like `public` data members, using `protected` data members locks your classes into a single implementation and can make code changes down the line difficult to impossible. Make sure that marking a data member `protected` is truly the right choice before proceeding. However, marking *member functions* `protected` is a common programming technique that lets you export member functions only usable by derived classes. We will see an example of `protected` member functions later in this chapter.

Virtual Destructors

An important topic we have ignored so far is *virtual destructors*. Consider the following two classes:

```
class BaseClass
{
public:
    BaseClass();
    ~BaseClass();
};

class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
    char* myString;
};

DerivedClass::DerivedClass()
{
    myString = new char[128]; // Allocate some memory
}

DerivedClass::~~DerivedClass()
{
    delete [] myString; // Deallocate the memory
}
```

Here, we have a trivial constructor and destructor for `BaseClass`. `DerivedClass`, on the other hand, has a constructor and destructor that allocate and deallocate a block of memory. What happens if we write the following code?

```
BaseClass* myClass = new DerivedClass;
delete myClass;
```

Intuitively, you'd think that since `myClass` points to a `DerivedClass` object, the `DerivedClass` destructor would invoke and clean up the dynamically-allocated memory. Unfortunately, this is not the case. The `myClass` pointer is statically-typed as a `BaseClass *` but points to an object of type `DerivedClass`, so `delete myClass` results in *undefined behavior*. The reason for this is that we didn't let C++ know that it should check to see if the object pointed at by a `BaseClass *` is really a `DerivedClass` when calling the destructor. Undefined behavior is never a good thing, so to fix this we mark the `BaseClass` destructor `virtual`. Unlike the other virtual functions we've encountered, though, derived class destructors do not replace the base class destructors. Instead, when invoking a destructor virtually, C++ will first call the derived class destructor, then the base class destructor. We thus change the two class declarations to look like this:

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass();
};
```

```

class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
    char *myString;
};

```

There is one more point to address here, the *pure virtual destructor*. Because virtual destructors do not act like regular virtual functions, even if you mark a destructor pure virtual, you must still provide an implementation. Thus, if we rewrote `BaseClass` to look like

```

class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass() = 0;
};

```

We'd then need to write a trivial implementation for the `BaseClass` destructor, as shown here:

```

BaseClass::~~BaseClass()
{
    // Do nothing
}

```

This is an unfortunate language quirk, but you should be aware of it since this will almost certainly come up in the future.

Runtime Costs of Virtual Functions

Virtual functions are incredibly useful and syntactically concise, but exactly how efficient are they? After all, a virtual function call invokes one of many possible functions, and somehow the compiler has to determine which version of the function to call. There could be an arbitrarily large number of derived classes overriding the particular virtual function, so a naïve `switch` statement that checks the type of the object would be prohibitively expensive. Fortunately, most C++ compilers use a particularly clever implementation of virtual functions that, while slower than regular function calls, are much faster than what you may have initially expected. Consider the following classes:

```

class BaseClass
{
public:
    virtual ~BaseClass() {} // Polymorphic classes need virtual destructors
    virtual void doSomething();
private:
    int baseX, baseY;
};

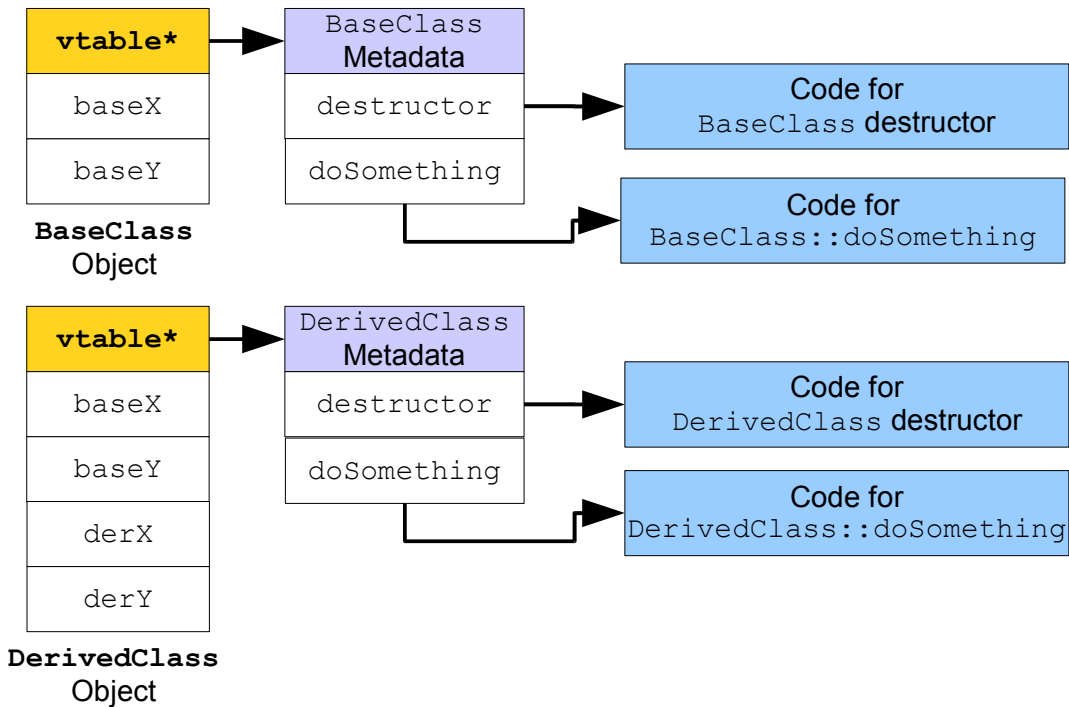
class DerivedClass: public BaseClass
{
public:
    virtual void doSomething(); // Override BaseClass version
private:
    int derX, derY;
};

```


Then the in-memory representation of a `DerivedClass` object looks like this:

Address 1000	vtable*	
1004	baseX	<- BaseClass members
1008	baseY	
1012	derX	<- DerivedClass -specific members
1016	derY	

As before, we see the `BaseClass` members followed by the `DerivedClass` members, but there is now another piece of data in this object: the `vtable*`, or *vtable-pointer*. This vtable-pointer is a *pointer to the virtual function table*. Whenever you create a class containing one or more virtual functions, C++ will create a table containing information about that class that includes metadata about the class along with a list of function pointers for each of the virtual functions in that class. For example, here's a diagram of a `BaseClass` object, a `DerivedClass` object, and their respective virtual function tables:



The virtual function table for `BaseClass` begins with metadata about the `BaseClass` type, then has two function pointers – one for the `BaseClass` destructor and one for `BaseClass`'s implementation of `doSomething`. The `DerivedClass` virtual function table similarly contains information about `DerivedClass`, as well as function pointers for the destructor and `doSomething` member functions. If you'll notice, the virtual function tables for `BaseClass` and `DerivedClass` have the member functions listed in the same order, with the destructor first and then `doSomething`. This allows C++ to invoke virtual functions quickly and efficiently. Suppose that we have the following code:

```
BaseClass* myPtr = RandomChance(0.5) ? new BaseClass : new DerivedClass;
myPtr->doSomething();
delete myPtr;
```

We assign a random object to `myPtr` that is either a `BaseClass` or a `DerivedClass` using the `RandomChance` function we wrote in the chapter on Snake. We then invoke the `doSomething` member function on the object and then `delete` it. To implement this functionality, the C++ compiler compiles the second two lines into machine code that performs the following operations:

```
// myPtr->doSomething();
1. Look at the first four bytes of the object pointed at by myPtr; this is the vtable* for the object.
2. Follow the vtable* and retrieve the second function pointer from the table; this corresponds to
   doSomething.
3. Call this function.

// delete myPtr;
1. Look at the first four bytes of the object pointed at by myPtr.
2. Follow the vtable* and retrieve the first function pointer from the table; this corresponds to the de-
   structor.
3. Call this function.
4. Deallocate the memory pointed at by myPtr.
```

This sequence of commands can be executed quickly and is efficient no matter how many subclasses of `BaseClass` exist. If there are millions of derived classes, this code still only has to make a single lookup through the virtual function table to call the proper function.

Although this above implementation of virtual function calls is considerably more efficient than a naïve approach, it is still noticeably slower than a regular function call because of the necessary virtual function table lookups. This extra overhead is the reason that C++ requires you to explicitly mark member functions you want to treat polymorphically `virtual` – if all functions were called this way, you would pay a performance hit irrespective of whether you actually used inheritance, a violation of the zero-overhead principle.

Invoking Virtual Member Functions Non-Virtually

From time to time, you will need to be able to explicitly invoke a base class's version of a virtual function. For example, suppose that you're designing a `HybridCar` that's a specialization of `Car`, both of which are defined below:

```
class Car
{
public:
    virtual ~Car(); // Polymorphic classes need virtual destructors
    virtual void applyBrakes();
    virtual void accelerate();
};

class HybridCar: public Car
{
public:
    virtual void applyBrakes();
    virtual void accelerate();
private:
    void chargeBattery();
    void dischargeBattery();
};
```

The `HybridCar` is exactly the same as a regular car, except that whenever a `HybridCar` slows down or speeds up, the `HybridCar` charges and discharges its electric motor to conserve fuel. We want to implement the ap-

plyBrakes and accelerate functions inside HybridCar such that they perform exactly the same tasks as the Car's version of these functions, but in addition perform the extra motor management.

Initially, we might consider implementing these functions like this:

```
void HybridCar::applyBrakes()
{
    applyBrakes(); // Uh oh...
    chargeBattery();
}

void HybridCar::accelerate()
{
    accelerate(); // Uh oh...
    dischargeBattery();
}
```

The above code is well-intentioned but incorrect. At a high level, we *want* to have the hybrid car accelerate or apply its brakes by doing whatever a regular car does, then managing the motor. As written, though, these functions will cause a stack overflow, since the calls to `applyBrakes()` and `accelerate()` recursively invoke the HybridCar's versions of these functions over and over. Since this doesn't work, what other approaches might we try? First, we could simply copy and paste the code from the Car class into the HybridCar class. This should cause you to cringe – a good solution to a problem should *never* involve copying and pasting code! More concretely, though, this approach has several problems. First, if we change the implementation of `accelerate()` or `applyBrakes()` in the Car class, we have to remember to make the same changes inside HybridCar. If we forget to do so, the code will compile but will be incorrect. Moreover, if the implementation of `accelerate()` or `applyBrakes()` in the Car class reference private data members or member functions of Car, the resulting code will be illegal. This clearly isn't the right way to solve this problem. What other options are available?

A second idea is to factor out the code for `applyBrakes` and `accelerate` into protected, non-virtual functions of the Car class. For example:

```
class Car
{
public:
    virtual ~Car();
    virtual void applyBrakes() { doApplyBrakes(); }
    virtual void accelerate() { doAccelerate(); }
protected:
    void doApplyBrakes(); // Non-virtual function that actually slows down.
    void doAccelerate(); // Non-virtual function that actually accelerates.
};

class HybridCar: public Car
{
public:
    virtual void applyBrakes() { doApplyBrakes(); chargeBattery(); }
    virtual void accelerate() { doAccelerate(); dischargeBattery(); }
private:
    void chargeBattery();
    void dischargeBattery();
};
```

Here, the virtual functions `applyBrakes` and `accelerate` are wrapped calls to non-virtual, protected functions written in the base class. To implement the derived versions of `applyBrakes` and `accelerate`, we can simply call these functions.

This approach is stylistically pleasing. The code that's common to `applyBrakes` and `accelerate` is factored out into helper member functions, so changes to one function appear in the other. But there's one minor problem with this approach: this solution only works if we can modify the `Car` class. In small projects this shouldn't be a problem, but if these classes are pieces in a much larger system the code may be off-limits – maybe it's being developed by another team, or perhaps it's been compiled into a program that expects the class definition to precisely match a specific pattern. This idea is clearly on the right track, but in some cases cannot work.

The optimal solution to this conundrum, however, is to simply have the `HybridCar`'s implementations of these functions directly call the versions of these functions defined in `Car`. When calling a virtual function through a pointer or reference, C++ ensures that the function call will “fall down” to the most derived class's implementation of that function. However, we can force C++ to call a specific version of a virtual function by calling it using the function's fully-qualified name. For example, consider this version of the `HybridCar`'s version of `applyBrakes`:

```
void HybridCar::applyBrakes()
{
    Car::applyBrakes(); // Call Car's version of applyBrakes, no polymorphism
    chargeBattery();
}
```

The syntax `Car::applyBrakes` instructs C++ to call the `Car` class's version of `applyBrakes`. Even though `applyBrakes` is virtual, since we've used the fully-qualified name, C++ will not resolve the call at runtime and we are guaranteed to invoke `Car`'s version of the function. We can write an `accelerate` function for `HybridCar` similarly.

When using the fully-qualified-name syntax, you're allowed to access *any* superclass's version of the function, not just the direct ancestor. For example, if `Car` were derived from the even more generic class `FourWheeledVehicle` that itself provides an `applyBrakes` method, we could invoke that version from `HybridCar` by writing `FourWheeledVehicle::applyBrakes()`. You can also use the fully-qualified name syntax as a class client, though it is rare to see this in practice.

Object Initialization in Derived Classes

Recall from several chapters ago that class construction proceeds in three steps – allocating space to hold the object, calling the constructors of all data members, and invoking the object constructor. While this picture is mostly correct, it omits an important step – initialization of base classes. Let's suppose we have the following classes:

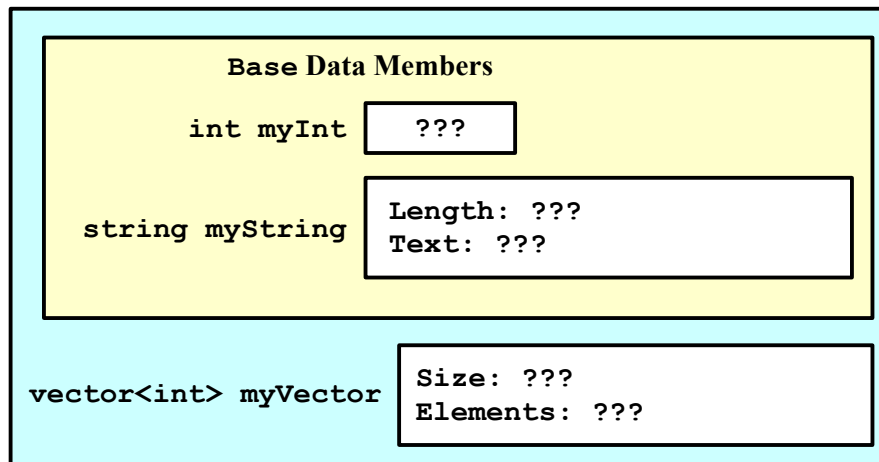
```

class Base
{
public:
    Base() : myInt(137), myString("Base string") {}
    virtual ~Base();
private:
    int myInt;
    string myString;
};

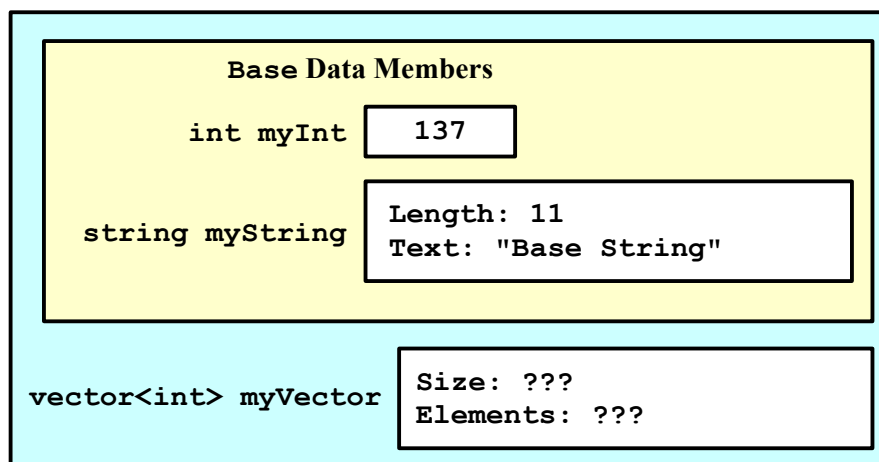
class Derived: public Base
{
private:
    vector<int> myVector;
};

```

Because we have not defined a constructor for `Derived`, C++ will automatically supply it with a default, zero-argument constructor that invokes the default constructor of the `Base` object. To see what this means, let's trace through the construction of a new `Derived` object. First, C++ gives the object a block of uninitialized memory with enough space to hold all of the parts of the `Derived`. This memory looks something like this:



At this point, C++ will initialize the `Base` class using its default constructor. Similarly, if `Base` has any parent classes, those parent classes would also be initialized. After this step, the object now looks like this:



From this point forward, construction will proceed as normal.

By default, derived class constructors invoke the default constructor for their base classes, or a zero-argument constructor if one has explicitly been defined. This is often, but not always, the desired behavior for a class. But what if you want to invoke a different constructor? For example, let's return to the `Car` example from earlier in this chapter. Suppose that `Car` exports a single constructor that accepts a `string` encoding the license number. For example:

```
class Car
{
public:
    explicit Car(const string& licenseNum) : license(licenseNum) {}
    virtual ~Car() {}

    virtual void accelerate();
    virtual void applyBrakes();
private:
    const string license;
};
```

Because `Car` no longer has a default constructor, the previous definition of `HybridCar` will cause a compile-time error because the `HybridCar` constructor cannot call the nonexistent default constructor for `Car`. How can we tell `HybridCar` to invoke the `Car` constructor with the proper arguments? The answer is similar to how we would construct a data member with a certain value – we use the initializer list. Here is a modified version of the `HybridCar` class that correctly initializes its `Car` base class:

```
class HybridCar: public Car
{
public:
    explicit HybridCar(const string& license) : Car(license) {}
    virtual void applyBrakes();
    virtual void accelerate();
private:
    void chargeBattery();
    void dischargeBattery();
};
```

Note that when using initializer lists to initialize base classes, you are only allowed to specify the names of *direct* base classes. As an example, suppose that we want to create a class called `ExperimentalHybridCar` that is similar to a `HybridCar` except that it contains extra instrumentation to monitor the state of the motor. Because `ExperimentalHybridCar` represents a prototype car, the car does not have a license plate, and so we want to communicate the string “None” up to `Car` to represent this information. Then if we define the `ExperimentalHybridCar` class as follows:

```
class ExperimentalHybridCar: public HybridCar
{
public:
    ExperimentalHybridCar();
    virtual void applyBrakes();
    virtual void accelerate();
};
```

It would be illegal to define the constructor as follows:

```
/* Note: This is not legal C++! */
ExperimentalHybridCar::ExperimentalHybridCar() : Car("None")
{
}
```

The problem with this code is that `Car` is an indirect base class of `ExperimentalHybridCar` and thus cannot be initialized from the `ExperimentalHybridCar` initializer list. The reason for this is simple. `ExperimentalHybridCar` inherits from `HybridCar`, which itself inherits from `Car`. What would happen if both `HybridCar` and `ExperimentalHybridCar` each tried to initialize `Car` in their initializer lists? Which constructor should take precedence? If it's `HybridCar`, then the initializer list for `ExperimentalHybridCar` would be ignored, leading to misleading code. If it's `ExperimentalHybridCar`, then if `HybridCar` needs to call the `Car` constructor with particular arguments, those arguments would be ignored and `HybridCar` might not be initialized correctly. To avoid this sort of confusion, C++ only lets you initialize direct base classes. Thus the proper version of the `ExperimentalHybridCar` constructor is as follows:

```
/* Tell HybridCar to initialize itself with the string "None" */
ExperimentalHybridCar::ExperimentalHybridCar() : HybridCar("None")
{
}
}
```

Since `HybridCar` forwards its constructor argument up to `Car`, this ends up producing the correct behavior.

Virtual Functions in Constructors

Let's take a quick look back at class construction for derived classes. If you'll recall, base classes are initialized before any of the derived class data members are set up. This means that there is a small window when the base class constructor executes where the base class is fully set up, but nothing in the derived class has yet been initialized. If the base class constructor could somehow access the data members of the derived class, it would read uninitialized memory and almost certainly crash the program. But this seems impossible – after all, the base class has no idea what's deriving from it, so how could it access any of the derived class's data members? Unfortunately, there is one way – virtual functions. Suppose the base class contains a virtual function and that one of the derived classes overrides that function to read a data member of the derived class. If the base class calls the virtual function in its constructor, it would be able to read the uninitialized value, causing a potential program crash.

The designers of C++ were well-aware of this edge case, and to prevent this error from occurring they added a restriction on the behavior of virtual function calls inside constructors. If you invoke a virtual function inside a class constructor, the function is *not* invoked polymorphically. That is, the virtual function call will always call the version of the function appropriate for the type of the base class rather than the type of the derived class. To see this in action, consider the following code:

```
class Base
{
public:
    Base()
    {
        fn();
    }
    virtual void fn()
    {
        cout << "Base" << endl;
    }
};
```

```

class Derived: public Base
{
public:
    virtual void fn()
    {
        cout << "Derived" << endl;
    }
};

```

Here, the `Base` constructor invokes its virtual function `fn`. While normally you would expect that this would invoke `Derived`'s version of `fn`, since we're inside the body of the `Base` constructor, the code will execute `Base`'s version of `fn`, which prints out “Base” instead of the expected “Derived.” Cases where you would invoke a virtual function in a constructor are rare, but if you plan on doing so remember that it will not behave as you might expect.

Everything we've discussed in this section has focused on class *constructors*, but these same restrictions apply to class *destructors* as well. C++ destructs classes from the outside inward, cleaning up derived classes before base classes, and if virtual functions were treated polymorphically inside destructors it would be possible to access data members of a derived class after they'd already been cleaned up.

Copy Constructors and Assignment Operators for Derived Classes

Copy constructors and assignment operators are complicated beasts that are even more perilous when mixed with inheritance. In particular, you must make sure to invoke the copy constructor and assignment operator for any base classes in addition to any other behavior. As an example, consider the following base class, which has a well-defined copy constructor and assignment operator:

```

class Base
{
public:
    Base();
    Base(const Base& other);
    Base& operator= (const Base& other);
    virtual ~Base();
private:
    /* ... implementation specific ... */
};

```

Now, consider the following derived class:

```

class Derived: public Base
{
public:
    Derived();
    Derived(const Derived& other);
    Derived& operator= (const Derived& other);
    virtual ~Derived();
private:
    char* theString; // Store a C string
    void copyOther(const Derived& other);
    void clear();
};

```

Using the template outlined in the chapter on copy functions, we might write the following code for the `Derived` assignment operator and copy constructor:


```

/* Generic "copy other" member function. */
void Derived::copyOther(const Derived& other)
{
    theString = new char[strlen(other.theString) + 1];
    strcpy(theString, other.theString);
}

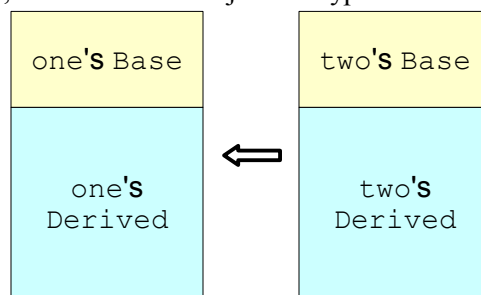
/* Clear-out member function. */
void Derived::clear()
{
    delete [] theString;
    theString = NULL;
}

/* Copy constructor. */
Derived::Derived(const Derived& other) // Wrong!
{
    copyOther(other);
}

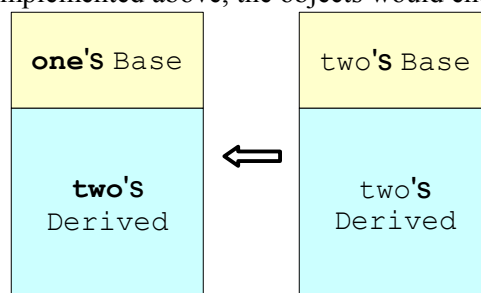
/* Assignment operator. */
Derived& Derived::operator= (const Derived& other) // Wrong!
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}

```

Initially, it seems like this code should work, but, alas, it is seriously flawed. During this copy operation, we never instructed C++ to copy over the data from `other`'s base class into the receiver object's base class. As a result, we'll end up with half-copied data, where the data specific to `Derived` is correctly cloned but `Base`'s data hasn't changed. To see this visually, if we have two objects of type `Derived` that look like this:



After invoking the copy functions implemented above, the objects would end up in this state:



We now have a partially-copied object, which will almost certainly crash at some point down the line.

When writing assignment operators and copy constructors for derived classes, you must make sure to manually invoke the assignment operators and copy constructors for base classes to guarantee that the object is fully-copied. Fortunately, this is not particularly difficult. Let's first focus on the copy constructor. Somehow, we need to tell the receiver's base object that it should initialize itself as a copy of the parameter's base object. Because *Derived is-a Base*, so we can pass the parameter to the *Derived* copy constructor as a parameter to *Base*'s copy constructor inside the initializer list. The updated version of the *Derived* copy constructor looks like this:

```
/* Copy constructor. */
Derived::Derived(const Derived &other) : Base(other) // Correct
{
    copyOther(other);
}
```

The code we have so far for the assignment operator correctly clears out the *Derived* part of the *Derived* class, but leaves the *Base* portion untouched. How should we go about assigning the *Base* part of the receiver object the *Derived* part of the parameter? Simple – we'll invoke the *Base*'s assignment operator and have *Base* do its own copying work. The code for this is a bit odd and is shown below:

```
/* Assignment operator. */
Derived& Derived::operator= (const Derived &other)
{
    if(this != &other)
    {
        clear();
        Base::operator= (other); // Invoke the assignment operator from Base.
        copyOther(other);
    }
    return *this;
}
```

Here we've inserted a call to *Base*'s assignment operator using the full name of the `operator =` function. This is one of the rare situations where you will need to use the full name of an overloaded operator. In case you're curious why just writing `*this = other` won't work, remember that this calls *Derived*'s version of `operator =`, causing infinite recursion.

All of the above discussion has assumed that your classes require their own assignment operator and copy constructor. However, if your derived class does not contain any data members that require manual copying and assignment (for example, a derived class that simply holds an `int`), none of the above code will be necessary. C++'s default assignment operator and copy constructor automatically invoke the assignment operator and copy constructor of any base classes, which is exactly what you'd want it to do.

Disallowing Copying

Using inheritance, it's possible to elegantly and concisely disallow copying for objects of a certain type. As mentioned above, a class's default copy constructor and assignment operator automatically invoke the copy constructor and assignment operator for any base classes. But what if for some reason the derived class can't call those functions? For example, suppose that we have the following class:

```

class Uncopyable
{
public:
    /* ... */
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator= (const Uncopyable&);
};

```

As mentioned in the chapter on copy constructors and assignment operators, this class cannot be copied because the copy constructor and assignment operator are marked private. What will happen if we then create a class that inherits from `Uncopyable`, as shown here:

```

class MyClass: public Uncopyable
{
    /* ... */
};

```

Let's assume that `MyClass` does not explicitly declare a copy constructor or assignment operator. This will cause C++ to try to create a default implementation for these functions. In the process of doing so, the compiler will realize that it needs to call the copy constructor and assignment operator of `Uncopyable`. But these functions are `private`, meaning that the derived class `MyClass` can't access them. Rather than reporting this as an error, instead the compiler doesn't create default implementations of these functions. This means that `MyClass` has no copy constructor or assignment operator, not even default implementations, and thus can't be copied or assigned. We've successfully disallowed copying!

However, by inheriting from `Uncopyable`, we've introduced some undesirable behavior. It is now legal for clients of `MyClass` to treat `MyClass` as though it were an `Uncopyable`, as shown here:

```

MyClass* mc = new MyClass;
Uncopyable* uPtr = mc;

```

This is unfortunate, since `Uncopyable` is an implementation detail of `MyClass`, not a supertype.

We are now in a rather interesting situation. We want to absorb the functionality provided by another class, but don't want to make our type a subtype of that class in the process. In other words, we want to absorb an *implementation* without its corresponding *interface*. Fortunately, using a technique called *private inheritance*, we can express this notion precisely.

So far, the inheritance you have seen has been *public inheritance*. When a class publicly inherits from a base class, it absorbs the public interface of the base class along with any implementations of the functions in that interface. In private inheritance, a derived class inherits from a base class solely to acquire its implementation. While the derived class retains the implementation of all `public` member functions from the base class, those functions become `private` in the derived class. For example, given these two classes:

```

class Base
{
public:
    void doSomething();
};

class Derived: private Base
{
    /* ... */
};

```

The following code is illegal:

```
Derived d;
d.doSomething(); // Error
```

Even though `doSomething` was declared `public` in `Base`, because `Derived` inherits privately from `Base` the `doSomething` member function is `private`.

Additionally, `private` inheritance does *not* define a subtyping relationship. That is, the following code is illegal:

```
Base* ptr = new Derived; // Error
```

While `public` inheritance models the *is-a* relationship, `private` inheritance represents the *is-implemented-in-terms-of* relationship. For example, we might use `private` inheritance to implement a `stack` in terms of a `deque`, since a `stack`'s entire functionality can be expressed through proper calls to `push_front`, `front`, and `pop_front`. This is shown here for a `stack` of integers:

```
class stack: private deque<int>
{
public:
    void push(int val)
    {
        push_front(val); // Calls deque<int>::push_front.
    }
    int pop()
    {
        const int result = front();
        pop_front();
        return result;
    }
    /* ... etc. ... */
};
```

Notice that `push` is implemented as a call to the `deque`'s `push_front` function, while `pop` is implemented through a series of calls to `front` and `pop_front`. Because we privately inherited from `deque<int>`, our class contains an implementation of all of the `deque`'s member functions, and it is as if we have our own private copy of a `deque` that we can work with.

`Public` and `private` inheritance are designed for entirely different purposes. We use `public` inheritance to design a collection of classes logically related to each other by some common behaviors. `Private` inheritance, on the other hand, is an implementation technique used to define one class's behaviors in terms of another's. One way to remember the difference between `public` and `private` inheritance is to recognize that they play entirely different roles in class design. `Public` inheritance is used during the design of a class *interface* (determining what behaviors the class should provide), while `private` inheritance is used during design of a class *implementation* (how those behaviors should be performed). This parallels the difference between a function prototype and a function definition – `public` inheritance defines a set of prototypes, while `private` inheritance provides implementations.

`Private` inheritance is not frequently encountered in practice because the *is-implemented-in-terms-of* relationship can be modeled more clearly through composition. If we wanted to implement a `stack` in terms of a `deque`, instead of using `private` inheritance, we could just have the `stack` contain a `deque` as a data member, as shown here:

```

class stack
{
public:
    void push(int val)
    {
        implementation.push_front(val);
    }
    int pop()
    {
        const int result = implementation.front();
        implementation.pop_front();
        return result;
    }
    /* ... etc. ... */
private:
    deque<int> implementation;
};

```

In practice it is recommended that you shy away from private inheritance in favor of this more explicit form of composition. However, there are several cases where private inheritance is precisely the tool for the job. Let's return to our discussion of the `Uncopyable` class. Recall that to make a class uncopyable, we had it publically inherit from a class `Uncopyable` that has its copy functions marked private. This led to problems where we could convert an object that inherited from `Uncopyable` into an `Uncopyable`. However, we can remedy this by having the derived class inherit *privately* from `Uncopyable`. That way, it is not considered a subtype of `Uncopyable` and instances of `MyClass` cannot be converted into instances of `Uncopyable`. For example:

```

class MyClass: private Uncopyable
{
    /* ... */
};

```

Now, `MyClass` cannot be copied, nor can it be treated as though it were an object of type `Uncopyable`. This is precisely the idea we want to express.

In C++, all inheritance is considered private inheritance unless explicitly mentioned otherwise; this is why you must write `public Base` to publicly inherit from `Base`. Thus we can rewrite the above class definition omitting the `private` keyword, as shown here:

```

class MyClass: Uncopyable
{
    /* ... */
};

```

This method of disallowing copying is particularly elegant – syntactically, we communicate that `MyClass` cannot be copied at the same time that we actually make it uncopyable through private inheritance.

Before concluding this section, let's make a quick change to the `Uncopyable` class by marking its constructor and destructor `protected`. This means that classes that inherit from `Uncopyable` can still access the constructor and destructor, but otherwise these functions are off-limits. This prevents us from accidentally instantiating `Uncopyable` directly and only lets us use it as a base class. The code for this is shown here:

```

class Uncopyable
{
protected:
    Uncopyable() {}
    ~Uncopyable() {}
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator= (const Uncopyable&);
};

```

Classes like `Uncopyable` are sometimes referred to as *mixin classes* since they are designed to be “mixed” into other classes to provide a particular functionality.

Slicing

In our discussion of copy constructor and assignment operators for derived classes, we encountered problems when we copied over the the data of the `Derived` class but not the `Base` class. However, there's a far more serious problem we can run into called *slicing* where we copy only the base class of an object while leaving its derived classes unmodified.

Suppose we have two `Base *` pointers called `one` and `two` that point to objects either of type `Base` or of type `Derived`. What happens if we write code like `*one = *two`? Here, we're copying the value of the object pointed at by `two` into the variable pointed at by `one`. While at first glance this might seem harmless, the above statement is one of the most potentially dangerous mistakes you can make when working with C++ inheritance. The problem is that this line expands into a call to

```
one->operator =(*two);
```

Note that the version of `operator =` we're calling here is the one defined in `Base`, not `Derived`, so this line will only copy over the `Base` portion of `two` into the `Base` portion of `one`, resulting in half-formed objects that are almost certainly not in the correct state and may be entirely corrupted.

Slicing can be even more insidious in scenarios like this one:

```

void DoSomething(Base baseObject)
{
    // Do something
}

```

```

Derived myDerived
DoSomething(myDerived);

```

Recall that the parameter `baseObject` will be initialized using the `Base` copy constructor, not the `Derived` copy constructor, so the object in `DoSomething` will not be a correct copy `myDerived`. Instead, it will only hold a copy of the `Base` part of the `myDerived` object.

You should almost never assign a base class object the value of a derived class. The second you do, you will open the door to runtime errors as your code tries to use incompletely-formed objects. While it may sound simple to follow this rule, at many times it might not be clear that you're slicing an object. For example, consider this code snippet:

```

vector<Base> myBaseVector;
Base* myBasePtr = SomeFunction();
myBaseVector.push_back(*myBasePtr);

```

Here, the object pointed at by `myBasePtr` could be of type `Base` or any type inheriting from `Base`. When we call `myBaseVector.push_back(*myBasePtr)`, there is a good chance that we will slice the object pointed at by `myBasePtr`, storing only its `Base` component in the `vector` and dropping the rest. You'll need to be extra vigilant when working with derived classes, since it's easy to generate dangerous, difficult-to-track bugs.

The C++ Casting Operators

One of the most useful features of C++ inheritance is the ability to use an object of one type in place of another. For example, a pointer of type `Derived *` can be used whenever a `Base *` would be expected, and the conversion is automatic. However, in many circumstances, we may want to perform this conversion in reverse. Suppose that we have a pointer that's statically typed as a `Base *`, but we know that the object it points to is actually of type `Derived *`. How can we use the `Derived` features of the pointee? Because the pointer to the object is a `Base *`, not a `Derived *`, we will have to use a `typeid` to convert the pointer from the base type to the derived type. Using the `typeid` casts most familiar to us in C++, the code to perform this conversion looks as follows:

```
Base* myBasePtr; // Assume we know that this points to a Derived object.
Derived* myDerivedPtr = (Derived *)myBasePtr;
```

There is nothing wrong with the above code as written, but it is risky because of the `typeid` `(Derived *)myBasePtr`. In C++, using a `typeid` of the form `(Type)` is extremely dangerous because there are only minimal compile-time checks to ensure that the `typeid` makes any sense. For example, consider the following C++ code:

```
Base* myBasePtr; // Assume we know that this points to a Derived object.
vector<double>* myVectorPtr = (vector<double> *)myBasePtr; // Uh oh!
```

This code is completely nonsensical, since there is no reasonable way that a pointer of type `Base *` can end up pointing to an object of type `vector<double>`. However, because of the explicit pointer-to-pointer `typeid`, this code is entirely legal. In the above case, it's clear that the conversion we're performing is incorrect, but in others it might be more subtle. Consider the following code:

```
const Base* myBasePtr; // Assume we know that this points to a Derived object.
Derived* myDerivedPtr = (Derived *)myBasePtr;
```

This code again is totally legal and at first glance might seem correct, but unfortunately it contains a serious error. In this example, our initial pointer was a pointer to a `const Base` object, but in the second line we removed that `constness` with a `typeid` and the resulting pointer is free to modify the object it points at. We've just subverted `const`, which could lead to a whole host of problems down the line.

The problem with the above style of C++ `typeid` is that it's just too powerful. If C++ can figure out a way to convert the source object to an object of the target type, it will, even if it's clear from the code that the conversion is an error. To resolve this issue, C++ has four special operators called *casting operators* that you can use to perform safer `typeid`s. When working with inheritance, two of these casting operators are particularly useful, the first of which is `static_cast`. The `static_cast` operator performs a `typeid` in the same way that the more familiar C++ `typeid` does, except that it checks at compile time that the cast “makes sense.” More specifically, `static_cast` can be used to perform the following conversions:

* There are several other conversions that you can perform with `static_cast`, especially when working with `void *` pointers, but we will not discuss them here.

1. Converting between primitive types (e.g. `int` to `float` or `char` to `double`).
2. Converting between pointers or references of a derived type to pointers or references of a base type (e.g. `Derived *` to `Base *`) where the target is at least as `const` as the source.
3. Converting between pointers or references of a base type to pointers or references of a derived type (e.g. `Base *` to `Derived *`) where the target is at least as `const` as the source.

If you'll notice, neither of the errors we made in the previous code snippets are possible with a `static_cast`. We can't convert a `Base *` to a `vector<double> *`, since `Base` and `vector<double>` are not related to each other via inheritance. Similarly, we cannot convert from a `const Base *` to a `Derived *`, since `Derived *` is less `const` than `const Base *`.

The syntax for the `static_cast` operator looks resembles that of templates and is illustrated below:

```
Base* myBasePtr; // Assume we know this points to a Derived object.
Derived* myDerivedPtr = static_cast<MyDerived *>(myBasePtr);
```

That is, `static_cast`, followed by the type to convert the pointer to, and finally the expression to convert enclosed in parentheses.

Throughout this discussion of typecasts, when converting between pointers of type `Base *` and `Derived *`, we have implicitly assumed that the `Base *` pointer we wanted to convert was pointing to an object of type `Derived`. If this isn't the case, however, the typecast can succeed but lead to a `Derived *` pointer pointing to a `Base` object, which can cause all sorts of problems at runtime when we try to invoke nonexistent member functions or access data members of the `Derived` class that aren't present in `Base`. The problem is that the `static_cast` operator doesn't check to see that the typecast it's performing makes any sense at runtime. To provide this functionality, you can use another of the C++ casting operators, `dynamic_cast`, which acts like `static_cast` but which performs additional checks before performing the cast. `dynamic_cast`, like `static_cast`, can be used to convert between pointer types related by inheritance (but not to convert between primitive types). However, if the typecast requested of `dynamic_cast` is invalid at runtime (e.g. attempting to convert a `Base` object to a `Derived` object), `dynamic_cast` will return a `NULL` pointer instead of a pointer to the derived type. For example, consider the following code:

```
Base* myBasePtr = new Base;
Derived* myDerivedPtr1 = (Derived *)myBasePtr;
Derived* myDerivedPtr2 = static_cast<Derived *>(myBasePtr);
Derived* myDerivedPtr3 = dynamic_cast<Derived *>(myBasePtr);
```

In this example, we use three different typecasts to convert a pointer that points to an object of type `Base` to a pointer to a `Derived`. In the above example, the first two casts will perform the type conversion, resulting in pointers of type `Derived *` that actually point to a `Base` object, which can be dangerous. However, the final typecast, which uses `dynamic_cast`, will return a `NULL` pointer because the cast cannot succeed.

When performing downcasts (casts from a base type to a derived type), unless you are absolutely sure that the cast will succeed, you should consider using `dynamic_cast` over a `static_cast` or raw C++ typecast. Because of the extra check at runtime, `dynamic_cast` is slower than the other two casts, but the extra safety is well worth the cost.

There are two more interesting points to take note of when working with `dynamic_cast`. First, you can only use `dynamic_cast` to convert between types if the base class type contains at least one virtual member function. This may seem like a strange requirement, but greatly improves the efficiency of the language as a whole and makes sense when you consider that it's rare to hold a pointer to a `Derived` in a pointer of type `Base` when `Base` isn't polymorphic. The other important note is that if you use `dynamic_cast` to convert between *refer-*

ences rather than pointers, `dynamic_cast` will throw an object of type `bad_cast` rather than returning a “NULL reference” if the cast fails. Consult a reference for more information on `bad_cast`.

Implicit Interfaces and Explicit Interfaces

In the chapter on templates we discussed the concept of an *implicit interface*, behaviors required of a template argument. For example, consider a function that returns the average of the values in an STL container of doubles, as shown here:

```
template <typename ContainerType> double GetAverage(const ContainerType& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

The function `GetAverage` may be parameterized over an arbitrary type, but will only compile if the type `ContainerType` exports functions `begin` and `end` that return iterators over doubles (or objects implicitly convertible to doubles) and a `size` function that returns some integer type.

Contrast this with a similar function that uses inheritance:

```
class Container
{
public:
    typedef something-dereferencable-to-a-double const_iterator;
    virtual const_iterator begin() const = 0;
    virtual const_iterator end() const = 0;
};

double GetAverage(const Container& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

This function is no longer a template and instead accepts as an argument an object that derives from `Container`.

In many aspects these functions are similar. Both of the implementations have a set of constraints enforced on their parameter, which can be of any type satisfying these constraints. But these similarities obscure a fundamental difference between how the two functions work – at what point the function calls are resolved. In the inheritance-based version of `GetAverage`, the calls to `begin`, `end`, and `size` are virtual function calls which are resolved at *runtime* using the system described earlier in this chapter. With the template-based version of `GetAverage`, the version of the `begin`, `end`, and `size` functions to call are resolved at *compile-time*.

When you call a template function, C++ instantiates the template by replacing all instances of the template argument with a concrete type. Thus if we call the template function `GetAverage` on a `vector<int>`, the compiler will instantiate `GetAverage` on `vector<int>` to yield the following code:

```
double GetAverage<vector<int>>(const vector<int>& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

Now that the template has been instantiated, it's clear what functions `c.begin()` and the like correspond to – they're the `vector<int>`'s versions of those functions. Since those functions aren't virtual, the compiler can hardcode which function is being called and can optimize appropriately.

The template version of this function is desirable from a performance perspective but is not always the best option. In particular, while we can pass as a parameter to `GetAverage` any object that conforms to the implicit interface, we cannot treat those classes polymorphically. For example, in the above case it's perfectly legal to call `GetAverage` on a `vector<double>` or a `set<double>`, but we cannot write code like this:

```
Container* ptr = RandomChance(0.5) ? new vector<double> : new set<double>;
```

Templates and inheritance are designed to solve fundamentally different problems. If you want to write code that operates over any type that meets some minimum requirements, the template system can help you do so efficiently and concisely provided that you know what types are being used at compile-time. If the types cannot be determined at compile-time, you can use inheritance to describe what behaviors are expected of function arguments.

More to Explore

1. **Multiple Inheritance:** Unlike other object-oriented languages like Java, C++ lets classes inherit from multiple base classes. You can use this build classes that act like objects of multiple types, or in conjunction with mixin classes to build highly-customizable classes. In most cases multiple inheritance is straightforward and simple, but there are many situations where it acts counterintuitively. If you plan on pursuing C++ more seriously, be sure to read up on multiple inheritance.
2. **`const_cast` and `reinterpret_cast`:** C++ has two other conversion operators, `const_cast`, which can add or remove `const` from a pointer, and `reinterpret_cast`, which performs fundamentally unsafe typecasts (such as converting an `int *` to a `string *`). While the use cases of these operators are far beyond the scope of this class, they do arise in practice and you should be aware of their existences. Consult a reference for more information.
3. **The Curiously Recurring Template Pattern (CRTP):** Virtual functions make your programs run slightly slower than programs with non-virtual functions because of the extra overhead of the dynamic lookup. In certain situations where you want the benefits of inheritance without the cost of virtual functions, you can use an advanced C++ trick called the *curiously recurring template pattern*, or CRTP. The CRTP is also known as “static interfacing” and lets you get some of the benefits of inheritance without any runtime cost.
4. **Policy Classes:** A nascent but popular C++ technique called *policy classes* combines multiple inheritance and templates to design classes that are simultaneously customizable and efficient. A full treatment of policy classes is far beyond the scope of this reader, but if you are interested in seeing exactly how customizable C++ can be I strongly encourage you to read more about them.

Practice Problems

1. In the `GenericBatchPrinter` example from earlier in this chapter, recall that the `Document` base class had several methods defined purely virtually, meaning that they don't actually have any code for those member functions. Inside `GenericBatchPrinter`, why don't we need to worry that the `Document *` pointer from the `queue` points to an object of type `Document` and thus might cause problems if we tried invoking those purely virtual functions? ♦
2. In the next exercises, we'll explore a set of classes that let you build and modify functions at runtime using tools similar to those in the STL `<functional>` programming library.

Consider the following abstract class:

```
class Function
{
public:
    virtual ~Function() = 0;
    virtual double evaluateAt(double value) = 0;
};
```

This class exports a single function, `evaluateAt`, that accepts a `double` as a parameter and returns the value of some function evaluated at that point. Write a derived class of `Function`, `SimpleFunction`, whose constructor accepts a regular C++ function that accepts and returns a `double` and whose `evaluateAt` function returns the value of the stored function evaluated at the parameter.

3. The composition of two functions **F** and **G** is defined as **F(G(x))** – that is, the function **F** applied to the value of **G** applied to **x**. Write a class `CompositionFunction` whose constructor accepts two `Function *` pointers and whose `evaluateAt` returns the composition of the two functions evaluated at a point.
4. The derivative of a function is the slope of the tangent line to that function at a point. The derivative of a function **F** can be approximated as $F'(x) \approx (F(x + \Delta x) - F(x - \Delta x)) / 2\Delta x$ for small values of Δx . Write a class `DerivativeFunction` whose constructor accepts a `Function *` pointer and a `double` representing Δx and whose `evaluateAt` approximates the derivative of the stored function using the initial value of Δx . ♦
5. For the above classes, why did we make a function named `evaluateAt` instead of providing an implementation of `operator()`? (*Hint: Will we be using actual Function objects, or pointers to them?*)
6. A common mistake is to try to avoid problems with slicing by declaring `operator =` as a virtual function in a base class. Why won't this solve the slicing problem? (*Hint: what is the parameter type to `operator =`?*)
7. Suppose you have three classes, `Base`, `Derived`, and `VeryDerived` where `Derived` inherits from `Base` and `VeryDerived` inherits from `Derived`. Assume all three have copy constructors and assignment operators. Inside the body of `VeryDerived`'s assignment operator, why shouldn't it invoke `Base::operator =` on the other object? What does this tell you about long inheritance chains, copying, and assignment?
8. The C++ casting operators were deliberately designed to take up space to discourage programmers from using typecasts. Explain why the language designers frowned upon the use of typecasts.
9. The `unary_function` and `binary_function` classes from `<functional>` do not define `operator()` as a virtual member function. Considering that every adaptable function must be a subclass of one of these two classes, it seems logical for the two classes to do so. Why don't they? (*Hint: the STL is designed for maximum possible efficiency. What would happen if `operator()` was virtual?*)

Chapter 30: Extended Example: Function

There are a huge number of ways to define a function or function-like object in C++, each of which has slightly different syntax and behavior. For example, suppose that we want to write a function that accepts as input a function that can accept an `int` and return a `double`. While of course we could accept a `double (*) (int)` – a pointer to a function accepting an `int` and returning a `double` – this is overly restrictive. For example, all of the following functions can be used as though they were functions taking in an `int` and returning a `double`:

```
double Fn1(const int&);    /* Accept int by reference-to-const, return double. */
int      Fn2(int);         /* Accept parameter as a int, return int. */
int      Fn3(const int&);  /* Accept reference-to-const int, return int. */
```

In addition, if we just accept a `double (*) (int)`, we also can't accept functors as input, meaning that neither of these objects below – both of which can accept an `int` and return a `double` – could be used:

```
/* Functor accepting an int and returning a double. */
struct MyFunctor
{
    double operator() (int);
};

/* Adaptable function accepting double and returning a double. */
bind2nd(multiplies<int>(), 137);
```

In the chapter on functors, we saw how we can write functions that accept any of the above functions using templates, as shown here:

```
template <typename UnaryFunction> void DoSomething(UnaryFunction fn)
{
    /* ... */
}
```

If we want to write a *function* that accepts a function as input we can rely on templates, but what if we want to write a *class* that needs to store a function of any arbitrary type? To give a concrete example, suppose that we're designing a class representing a graphical window and we want the client to be able to control the window's size and position. The window object, which we'll assume is of type `Window`, thus allows the user to provide a function that will be invoked whenever the window is about to change size. The user's function then takes in an `int` representing the potential new width of the window and returns an `int` representing what the user wants the new window size to be. For example, if we want to create a window that can't be more than 100 pixels wide, we could pass in this function:

```
int ClampTo100Pixels(int size)
{
    return min(size, 100);
}
```

Alternatively, if we want the window size to always be 100 pixels, we could pass in this function:

```
int Always100Pixels(int unused)
{
    return 100; // Ignore parameter
}
```

Given that we need to store a function of an arbitrary type inside the `Window` class, how might we design `Window`? Using the approach outlined above, we could templatize the `Window` class over the type of the function it stores, as shown here:

```
template <typename WidthFunction> class Window
{
public:
    Window(WidthFunction fn, /* ... */);

    /* ... */

private:
    WidthFunction width;

    /* ... */
};
```

Given this implementation of `Window`, we could then specify that a window should be no more than 100 pixels wide by writing

```
Window<int (*)(int)> myWindow(ClampTo100Pixels);
```

This `Window` class lets us use any reasonable function to determine the window size, but has several serious drawbacks. First, it requires the `Window` client to explicitly parameterize `Window` over the type of callback being stored. When working with function pointers this results in long and convoluted variable declarations (look above for an example), and when working with library functors such as those in the STL `<functional>` library (e.g. `bind2nd(ptr_fun(MyFunction), 137)`)*, we could end up with a `Window` of such a complicated type that it would be infeasible to use without the aid of `typedef`. But a more serious problem is that this approach causes two `Windows` that don't use the same type of function to compute width to have completely different types. That is, a `Window` using a raw C++ function to compute its size would have a different type from a `Window` that computed its size with a functor. Consequently, we couldn't make a `vector<Window>`, but instead would have to make a `vector<Window<int (*)(int)>>` or a `vector<Window<MyFunctorType>>`. Similarly, if we want to write a function that accepts a `Window`, we can't just write the following:

```
void DoSomething(const Window& w) // Error - Window is a template, not a type
{
    /* ... */
}
```

We instead would have to write

```
template <typename WindowType>
void DoSomething(const WindowType& w) // Legal but awkward
{
    /* ... */
}
```

It should be clear that templatizing the `Window` class over the type of the callback function does not work well. How can we resolve this problem? In this extended example, we'll explore how to solve this problem using inheritance, then will abstract away from the concrete problem at hand to create a versatile class encapsulating a function of any possible type.

* As an FYI, the type of `bind2nd(ptr_fun(MyFunction), 137)` is

```
binder2nd<pointer_to_binary_function<Arg1, Arg2, Ret>>
```

where `Arg1`, `Arg2`, and `Ret` are the argument and return types of the `MyFunction` function.

Inheritance to the Rescue

Let's take a few minutes to think about the problem we're facing. We have a collection of different objects that each have similar functionality (they can be called as functions), but we don't know exactly which object the user will provide. This sounds exactly like the sort of problem we can solve using inheritance and virtual functions. But we have a problem – inheritance only applies to *objects*, but some of the values we might want to store are simple function pointers, which are primitives. Fortunately, we can apply a technique called the *Fundamental Rule of Software Engineering* to solve this problem:

Theorem (The Fundamental Theorem of Software Engineering): Any problem can be solved by adding enough layers of indirection.

This is a very useful programming concept that will prove relevant time and time again – make sure you remember it!

In this particular application, the FTSE says that we need to distance ourselves by one level from raw function pointers and functor classes. This leads to the following observation: while we might not be able to treat functors and function pointers polymorphically, we certainly can create a new class hierarchy and then treat that class polymorphically. The idea goes something like this – rather than having the user provide us a functor or function pointer which could be of any type, instead we'll define an abstract class exporting the callback function, then will have the user provide a subclass which implements the callback.

One possible base class in this hierarchy is shown below:

```
class IntFunction
{
public:
    /* Polymorphic classes should have virtual destructors. */
    virtual ~IntFunction() { }

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;
};
```

`IntFunction` exports a single function called `execute` which accepts an `int` and returns an `int`. This function is marked purely virtual since it's unclear exactly what this function should do. After all, we're trying to store an arbitrary function, so there's no clearly-defined default behavior for `execute`.

We can now modify the implementation of `Window` to hold a pointer to an `IntFunction` instead of being templated over the type of the function, as shown here:

```
class Window
{
public:
    Window(IntFunction* sizeFunction, /* ... */)

    /* ... */
private:
    IntFunction* fn;
};
```

Now, if we wanted to clamp the window to 100 pixels, we can do the following:

```

class ClampTo100PixelsFunction: public IntFunction
{
public:
    virtual int execute(int size) const
    {
        return min(size, 100);
    }
};

Window myWindow(new ClampTo100PixelsFunction, /* ... */);

```

Similarly, if we want to have a window that's always 100 pixels wide, we could write

```

class FixedSizeFunction: public IntFunction
{
public:
    virtual int execute(int size) const
    {
        return 100;
    }
};

Window myWindow(new FixedSizeFunction, /* ... */);

```

It seems as though we've solved the problem – we now have a `Window` class that allows us to fully customize its resizing behavior – what more could we possibly want?

The main problem with our solution is the sheer amount of boilerplate code clients of `Window` have to write if they want to change the window's resizing behavior. Our initial goal was to let class clients pass raw C++ functions and functors to the `Window` class, but now clients have to subclass `IntFunction` to get the job done. Both of the above subclasses are lengthy even though they only export a single function. Is there a simpler way to do this?

The answer, of course, is yes. Suppose we have a pure C++ function that accepts an `int` by value and returns an `int` that we want to use for our resizing function in the `Window` class; perhaps it's the `ClampTo100Pixels` function we defined earlier, or perhaps it's `Always100Pixels`. Rather than defining a new subclass of `IntFunction` for every single one of these functions, instead we'll build a single class that's designed to wrap up a function pointer in a package that's compatible with the `IntFunction` interface. That is, we can define a subclass of `IntFunction` whose constructor accepts a function pointer and whose `execute` calls this function. This is the Fundamental Theorem of Software Engineering in action – we couldn't directly treat the raw C++ function polymorphically, but by abstracting by a level we can directly apply inheritance.

Here's one possible implementation of the subclass:

```

class ActualFunction: public IntFunction
{
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }
private:
    int (*function)(int);
};

```

Now, if we want to use `ClampTo100Pixels` inside of `Window`, we can write:


```
Window myWindow(new ActualFunction(ClampTo100Pixels), /* ... */);
```

There is a bit of extra code for creating the `ActualFunction` object, but this is a one-time cost. We can now use `ActualFunction` to wrap any raw C++ function accepting an `int` and returning an `int` and will save a lot of time typing out new subclasses of `IntFunction` for every callback.

Now, suppose that we have a functor class, which we'll call `MyFunctor`, that we want to use inside the `Window` class. Then we could define a subclass that looks like this:

```
class MyFunctorFunction: public IntFunction
{
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }
private:
    MyFunctor function;
};
```

And could then use it like this:

```
Window myWindow(new MyFunctorFunction(MyFunctor(137)), /* ... */);
```

Where we assume for simplicity that the `MyFunctor` class has a unary constructor.

We're getting much closer to an ideal solution. Hang in there; the next step is pretty exciting.

Templates to the Rescue

At this point we again could just call it quits – we've solved the problem we set out to solve and using the above pattern our `Window` class can use any C++ function or functor we want. However, we are close to an observation that will greatly simplify the implementation of `Window` and will yield a much more general solution.

Let's reprint the two subclasses of `IntFunction` we just defined above which wrap function pointers and functors:

```

class ActualFunction: public IntFunction
{
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }
private:
    int (*const function)(int);
};

class MyFunctorFunction: public IntFunction
{
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }
private:
    MyFunctor function;
};

```

If you'll notice, besides the name of the classes, the only difference between these two classes is what type of object is being stored. This similarity is no coincidence – any callable function or functor would require a subclass with exactly this structure. Rather than requiring the client of `Window` to reimplement this subclass from scratch each time, we can instead create a *template class* that's a subclass of `IntFunction`. It's rare in practice to see templates and inheritance mixed this way, but here it works out beautifully. Here's one implementation:

```

template <typename UnaryFunction> class SpecificFunction: public IntFunction
{
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }
private:
    UnaryFunction function;
};

```

We now can use the `Window` class as follows:

```

Window myWindow(new SpecificFunction<int (*) (int)>(ClampTo100Pixels), /* ... */);
Window myWindow(new SpecificFunction<MyFunctor>(MyFunctor(137)), /* ... */);

```

The syntax here might be a bit tricky, but we've greatly reduced the complexity associated with the `Window` class since clients no longer have to implement their own subclasses of `IntFunction`.

One More Abstraction

This extended example has consisted primarily of adding more and more abstractions on top of the system we're designing, and it's time for one final leap. Let's think about what we've constructed so far. We've built a class

hierarchy with a single base class and a template for creating as many subclasses as we need. However, everything we've written has been hardcoded with the assumption that the `Window` class is the only class that might want this sort of functionality. Having the ability to store and call a function of any conceivable type is enormously useful, and if we can somehow encapsulate all of the necessary machinery to get this working into a single class, we will be able to reuse what we've just built time and time again. In this next section, that's exactly what we'll begin doing.

We'll begin by moving the code from `Window` that manages the stored function into a dedicated class called `Function`. The basic interface for `Function` is shown below:

```
class Function
{
public:
    /* Constructor and destructor. We'll implement copying in a bit. */
    Function(IntFunction* fn);
    ~Function();

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;
};
```

We'll leave the `Function` constructor left as an implicit conversion constructor, since that way we can implicitly convert between a callable `IntFunction` pointer and a `Function` functor. We can then implement the above functions as follows:

```
/* Constructor accepts an IntFunction and stores it. */
Function::Function(IntFunction* fn) : function(fn)
{
}

/* Destructor deallocates the stored function. */
Function::~~Function()
{
    delete function;
}

/* Function call just calls through to the pointer and returns the result. */
int Function::operator() (int value) const
{
    return function->execute(value);
}
```

Nothing here should be too out-of-the-ordinary – after all, this is pretty much the same code that we had inside the `Window` class.

Given this version of `Function`, we can write code that looks like this:

```
Function myFunction = new SpecificFunction<int (*) (int)>(ClampTo100Pixels);
cout << myFunction(137) << endl; // Prints 100
cout << myFunction(42) << endl; // Prints 42
```

If you're a bit worried that the syntax `new SpecificFunction<int (*) (int)>(ClampTo100Pixels)` is unnecessarily bulky, that's absolutely correct. Don't worry, in a bit we'll see how to eliminate it. In the meantime, however, let's implement the copy behavior for the `Function` class. After all, there's no reason that we

shouldn't be able to copy `Function` objects, and defining copy behavior like this will lead to some very impressive results in a bit.

We'll begin by defining the proper functions inside the `Function` class, as seen here:

```
class Function
{
public:
    /* Constructor and destructor. */
    Function(IntFunction* fn);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

Now, since the `Function` class contains only a single data member (the `IntFunction` pointer), to make a deep-copy of a `Function` we simply need to make a deep copy of its requisite `IntFunction`. But here we run into a problem. `IntFunction` is an abstract class and we can't tell at compile-time what type of object is actually being pointed at by the function pointer. How, then, can we make a deep-copy of the `IntFunction`? The answer is surprisingly straightforward – we'll just introduce a new virtual function to the `IntFunction` class that returns a deep copy of the receiver object. Since this function duplicates an existing object, we'll call it `clone`. The interface for `IntFunction` now looks like this:

```
class IntFunction
{
public:
    /* Polymorphic classes should have virtual destructors. */
    virtual ~IntFunction() { }

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;

    /* clone() returns a deep-copy of the receiver object. */
    virtual IntFunction* clone() const = 0;
};
```

We can then update the template class `SpecificFunction` to implement `clone` as follows:

```

template <typename UnaryFunction> class SpecificFunction: public IntFunction
{
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const
    {
        return function(value);
    }

    virtual IntFunction* clone() const
    {
        return new SpecificFunction(*this);
    }
private:
    UnaryFunction function;
};

```

Here, the implementation of `clone` returns a new `SpecificFunction` initialized via the copy constructor as a copy of the receiver object. Note that we haven't explicitly defined a copy constructor for `SpecificFunction` and are relying here on C++'s automatically-generated copy function to do the trick for us. This assumes, of course, that the `UnaryFunction` type correctly supports deep-copying, but this isn't a problem since raw function pointers can trivially be deep-copied as can all primitive types and it's rare to find functor classes with no copy support.

We can then implement the copy constructor, assignment operator, destructor, and helper functions for `Function` as follows:

```

Function::~~Function()
{
    clear();
}

Function::Function(const Function& other)
{
    copyOther(other);
}

Function& Function::operator= (const Function& other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}

void Function::clear()
{
    delete function;
}

void Function::copyOther(const Function& other)
{
    /* Have the stored function tell us how to copy itself. */
    function = other.function->clone();
}

```

Our `Function` class is now starting to take shape!

Hiding `SpecificFunction`

Right now our `Function` class has full deep-copy support and using `SpecificFunction<T>` can store any type of callable function. However, clients of `Function` have to explicitly wrap any function they want to store inside `Function` in the `SpecificFunction` class. This has several problems. First and foremost, this breaks encapsulation. `SpecificFunction` is only used internally to the `Function` class, never externally, so requiring clients of `Function` to have explicit knowledge of its existence violates encapsulation. Second, it requires the user to know the type of every function they want to store inside the `Function` class. In the case of `ClampTo100Pixels` this is rather simple, but suppose we want to store `bind2nd(multiplies<int>(), 137)` inside of `Function`. What is the type of the object returned by `bind2nd(multiplies<int>(), 137)`? For reference, it's `binder2nd<multiplies<int> >, >`, so if we wanted to store this in a `Function` we'd have to write

```
Function myFunction =
new SpecificFunction<binder2nd<multiplies<int> >, >>(bind2nd(multiplies<int>(), 137));
```

This is a syntactic nightmare and makes the `Function` class terribly unattractive.

Fortunately, however, this problem has a quick fix – we can templatize the `Function` constructor over the type of argument passed into it, then construct the relevant `SpecificFunction` for the `Function` client. Since C++ automatically infers the parameter types of template functions, this means that clients of `Function` never need to know the type of what they're storing – the compiler will do the work for them. Excellent!

If we do end up making the `Function` constructor a template, we should also move the `IntFunction` and `SpecificFunction` classes so that they're inner classes of `Function`. After all, they're specific to the implementation of `Function` and the outside world has no business using them.

The updated interface for the `Function` class is shown here:

```
class Function
{
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    class IntFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ };

    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

We can then implement the constructor as follows:

```
template <typename UnaryFunction> Function::Function(UnaryFunction fn)
{
    function = new SpecificFunction<UnaryFunction>(fn);
}
```

Since we've left the `Function` constructor not marked `explicit`, this template constructor is a conversion constructor. Coupled with the assignment operator, this means that we can use `Function` as follows:

```
Function fn = ClampTo100Pixels;
cout << fn(137) << endl; // Prints 100
cout << fn(42) << endl; // Prints 42

fn = bind2nd(multiplies<int>(), 2);
cout << fn(137) << endl; // Prints 274
cout << fn(42) << endl; // Prints 84
```

This is exactly what we're looking for – a class that can store any callable function that takes in an `int` and returns an `int`. If this doesn't strike you as a particularly elegant piece of code, take some time to look over it again.

There's one final step we should take, and that's to relax the restriction that `Function` always acts as a function from `ints` to `ints`. There's nothing special about `int`, and by giving `Function` clients the ability to specify their own parameter and return types we'll increase the scope of what `Function` is capable of handling. We'll thus templatize `Function` as `Function<ArgType, ReturnType>` and have it inherit from `unary_function<ArgType, ReturnType>` so that STL-minded clients can treat it as an adaptable function. We also need to make some minor edits to `IntFunction` (which we'll rename to `ArbitraryFunction` since `IntFunction` is no longer applicable), but in the interest of brevity we won't reprint them here.

The final interface for `Function` thus looks like this (a full listing is included at the end of this chapter):

```
template <typename ArgType, typename ReturnType>
class Function: public unary_function<ArgType, ReturnType>
{
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (const ArgType& value) const;

private:
    class ArbitraryFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ }

    ArbitraryFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

To conclude our discussion of `Window`, using the new `Function` type we could rewrite the `Window` class using `Function` as follows:

```
class Window
{
public:
    Window(const Function<int, int>& widthFn, /* ... */
           /* ... other member functions ... */

private:
    Function<int, int> widthFunction;
};
```

Now, clients can pass any unary function (or functor) that maps from `ints` to `ints` as a parameter to `Window` and the code will compile correctly.

External Polymorphism

The `Function` type we've just developed is subtle in its cleverness. Because we can convert any callable unary function into a `Function`, when writing code that needs to work with some sort of unary function, we can have that code use `Function` instead of any specific function type. This technique of abstracting away from the particular types that provide a behavior into an object representing that behavior is sometimes known as *external polymorphism*. As opposed to *internal polymorphism*, where we explicitly define a set of classes containing virtual functions, external polymorphism “grafts” a set of virtual functions onto any type that supports the requisite behavior.

As mentioned in the previous chapter, virtual functions can be slightly more expensive than regular functions because of the virtual function table lookup required. External polymorphism is implemented using inheritance and thus also incurs an overhead, but the overhead is slightly greater than regular inheritance. Think for a minute how the `Function` class we just implemented is designed. Calling `Function::operator()` requires the following:

1. Following the `ArbitraryFunction` pointer in the `Function` class to its virtual function table.
2. Calling the function indicated by the virtual function table, which corresponds to the particular `SpecificFunction` being pointed at.
3. Calling the actual function object stored inside the `SpecificFunction`.

This is slightly more complex than a regular virtual function call, and illustrates the cost associated with external polymorphism. That said, in some cases (such as the `Function` case outlined here) the cost is overwhelming offset by the flexibility afforded by external polymorphism.

More to Explore

The `Function` class we've just finished writing was inspired by the Boost library's `function` template, which acts like `Function` but works on functions of arbitrary arity and with a slightly clearer template syntax. For example, to create a Boost `function` object that holds a function taking three arguments and returning a `bool`, you could write

```
boost::function<bool (int, string, double)> myFunction;
```

The Boost library's `function` class is so overwhelmingly popular among C++ programmers that it has been adopted as part of the next generation of C++, nicknamed C++0x (see the upcoming chapter for more information).

The implementation of `boost::function` is downright awful and uses a more than its fair share of advanced preprocessor and template tricks to work correctly, but if you're interested in seeing how the techniques covered here can be applied to a real-world system you might want to check it out.

If you're interested in learning more about external polymorphism, consider looking into Adobe Software Technology Lab's `any_iterator` class, which encapsulates an arbitrary iterator. The documentation can be found on the Adobe STL's website at <http://stlab.adobe.com/>

Complete Function Listing

Here is the complete implementation of `Function`. Comments have been added to the relevant sections:

```
template <typename ArgType, typename ReturnType>
class Function: public unary_function<ArgType, ReturnType>
{
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (const ArgType& value) const;

private:
    /* Base class which represents some function that can be called with an ArgType
     * that returns a ReturnType. ArbitraryFunctions can also be deep-copied using
     * the clone() function.
     *
     * ArbitraryFunction is an abstract class since there is no good default
     * implementation for any of its member functions.
     */
    class ArbitraryFunction
    {
    public:
        /* Virtual destructor necessary for cleanup. */
        virtual ~ArbitraryFunction() {}

        /* execute calls the stored function and returns its return value. */
        virtual ReturnType execute(const ArgType& param) const = 0;

        /* clone returns a deep-copy of the receiver object. */
        virtual ArbitraryFunction* clone() const = 0;
    };
};
```

```

/* For any type of unary function, we define a subclass of ArbitraryFunction
 * which wraps that object so it can be called through the ArbitraryFunction
 * interface.
 */
template <typename UnaryFunction>
class SpecificFunction: public ArbitraryFunction
{
public:
    /* Constructor accepts and stores a UnaryFunction. */
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    /* execute just calls down to the function. */
    virtual ReturnT execute(const ArgType& param) const
    {
        return function(param);
    }

    /* Clone returns a deep-copy of this object. */
    virtual ArbitraryFunction* clone() const
    {
        return new ArbitraryFunction(*this);
    }
private:
    UnaryFunction function;
};

ArbitraryFunction* function;

void clear();
void copyOther(const Function& other);
};

/* Constructor accepts a UnaryFunction of the proper type, then wraps it inside a
 * SpecificFunction wrapper. Note that there are two template headers here since
 * the class and constructor are both templates.
 */
template <typename ArgType, typename ReturnT>
    template <typename UnaryFunction>
Function<ArgType, ReturnT>::Function(UnaryFunction fn)
{
    function = new SpecificFunction<UnaryFunction>(fn);
}

/* Destructor calls clear. */
template <typename ArgType, typename ReturnT>
Function<ArgType, ReturnT>::~~Function()
{
    clear();
}

/* Copy ctor calls copyOther. */
template <typename ArgType, typename ReturnT>
Function<ArgType, ReturnT>::Function(const Function& other)
{
    copyOther(other);
}

```

```
/* Standard assignment operator. */
template <typename ArgType, typename ReturnType>
Function<ArgType, ReturnType>&
    Function<ArgType, ReturnType>::operator=(const Function& other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}

/* clear deletes the stored pointer. */
template <typename ArgType, typename ReturnType>
void Function<ArgType, ReturnType>::clear()
{
    delete function;
}

/* copyOther uses the clone() member function to do the copy. Note that the copy
 * is necessary instead of using a shallow copy because the function might be a
 * functor with internal state.
 */
template <typename ArgType, typename ReturnType>
void Function<ArgType, ReturnType>::copyOther(const Function& other)
{
    function = other.function->clone();
}

/* Finally, operator() just calls down into the function and returns the result. */
template <typename ArgType, typename ReturnType>
ReturnType Function<ArgType, ReturnType>::operator()(const ArgType& param) const
{
    return function->execute(param);
}
```


Part Three

More to Explore

Whew! You've made it through the first three sections and are now a seasoned and competent C++ programmer. But your journey has just begun. There are many parts of the C++ programming language that we have not covered, and it's now up to you to begin the rest of your journey.

This last section of the book contains two chapters. The first, on C++0x, discusses what changes are expected for the C++ programming language over the next few years. Now that you've seen C++'s strengths and weaknesses, I hope that this chapter proves enlightening and exciting. The second chapter is all about how to continue your journey into further C++ mastery and hopefully can give you a boost in the right direction.

Chapter 31: C++0x

C++0x feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before. Rely on the old mantra: If you think [o]f it as a separate idea or object, represent it directly in the program; model real-world objects, concepts, and abstractions directly in code. It's easier now: Your ideas will map to enumerations, objects, classes (e.g. control of defaults), class hierarchies (e.g. inherited constructors), templates, concepts, concept maps, axioms, aliases, exceptions, loops, threads, etc., rather than to a single “one size fits all” abstraction mechanism.

My ideal is to use programming language facilities to help programmers think differently about system design and implementation. I think C++0x can do that – and do it not just for C++ programmers but for programmers used to a variety of modern programming languages in the general and very broad area of systems programming.

In other words, I'm still an optimist.

– Bjarne Stroustrup, inventor of C++. [Str09.3]

C++ is constantly evolving. Over the past few years the C++ standards body has been developing the next revision of C++, nicknamed C++0x. C++0x is a major upgrade to the C++ programming language and as we wrap up our tour of C++, I thought it appropriate to conclude by exploring what C++0x will have in store. This chapter covers some of the more impressive features of C++0x and what to expect in the future.

Be aware that C++0x has not yet been finalized, and the material in this chapter may not match the final C++0x specification. However, it should be a great launching point so that you know where to look to learn more about the next release of C++.

Automatic Type Inference

Consider the following piece of code:

```
void DoSomething(const multimap<string, vector<int> >& myMap)
{
    const pair<multimap<string, vector<int> >::const_iterator,
              multimap<string, vector<int> >::const_iterator> eq =
        myMap.equal_range("String!");
    for(multimap<string, vector<int> >::const_iterator itr = eq.first;
        itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

This above code takes in a `multimap` mapping from strings to `vector<int>`s and prints out the length of all vectors in the `multimap` whose key is “String!” While the code is perfectly legal C++, it is extremely difficult to follow because more than half of the code is spent listing the types of two variables, `eq` and `itr`. If you'll notice, these variables can only take on one type – the type of the expression used to initialize them. Since the compiler knows all of the types of the other variables in this code snippet, couldn't we just ask the compiler to give `eq` and `itr` the right types? Fortunately, in C++0x, the answer is yes thanks to a new language feature called *type inference*. Using type inference, we can rewrite the above function in about half as much space:

```
void DoSomething(const multimap<string, vector<int>>& myMap)
{
    const auto eq = myMap.equal_range("String!");
    for(auto itr = eq.first; itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

Notice that we've replaced all of the bulky types in this expression with the keyword `auto`, which tells the C++0x compiler that it should infer the proper type for a variable. The standard iterator loop is now considerably easier to write, since we can replace the clunky `multimap<string, vector<int> >::const_iterator` with the much simpler `auto`. Similarly, the hideous return type associated with `equal_range` is entirely absent.

Because `auto` must be able to infer the type of a variable from the expression that initializes it, you can only use `auto` when there is a clear type to assign to a variable. For example, the following is illegal:

```
auto x;
```

Since `x` could theoretically be of any type.

`auto` is also useful because it allows complex libraries to hide implementation details behind-the-scenes. For example, recall that the `ptr_fun` function from the STL `<functional>` library takes as a parameter a regular C++ function and returns an adaptable version of that function. In our discussion of the library's implementation, we saw that the return type of `ptr_fun` is either `pointer_to_unary_function<Arg, Ret>` or `pointer_to_binary_function<Arg1, Arg2, Ret>`, depending on whether the parameter is a unary or binary function. This means that if you want to use `ptr_fun` to create an adaptable function and want to store the result for later use, using current C++ you'd have to write something to the effect of

```
pointer_to_unary_function<int, bool> ouchies = ptr_fun(SomeFunction);
```

This is terribly hard to read but more importantly breaks the wall of abstraction of `ptr_fun`. The entire purpose of `ptr_fun` is to hide the transformation from function to functor, and as soon as you are required to know the return type of `ptr_fun` the benefits of the automatic wrapping facilities vanish. Fortunately, `auto` can help maintain the abstraction, since we can rewrite the above as

```
auto howNice = ptr_fun(SomeFunction);
```

C++0x will provide a companion operator to `auto` called `decltype` that returns the type of a given expression. For example, `decltype(1 + 2)` will evaluate to `int`, while `decltype(new char)` will be `char *`. `decltype` does not evaluate its argument – it simply yields its type – and thus incurs no cost at runtime.

One potential use of `decltype` arises when writing template functions. For example, suppose that we want to write a template function as follows:

```
template <typename T> /* some type */ MyFunction(const T& val)
{
    return val.doSomething();
}
```

This function accepts a `T` as a template argument, invokes that object's `doSomething` member function, then returns its value (note that if the type `T` doesn't have a member function `doSomething`, this results in a compile-time error). What should we use as the return type of this function? We can't tell by simply looking at the type `T`, since the `doSomething` member function could theoretically return any type. However, by using `decltype` and a new function declaration syntax, we can rewrite this as


```
template <typename T>
auto MyFunction(const T& val) -> decltype(val.doSomething())
{
    return val.doSomething();
}
```

Notice that we defined the function's return type as `auto`, and then after the parameter list said that the return type is `decltype(val.doSomething())`. This new syntax for function declarations is optional, but will make complicated function prototypes easier to read.

Move Semantics

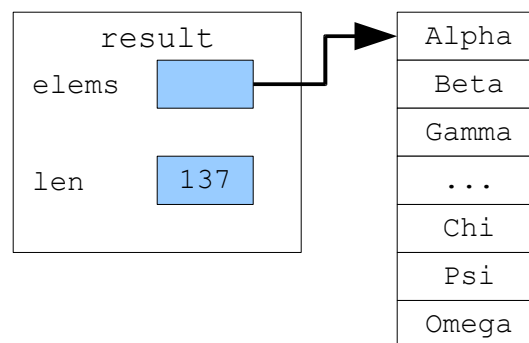
If you'll recall from our discussion of copy constructors and assignment operators, when returning a value from a function, C++ initializes the return value by invoking the class's copy constructor. While this method guarantees that the returned value is always valid, it can be grossly inefficient. For example, consider the following code:

```
vector<string> LoadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open())
        throw runtime_error("File not found!");

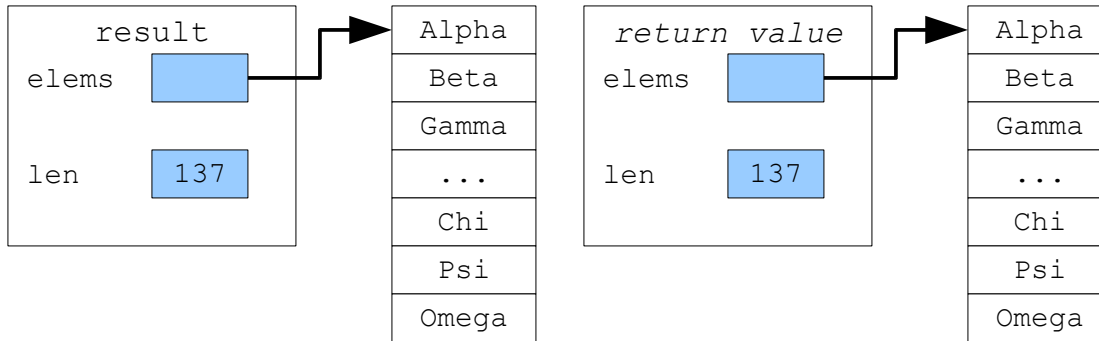
    /* Use the vector's insert function, plus some istream_iterators, to
     * load the contents of the file.
     */
    vector<string> result;
    result.insert(result.begin(), istream_iterator<string>(input),
                 istream_iterator<string>());

    return result;
}
```

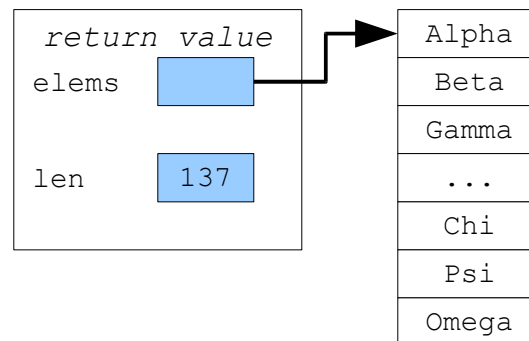
Here, we open the file specified by `filename`, then use a pair of `istream_iterators` to load the contents of the file into the `vector`. At the end of this function, before the `return result` statement executes, the memory associated with the `result` vector looks something like this (assuming a `vector` is implemented as a pointer to a raw C++ array):



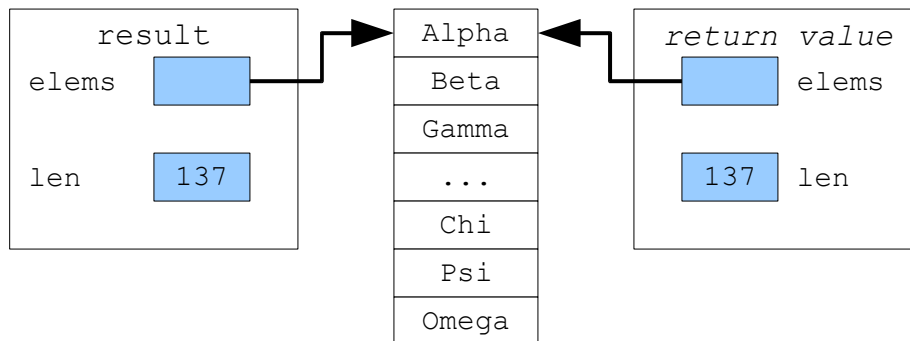
Now, the statement `return result` executes and C++ initializes the return value by invoking the `vector` copy constructor. After the copy the program's memory looks like this:



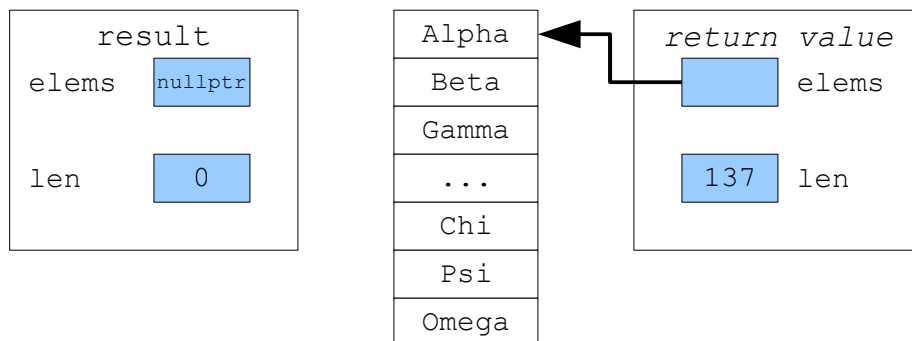
After the return value is initialized, `result` will go out of scope and its destructor will clean up its memory. Memory now looks like this:



Here, we made a full deep copy of the contents of the returned object, then deallocated all of the original memory. This is inefficient, since we needlessly copied a long list of strings. There is a much better way to return the `vector` from the function. Instead of initializing the return value by making a deep copy, instead we'll make it a shallow copy of `vector` we're returning. The in-memory representations of these two vectors thus look like this:



Although the two vectors share the same memory, the returned `vector` has the same contents as the source `vector` and is in fact indistinguishable from the original. If we then modify the original `vector` by detaching its pointer from the array and having it point to `NULL` (or, since this is C++0x, the special value `nullptr`), then we end up with a picture like this:



Now, `result` is an empty `vector` whose destructor will not clean up any memory, and the calling function will end up with a `vector` whose contents are exactly those returned by the function. We've successfully returned the value from the function, but avoided the expensive copy. In our case, if we have a `vector` of n strings of length at most m , then the algorithm for copying the `vector` will take $O(mn)$. The algorithm for simply transferring the pointer from the source `vector` to the destination, on the other hand, is $O(1)$ for the pointer manipulations.

The difference between the current method of returning a value and this improved version of returning a value is the difference between copy semantics and move semantics. An object has *copy semantics* if it can be duplicated in another location. An object has *move semantics* (a feature introduced in C++0x) if it can be moved from one variable into another, destructively modifying the original. The key difference between the two is the number of copies at any point. Copying an object duplicates its data, while moving an object transfers the contents from one object to another without making a copy.

To support move semantics, C++0x introduces a new variable type called an *rvalue reference* whose syntax is `Type &&`. For example, an rvalue reference to a `vector<int>` would be a `vector<int> &&`. Informally, you can view an rvalue reference as a reference to a temporary object, especially one whose contents are to be moved from one location to another.

Let's return to the above example with returning a `vector` from a function. In the current version of C++, we'd define a copy constructor and assignment operator for `vector` to allow us to return `vectors` from functions and to pass `vectors` as parameters. In C++0x, we can optionally define another special function, called a *move constructor*, that initializes a new `vector` by moving data out of one `vector` into another. In the above example, we might define a move constructor for the `vector` as follows:

```

/* Move constructor takes a vector&& as a parameter, since we want to move
 * data from the parameter into this vector.
 */
template <typename T> vector<T>::vector(vector&& other)
{
    /* We point to the same array as the other vector and have the same length. */
    elems = other.elems;
    len = other.len;

    /* Destructively modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}

```

Now, if we return a `vector` from a function, the new `vector` will be initialized using the move constructor rather than the regular copy constructor.

We can similarly define a *move assignment operator* (as opposed to the traditional *copy* assignment operator), as shown here:

```

template <typename T> vector<T>& vector<T>::operator= (vector&& other)
{
    if(this != &other)
    {
        delete [] elems;

        elems = other.elems;
        len = other.len;

        /* Modify the source vector to stop sharing the array. */
        other.elems = nullptr;
        other.len = 0;
    }
    return *this;
}

```

The similarity between a copy constructor and copy assignment operator is also noticeable here in the move constructor and move assignment operator. In fact, we can rewrite the pair using helper functions `clear` and `moveOther`:

```

template <typename T> void vector<T>::moveOther(vector&& other)
{
    /* We point to the same array as the other vector and have the same
     * length.
     */
    elems = other.elems;
    len = other.len;

    /* Modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}

template <typename T> void vector<T>::clear()
{
    delete [] elems;
    len = 0;
}

```

```

template <typename T> vector<T>::vector(vector&& other)
{
    moveOther(move(other)); // See later section for move
}

template <typename T> vector<T>& vector<T>::operator =(vector&& other)
{
    if(this != &other)
    {
        clear();
        moveOther(move(other));
    }
    return *this;
}

```

Move semantics are also useful in situations other than returning objects from functions. For example, suppose that we want to insert an element into an array, shuffling all of the other values down one spot to make room for the new value. Using current C++, the code for this operation is as follows:

```

template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = elems[i - 1]; // Shuffle elements down.
    elems[i] = toAdd;
}

```

There is nothing wrong *per se* with this code as it's written, but if you'll notice we're using copy semantics to shuffle the elements down when move semantics is more appropriate. After all, we don't want to *copy* the elements into the spot one element down; we want to *move* them.

In C++0x, we can use an object's move semantics (if any) by using the special helper function `move`, exported by `<utility>`, which simply returns an rvalue reference to an object. Now, if we write

```
a = move(b);
```

If `a` has support for move semantics, this will move the contents of `b` into `a`. If `a` does *not* have support for move semantics, however, C++ will simply fall back to the default object copy behavior using the assignment operator. In other words, supporting move operations is purely optional and a class can still use the old fashioned copy constructor and assignment operator pair for all of its copying needs.

Here's the rewritten version of `InsertIntoArray`, this time using move semantics:

```

template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.
    elems[i] = toAdd;
}

```

Curiously, we can potentially take this one step further by moving the new element into the array rather than copying it. We thus provide a similar function, which we'll call `MoveIntoArray`, which moves the parameter into the specified position:

```

template <typename T>
void MoveIntoArray(T* elems, int size, int position, T&& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.

    /* Note that even though toAdd is an rvalue reference, we still must
     * explicitly move it in. This prevents us from accidentally using
     * move semantics in a few edge cases.
     */
    elems[i] = move(toAdd);
}

```

Move semantics and copy semantics are independent and in C++0x it will be possible to construct objects that can be moved but not copied or vice-versa. Initially this might seem strange, but there are several cases where this is exactly the behavior we want. For example, it is illegal to copy an `ofstream` because the behavior associated with the copy is undefined – should we duplicate the file? If so, where? Or should we just share the file? However, it is perfectly legitimate to *move* an `ofstream` from one variable to another, since at any instant only one `ofstream` variable will actually hold a reference to the file stream. Thus functions like this one:

```

ofstream GetTemporaryOutputFile()
{
    /* Use the tmpnam() function from <cstdio> to get the name of a
     * temporary file. Consult a reference for more detail.
     */
    char tmpnamBuffer[L_tmpnam];
    ofstream result(tmpnam(tmpnamBuffer));
    return result; // Uses move constructor, not copy constructor!
}

```

Will be perfectly legal in C++0x because of move constructors, though the same code will not compile in current C++ because `ofstream` has no copy constructor.

Another example of an object that has well-defined move behavior but no copy behavior is the C++ `auto_ptr` class. If you'll recall, assigning one `auto_ptr` to another destructively modifies the original `auto_ptr`. This is exactly the definition of move semantics. However, under current C++ rules, implementing `auto_ptr` is extremely difficult and leads to all sorts of unexpected side effects. Using move constructors, however, we can eliminate these problems. C++0x will introduce a replacement to `auto_ptr` called `unique_ptr` which, like `auto_ptr`, represents a smart pointer that automatically cleans up its underlying resource when it goes out of scope. Unlike `auto_ptr`, however, `unique_ptr` cannot be copied or assigned but can be moved freely. Thus code of this sort:

```

unique_ptr<int> myPtr(new int);
unique_ptr<int> other = myPtr; // Error! Can't copy unique_ptr.

```

Will not compile. However, by explicitly indicating that the operation is a move, we can transfer the contents from one `unique_ptr` to another:

```

unique_ptr<int> myPtr(new int);
unique_ptr<int> other = move(myPtr); // Legal; myPtr is now empty

```

Move semantics and rvalue references may seem confusing at first, but promise to be a powerful and welcome addition to the C++ family.

Lambda Expressions

Several chapters ago we considered the problem of counting the number of `strings` in a `vector` whose lengths were less than some value determined at runtime. We explored how to solve this problem using the `count_if` algorithm and a functor. Our solution was as follows:

```
class ShorterThan
{
public:
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string& str) const
    {
        return str.length() < length;
    }
private:
    int length;
};

const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));
```

This functor-based approach works correctly, but has a huge amount of boilerplate code that obscures the actual mechanics of the solution. What we'd prefer instead is the ability to write code to this effect:

```
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), the string is shorter than myValue);
```

Using a new C++0x language feature known as *lambda expressions* (a term those of you familiar with languages like Scheme, ML, or Haskell might recognize), we can write code that very closely mirrors this structure. One possibility looks like this:

```
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(),
    [myValue](const string& x) { return x.length() < myValue; });
```

The construct in the final line of code is a *lambda expression*, an unnamed (“anonymous”) function that exists only as a parameter to `count_if`. In this example, we pass as the final parameter to `count_if` a temporary function that accepts a single `string` parameter and returns a `bool` indicating whether or not its length is less than `myValue`. The bracket syntax `[myValue]` before the parameter declaration (`int x`) is called the *capture list* and indicates to C++ that the lambda expression can access the value of `myValue` in its body.

Behind the scenes, C++ converts lambda expressions such as the one above into uniquely-named functors, so the above code is identical to the functor-based approach outlined above.

For those of you with experience in a functional programming language, the example outlined above should strike you as an extraordinarily powerful addition to the C++ programming language. Lambda expressions greatly simplify many tasks and represent an entirely different way of thinking about programming. It will be interesting to see how rapidly lambda expressions are adopted in professional code.

Variadic Templates

In the previous chapter we implemented a class called `Function` that wrapped an arbitrary unary function. Recall that the definition of `Function` is as follows:

```
template <typename ArgType, typename ReturnType> class Function
{
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (const ArgType& value) const;
private:
    /* ... */
};
```

What if we want to generalize `Function` to work with functions of arbitrary arity? That is, what if we want to create a class that encapsulates a binary, nullary, or ternary function? Using standard C++, we could do this by introducing new classes `BinaryFunction`, `NullaryFunction`, and `TernaryFunction` that were implemented similarly to `Function` but which accepted a different number of parameters. For example, here's one possible interface for `BinaryFunction`:

```
template <typename ArgType1, typename ArgType2, typename ReturnType>
class BinaryFunction
{
public:
    /* Constructor and destructor. */
    template <typename BinaryFn> BinaryFunction(BinaryFn);
    ~BinaryFunction();

    /* Copy support. */
    BinaryFunction(const BinaryFunction& other);
    BinaryFunction& operator= (const BinaryFunction& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (const ArgType& arg1, const ArgType& arg2) const;
private:
    /* ... */
};
```

Writing different class templates for functions of each arity is troublesome. As was the case with the `DimensionType` example, if we write `Function`-like classes for a fixed number of arities (say, functions between zero and ten arguments) and then discover that we need a wrapper for a function with more arguments, we'll have to write that class from scratch. Moreover, the structure of each function wrapper is almost identical. Compare the `BinaryFunction` and `Function` class interfaces mentioned above. If you'll notice, the only difference between the classes is the number of template arguments and the number of arguments to `operator()`. Is there some way that we can use this commonality to implement a single class that works with functions of arbitrary arity? Using the current incarnation of C++ this is not possible, but using a C++0x feature called *variadic templates* we can do just this.

A *variadic template* is a template that can accept an arbitrary number of template arguments. These arguments are grouped together into arguments called *parameter packs* that can be expanded out to code for each argument in the pack. For example, the following class is parameterized over an arbitrary number of arguments:

```
template <typename... Args> class Tuple
{
    /* ... */
};
```

The syntax `typename... Args` indicates that `Args` is a parameter pack that represents an arbitrary number of arguments. Since `Args` represents a list of arguments rather than an argument itself, it is illegal to use `Args` in an expression by itself. Instead, `Args` must be used in a *pattern expression* indicating what operation should be applied to each argument in the pack. For example, if we want to create a constructor for `Tuple` that accepts a list of arguments with one argument for each type in `Args`, we could write the following:

```
template <typename... Args> class Tuple
{
public:
    Tuple(const Args& ...);
};
```

Here, the syntax `const Args& ...` is a pattern expression indicating that for each argument in `Args`, there should be a parameter to the constructor that's passed by reference-to-const. For example, if we created a `Tuple<int>`, the constructor would be `Tuple<int>(const int&)`, and if we create a `Tuple<int, double>`, it would be `Tuple<int, double>(const int&, const double&)`.

Let's return to the example of `Function`. Suppose that we want to convert `Function` from encoding a unary function to encoding a function of arbitrary arity. Then we could change the class interface to look like this:

```
template <typename ReturnType, typename... ArgumentTypes> class Function
{
public:
    /* Constructor and destructor. */
    template <typename Callable> Function(Callable);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (const ArgumentTypes& ... args) const;
private:
    /* ... */
};
```

`Function` is now parameterized such that the first argument is the return type and the remaining arguments are argument types. For example, a `Function<int, string>` is a function that accepts a `string` and returns an `int`, while a `Function<bool, int, int>` would be a function accepting two `ints` and returning a `bool`.

We've just seen how the interface for `Function` looks with variadic templates, but what about the implementation? If you'll recall, the original implementation of `Function's operator()` function looked as follows:

```
template <typename ArgType, typename ReturnType>
ReturnType Function<ArgType, ReturnType>::operator() (const ArgType& param) const
{
    return function->execute(param);
}
```

Let's begin converting this to use variadic templates. The first step is to adjust the signature of the function, as shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator() (const ArgTypes&... args) const
{
    /* ... */
}
```

Notice that we've specified that this is a member of `Function<RetType, ArgTypes...>`.

In the unary version of `Function`, we implemented `operator()` by calling a stored function object's `execute` member function, passing in the parameter given to `operator()`. But how can we now call `execute` passing in an arbitrary number of parameters? The syntax for this again uses `...` to tell C++ to expand the `args` parameters to the function into an actual list of parameters. This is shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator() (const ArgTypes&... args) const
{
    return function->execute(args...);
}
```

Just as using `...` expands out a parameter pack into its individual parameters, using `args...` here expands out the variable-length argument list `args` into each of its individual parameters. This syntax might seem a bit tricky at first, but is easy to pick up with practice.

Another case where variadic templates is useful is with the `DimensionType` example from an earlier chapter. Recall that we defined a class called `DimensionType` parameterized over the kilograms, meters, and seconds of the stored quantity that let the compiler check unit correctness at compile-time. But what if we want to expand `DimensionType` to work with other SI units, such as electrical current, angle, or luminous intensity? One option would be to template the `DimensionType` type over more integer parameters, as shown here:

```
template <int kg, int m, int s, int rad, int amp, int cd> class DimensionType
{
public:
    explicit DimensionType(double amount) : quantity(amount) {}

    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};
```

However, what will happen now if we want to implement the mathematical operators `+`, `-`, `*`, and `/`? If you'll recall, those functions had very complex signatures. I've reprinted the implementation of `operator+` and `operator*` here:

```

template <int kg, int m, int s>
const DimensionType<kg, m, s> operator+ (const DimensionType<kg, m, s>& one,
                                         const DimensionType<kg, m, s>& two)
{
    return DimensionType<kg, m, s>(one.getQuantity() + two.getQuantity());
}

template <int kg1, int m1, int s1, int kg2, int m2, int s2>
    const DimensionType<kg1 + kg2, m1 + m2, s1 + s2>
operator* (const DimensionType<kg1, m1, s1>& one,
          const DimensionType<kg2, m2, s2>& two)
{
    return DimensionType<kg1 + kg2, m1 + m2, s1 + s2>
        (one.getQuantity() * two.getQuantity());
}

```

This signatures are not at all clean, and if we try to update them to work with even more types of units it will quickly become impossible to read or maintain.

But what if we opt for a different approach? Instead of parameterizing `DimensionType` over a fixed number of units, let's instead parameterize them over an *arbitrary number* of units. For example:

```

template <int... units> class DimensionType
{
public:
    explicit DimensionType(double amount) : quantity(amount) {}

    double getQuantity() const
    {
        return quantity;
    }
private:
    double quantity;
};

```

Here, the syntax `int... units` indicates that there can be multiple integer parameters to `DimensionType`. What would the implementation of `operator+` look like given this representation? Recall that it's only legal to add two dimensioned quantities if their units agree. Thus we'll have `operator+` accept two `DimensionTypes` of the same type, then perform the addition. This is shown here:

```

template <int... units>
const DimensionType<units...> operator+ (const DimensionType<units...>& one,
                                         const DimensionType<units...>& two)
{
    return DimensionType<units...>(one.getQuantity() + two.getQuantity());
}

```

Compare this with the original version from above:

```

template <int kg, int m, int s>
const DimensionType<kg, m, s> operator+ (const DimensionType<kg, m, s>& one,
                                         const DimensionType<kg, m, s>& two)
{
    return DimensionType<kg, m, s>(one.getQuantity() + two.getQuantity());
}

```

This new code is easier to read than the original, since the units are all bundled together in the single parameter pack `units` instead of three individual unit types.

Now, let's see how we might implement `operator*`. Recall that `operator*` accepts two parameters representing dimensioned types of arbitrary units, then returns a new dimensioned type whose quantity is the product of the input quantities and whose units are the sum of the original units. Using traditional templates, this is quite difficult – as you saw, we have to accept two parameters for each unit and have to explicitly indicate that the return type is formed by adding each individual unit. But using variadic templates and pattern expressions, this can be greatly simplified. For technical reasons, we'll define `operator*` as a template member function of `DimensionType`, as shown here:

```
template <int... units> class DimensionType
{
public:
    explicit DimensionType(double amount) : quantity(amount) {}

    double getQuantity() const
    {
        return quantity;
    }

    template <int... otherUnits>
    const DimensionType<units + otherUnits...>
        operator *(const DimensionType<otherUnits...>& other) const
    {
        return DimensionType<units + otherUnits...>(quantity * other.getQuantity());
    }
private:
    double quantity;
};
```

This code is still very dense, but is much, much clearer than the original version of the code. Notice that the return type of the function is a `const DimensionType<units + otherUnits...>`. Here, the pattern expression is `units + otherUnits...`, indicating that the units in the return type can be computed by adding pairwise the units in the input types.

One potential problem with this new approach is that it's legal to construct `DimensionTypes` parameterized over differing numbers of arguments; for example, a `DimensionType<1, 2, 3>` and a `DimensionType<0>`. Mixing and matching different numbers of units likely indicates a programmer error, so we'd like to enforce the fact that each `DimensionType` has the same number of arguments. Using a C++0x construct called a *static assertion*, it's possible to check this at compile-time. Like `assert`, `static_assert` accepts a predicate as a parameter and then checks whether the predicate is true, causing an error if not. Unlike `assert`, however, `static_assert` checks occur at *compile-time* rather than *runtime*. Using `static_assert` and the new `sizeof... operator` which returns the number of values in a parameter pack, we can enforce that `DimensionType` variables have a specific number of arguments as follows:

```

template <int... units> class DimensionType
{
public:

    static_assert(sizeof...(units) == kNumDimensions,
                  "Wrong number of arguments to DimensionType!");

    explicit DimensionType(double amount) : quantity(amount) {}

    double getQuantity() const
    {
        return quantity;
    }

    template <int... otherUnits>
    const DimensionType<units + otherUnits...>
        operator *(const DimensionType<otherUnits...>& other) const
    {
        return DimensionType<units + otherUnits...>(quantity * other.getQuantity());
    }
private:
    double quantity;
};

```

These two examples of variadic templates should demonstrate exactly how powerful C++0x language feature promises to be. It will be interesting to see what other ideas future C++ developers dream up.

Library Extensions

In addition to all of the language extensions mentioned in the above sections, C++0x will provide a new set of libraries that should make certain common tasks much easier to perform:

- **Enhanced Smart Pointers.** C++0x will support a wide variety of smart pointers, such as the reference-counted `shared_ptr` and the aforementioned `unique_ptr`.
- **New STL Containers.** The current STL associative containers (`map`, `set`, etc.) are layered on top of balanced binary trees, which means that traversing the `map` and `set` always produce elements in sorted order. However, the sorted nature of these containers means that insertion, lookup, and deletion are all $O(\lg n)$, where n is the size of the container. In C++0x, the STL will be enhanced with `unordered_map`, `unordered_set`, and multicontainer equivalents thereof. These containers are layered on top of hash tables, which have $O(1)$ lookup and are useful when ordering is not important.
- **Multithreading Support.** Virtually all major C++ programs these days contain some amount of multithreading and concurrency, but the C++ language itself provides no support for concurrent programming. The next incarnation of C++ will support a threading library, along with atomic operations, locks, and all of the bells and whistles needed to write robust multithreaded code.
- **Regular Expressions.** The combination of C++ `strings` and the STL algorithms encompasses a good deal of string processing functionality but falls short of the features provided by other languages like Java, Python, and (especially) Perl. C++0x will augment the `strings` library with full support for regular expressions, which should make string processing and compiler-authoring considerably easier in C++.
- **Upgraded <functional> library.** C++0x will expand on `<functional>` with a generic `function` type akin to the one described above, as well as a supercharged `bind` function that can bind arbitrary parameters in a function with arbitrary values.

- **Random Number Generation.** C++'s only random number generator is `rand`, which has extremely low randomness (on some implementations numbers toggle between even and odd) and is not particularly useful in statistics and machine learning applications. C++0x, however, will support a rich random number generator library, complete with a host of random number generators and probability distribution functors.
- **Metaprogramming Traits Classes.** C++0x will provide a large number of classes called *traits classes* that can help generic programmers write optimized code. Want to know if a template argument is an abstract class? Just check if `is_abstract<T>::type` evaluates to `true_type` or `false_type`.

Other Key Language Features

Here's a small sampling of the other upgrades you might find useful:

- **Unified Initialization Syntax:** It will be possible to initialize C++ classes by using the curly brace syntax (e.g. `vector<int> v = {1, 2, 3, 4, 5};`).
- **Delegating Constructors:** Currently, if several constructors all need to access the same code, they must call a shared member function to do the work. In C++0x, constructors can invoke each other in initializer lists.
- **Better Enumerations:** Currently, `enum` can only be used to create integral constants, and those constants can be freely compared against each other. In C++0x, you will be able to specify what type to use in an enumeration, and can disallow automatic conversions to `int`.
- **Angle Brackets:** It is currently illegal to terminate a nested template by writing two close brackets consecutively, since the compiler confuses it with the stream insertion operator `>>`. This will be fixed in C++0x.
- **C99 Compatibility:** C++0x will formally introduce the `long long` type, which many current C++ compilers support, along with various preprocessor enhancements.

C++0x Today

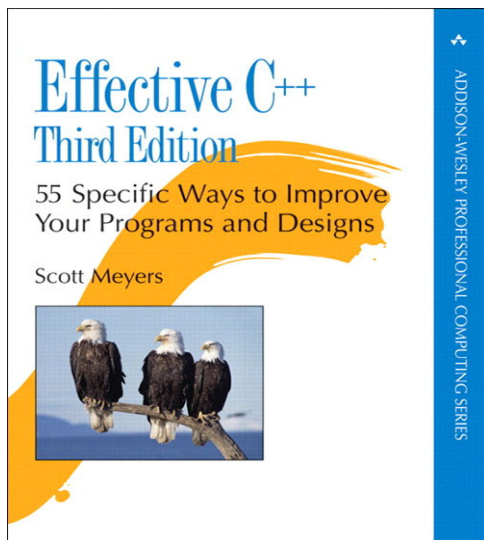
Although C++0x has not yet been adopted as a standard, there are several freely-available compilers that support a subset of C++0x features. For example, the Linux compiler `g++` version 4.4.0 has support for much of C++0x, and Microsoft Visual Studio 2010 (still only available in beta form) has a fair number of features implemented, including lambda expressions and the `auto` keyword. If you want to experience the future of C++ today, consider downloading one of these compilers.

Chapter 32: Where to Go From Here

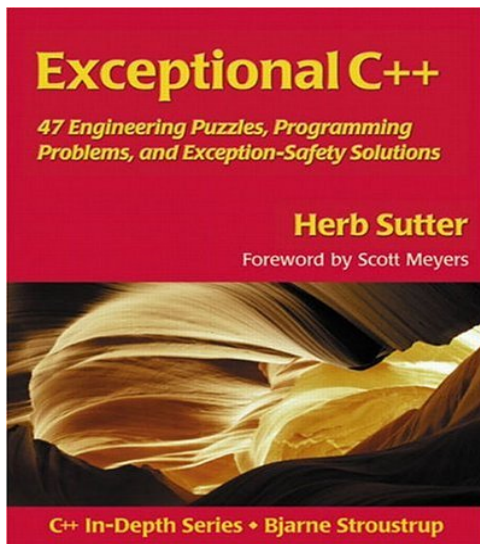
Congratulations! You've made it through CS106L. You've taken the first step on your journey toward a mastery of the C++ programming language. This is no easy feat! In the course of reading through this far, you now have a command of the following concepts:

- The streams library, including how to interface with it through operator overloading.
- STL containers, iterators, algorithms, adapters, and functional programming constructs, including a working knowledge of how these objects are put together.
- C strings, low-level pointer arithmetic, and how the operating system allocates memory to your program.
- The preprocessor and how to harness it to automatically generate C++ code.
- Generic programming in C++ and just how powerful the C++ template system can be.
- The `const` keyword and how to use it to communicate function side-effects to other programmers.
- Object layout and in-memory representation.
- Copy semantics and how to define implicit conversions between types.
- Operator overloading and how to make a C++ class act like a primitive type.
- What a functor is and how surprisingly useful and flexible they are.
- Exception handling and how to use objects to automatically manage resources.
- Inheritance, both at a high-level and at the nuanced level of C++.
- C++0x and what C++ will look like in the future.
- ... and a whole host of real-world examples of each of these techniques.

Despite all of the material we've covered here, there is *much* more to learn in the world of C++ and your journey has just begun. I feel that it is a fitting conclusion to this course reader to direct you toward other C++ resources that will prove invaluable along your journey into the wondrous realm of this language. In particular, there are several excellent C++ resources I would be remiss to omit:

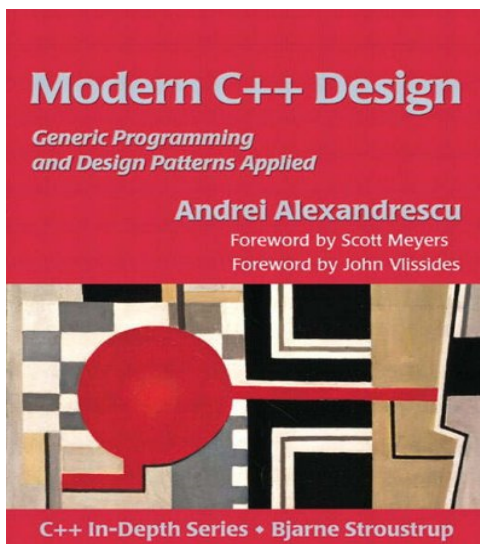
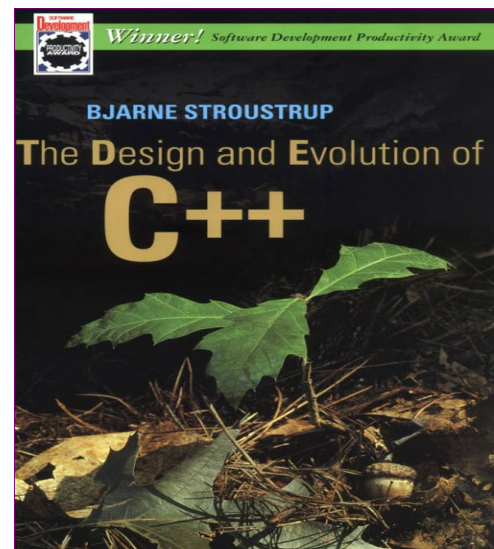


Effective C++, *More Effective C++*, and *Effective STL* by Scott Meyers. Picking up and reading this trio of books is perhaps the best thing you can do for yourself as a C++ programmer. The books in the *Effective C++* series will help you transition from a solid C++ programmer into an excellent C++ programmer and are widely regarded as among the best C++ books on the market. What separates the *Effective C++* series from most other C++ books is that *Effective C++* focuses almost exclusively on correct usage of core C++ language features and how to avoid common pitfalls. If you plan on using C++ in the professional world, you should own copies of this book.



Exceptional C++ by Herb Sutter. This book is an invaluable tool in any C++ programmer's arsenal. The book is largely organized as a set of puzzles that give you a chance to think about the best way to solve a problem and what C++ issues you'll encounter in the process. Along with *Effective C++*, *Exceptional C++* is one of the most highly-recommended C++ books out there. Herb Sutter's personal website is also an excellent resource for all your C++ needs.

The Design and Evolution of C++ by Bjarne Stroustrup. This book, affectionately known to hardcore C++ programmers as *D&E*, is a glimpse into Bjarne Stroustrup's thought processes as he went about designing C++. D&E is not a programming guide, but rather a history of the evolution of C++ from the small language C with Classes into the modern language we know and love today. D&E was written before C++ had been ISO standardized and even predates the STL, meaning that it can offer a new perspective on some of the language features and libraries you may take for granted. If you want an interesting glimpse into the mind of the man behind C++, this is the book for you.



Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu. Considered the seminal work in modern C++ programming, this book is an excellent introduction into an entirely new way of thinking in C++. Alexandrescu takes many advanced language features like templates and multiple inheritance, then shows how to harness them to achieve synergistic effects that are far more powerful than any of the individual features used. As an example, the first chapter shows how to write a single smart pointer class that is capable of storing any type of value, performing any sort of resource management, and having any copy behavior that the client desires. The book is very language-intensive and requires you to have a grasp of C++ slightly beyond the scope of this reader, but is a most wonderful text for all who are interested.

Final Thoughts

It's been quite a trip since we first started with the stream library. You now know how to program with the STL, write well-behaved C++ objects, and even how to use functional programming constructs. But despite the immense amount of material we've covered, we have barely scratched the surface of C++. There are volumes of articles and books out there that cover all sorts of amazing C++ tips and tricks, and by taking the initiative and exploring what's out there you can hone your C++ skills until problem solving in C++ transforms from “how do I solve this problem?” to “which of these many options is best for solving this problem?”

C++ is an amazing language. It has some of the most expressive syntax of any modern programming language, and affords an enormous latitude in programming styles. Of course, it has its flaws, as critics are eager to point out, but despite the advent of more modern languages like Java and Python C++ still occupies a prime position in the software world.

I hope that you've enjoyed reading this course reader as much as I enjoyed writing it. If you have any comments, suggestions, or criticisms, feel free to email me at htiek@cs.stanford.edu. Like the C++ language, CS106L and this course reader are constantly evolving, and if there's anything I can do to make the class more enjoyable, be sure to let me know!

Have fun with C++, and I wish you the best of luck wherever it takes you!

Appendices

Appendix 0: Moving from C to C++

C++ owes a great debt to the C programming language. Had it not been rooted in C syntax, C++ would have attracted fewer earlier adopters and almost certainly would have vanished into the mists of history. Had it not kept C's emphasis on runtime efficiency, C++ would have lost relevance over time and would have gone extinct. But despite C++'s history in C, C and C++ are very different languages with their own idioms and patterns. It is a common mistake to think that knowledge of C entails a knowledge of C++ or vice-versa, and experience with one language often leads to suboptimal coding skills in the other. In particular, programmers with a background in pure C often use C constructs in C++ code where there is a safer or more elegant alternative. This is not to say that C programmers are somehow worse coders than C++ programmers, but rather that some patterns engrained into the C mentality are often incompatible with the language design goals of C++.

This appendix lists ten idiomatic C patterns that are either deprecated or unsafe in C++ and suggests replacements. There is no new C++ content here that isn't already covered in the main body of the course reader, but I highly recommend reading through it anyway if you have significant background in C. This by no means an exhaustive list of differences between C and C++, but should nonetheless help you transition between the languages.

Tip 0: Prefer streams to `stdio.h`

C++ contains the C runtime library in its standard library, so all of the I/O functions you've seen in `<stdio.h>` (`printf`, `scanf`, `fopen`) are available in C++ through the `<cstdio>` header file. While you're free to use `printf` and `scanf` for input and output in C++, I strongly advise you against doing so because the functions are inherently unsafe. For example, consider the following C code:

```
char myString[1024] = {'\0'};
int myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Here, we prompt the user for an integer and a string, then print them back out if the user entered them correctly. As written there is nothing wrong with this code. However, consider the portions of the code I've highlighted here:

```
char myString[1024] = {'\0'};
int myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Consider the size of the buffer, 1024. When reading input from the user, if we don't explicitly specify that we want to read at most 1023 characters of input, we risk a buffer overrun that can trash the stack and allow an attacker to fully compromise the system. What's worse, if there is a mismatch between the declared size of the buffer (1024) and the number of characters specified for reading (1023), the compiler will not provide any warnings. In fact, the only way we would discover the problem is if we were very careful to read over the code checking for this sort of mistake, or to run an advanced tool to double-check the code for consistency.

Similarly, consider the highlighted bits here:

```
char myString[1024] = {'\0'};
int myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Notice that when reading values from the user or writing values to the console, we have to explicitly mention what types of variables we are reading and writing. The fact that `myInt` is an `int` and `myString` is a `char*` is insufficient for `printf` and `scanf`; we have to mention to read in an `int` with `%d` and a string with `%s`. If we get these backwards or omit one, the program contains a bug but will compile with no errors.* Another vexing point along these lines is the parameter list in `scanf` – we must pass in a pointer to `myInt`, but can just specify `myString` by itself. Confusing these or getting them backwards will cause a crash or a compiler warning, which is quite a price to pay for use input.

The problem with the C I/O libraries is that they completely bypass the type system. Recall that the signatures of `printf` and `scanf` are

```
int printf(const char* formatting, ...);
int scanf (const char* formatting, ...);
```

The `...` here means that the caller can pass in any number of arguments of any type, and in particular this means that the C/C++ compiler cannot do any type analysis to confirm that you're using the arguments correctly. Don't get the impression that C or C++ are type-safe – they're not – but the static type systems they have are designed to prevent runtime errors from occurring and subverting this system opens the door for particularly nasty errors.

In pure C, code like the above is the norm. In C++, however, we can write the following code instead:

```
int myInt;
string myString;

cout << "Enter an int and a string: ";
cin >> myInt >> myString;
cout << "You entered " << myInt << " and " << myString << endl;
```

If you'll notice, the only time that the types of `myInt` and `myString` are mentioned is at the point of declaration. When reading and writing `myInt` and `myString`, the C++ can automatically infer which version of `operator >>` and `operator <<` to call to perform I/O and thus there is no chance that we can accidentally read a string value into an `int` or vice-versa. Moreover, since we're using a C++-style string, there is no chance that we'll encounter a buffer overflow. In short, the C++ streams library is just plain safer than the routines in `<stdio.h>`.

When working in pure C++, be wary of the `<stdio.h>` functions. You are missing out on the chance to use the streams library and are exposing yourself and your code to all sorts of potential security vulnerabilities.

Tip 1: Use C++ **strings** instead of C-style strings

Life is short, nasty, and brutish, and with C strings it will be even worse. C strings are notoriously tricky to get right, have a cryptic API, and are the cause of all sorts of security bugs. C++ **strings**, on the other hand, are elegant, pretty, and difficult to use incorrectly. If you try truncating a C++ **string** at an invalid index with `erase`, the **string** will throw an exception rather than clobbering memory. If you append data to a C++ **string**, you don't need to worry about reallocating any memory – the object does that for you. In short, C

* Many compilers will report errors if you make this sort of mistake, but they are not required to do so.

strings are tricky to get *right*, and C++ strings are tricky to get *wrong*. “But wait!,” you might exclaim, “Because C strings are so low-level, I can sometimes outperform the heavyweight C++ `string`.” This is absolutely true – because C strings are so exposed, you have a great deal of flexibility and control over how the memory is managed and what operations go on behind the scenes. But is it really worth it? Here's a small sampling of what can go wrong if you're not careful with C strings:

1. You might write off the end of a buffer, clobbering other data in memory and paving the way for a massive security breach.
2. You might forget to deallocate the memory, causing a memory leak.
3. You might overwrite the terminating null character, leading to a runtime error or incomprehensible program outputs.

Are C strings faster than their C++ counterparts? Of course. But should you nonetheless sacrifice a little speed for the peace of mind that your program isn't going to let hackers take down your system? Absolutely.

Tip 2: Use C++ typecasts instead of C typecasts

Both C and C++ have static type systems – that is, if you try to use a variable of one type where a variable of another type is expected, the compiler will report an error. Both C and C++ let you use typecasts to convert between types when needed, sometimes safely and sometimes unsafely.

C has only one style of typecast, which is conveniently dubbed a “C-style typecast.” As mentioned in the chapter on inheritance, C-style typecasts are powerful almost to a fault. Converting between a `double` and an `int` uses the same syntax for unsafe operations like converting pointers to integers, integers to pointers, `const` variables to non-`const` variables, and pointers of one type to pointers of another type. As a result, it is easy to accidentally perform a typecast other than the one you wanted. For example, suppose we want to convert a `char*` pointer to an `int*` pointer, perhaps because we're manually walking over a block of memory. We write the following code:

```
const char* myPtr = /* ... */
int* myIntPtr = (int *)myPtr;
```

Notice that in this typecast we've converted a `const char*` to an `int*`, subverting `const`ness. Is this deliberate? Is this a mistake? Given the above code there's no way to know because the typecast does not communicate what sort of cast is intended. Did we mean to strip off `const`ness, convert from a `char*` to an `int*`, or both?

C++ provides three casting operators (`const_cast`, `static_cast`, `reinterpret_cast`) that are designed to clarify the sorts of typecasts performed in your code. Each performs exactly one function and causes a compile-time error if used incorrectly. For example, if in the above code we only meant to convert from a `const char*` to a `const int*` without stripping `const`ness, we could write it like this:

```
const char* myPtr = /* ... */
const int* myIntPtr = reinterpret_cast<const int*>(myPtr);
```

Now, if we leave off the `const` in the typecast, we'll get a compile-time error because `reinterpret_cast` can't strip off `const`ness. If, on the other hand, we want to convert the pointer from a `const char*` to a regular `int*`, we could write it as follows:

```
const char* myPtr = /* ... */
int* myIntPtr = const_cast<int*>(reinterpret_cast<const int*>(myPtr));
```

This is admittedly much longer and bulkier than the original C version, but it is also more explicit about exactly what it's doing. It also is safer, since the compiler can check that the casts are being used correctly.

When writing C++ code that uses typecasts, make sure that you use the C++-style casting operators. Are they lengthy and verbose? Absolutely. But the safety and clarity guarantees they provide will more than make up for it.

Tip 3: Prefer `new` and `delete` to `malloc` and `free`

In C++, you can allocate and deallocate memory either using `new` and `delete` or using `malloc` and `free`. If you're used to C programming, you may be tempted to use `malloc` and `free` as you have in the past. This can lead to very subtle errors because `new` and `delete` do *not* act the same as `malloc` and `free`. For example, consider the following code:

```
string* one = new string;
string* two = static_cast<string*>(malloc(sizeof string));
```

Here, we create two `string` objects on the heap – one using `new` and one using `malloc`. Unfortunately, the `string` allocated with `malloc` is a ticking timebomb waiting to explode. Why is this? The answer has to do with a subtle but critical difference between the two allocation routines.

When you write `new string`, C++ performs two steps. First, it conjures up memory from the heap so that the new `string` object has a place to go. Second, it calls the `string` constructor on the new memory location to initialize the `string` data members. On the other hand, if you write `malloc(sizeof string)`, you only perform the memory allocation. In the above example, this means that the `string` object pointed at by `two` has the right size for a `string` object, but isn't actually a `string` because none of its data members have been set appropriately. If you then try using the `string` pointed at by `two`, you'll get a nasty crash since the object is in a garbage state. To avoid problems like this, make sure that you always allocate objects using `new` rather than `malloc`.

If you do end up using both `new` and `malloc` in a C++ program (perhaps because you're working with legacy code), make sure that you are careful to deallocate memory with the appropriate deallocator function. That is, don't `free` an object allocated with `new`, and don't `delete` an object allocated with `malloc`. `malloc` and `new` are not the same thing, and memory allocated with one is not necessarily safe to clean up with the other. In fact, doing so leads to undefined behavior, which can really ruin your day.

Tip 4: Avoid `void*` Pointers

Code in pure C abounds with `void*` pointers, particularly in situations where a function needs to work with data of any type. For example, the C library function `qsort` is prototyped as

```
void qsort(void* elems, size_t numElems, size_t elemSize,
          int (*cmpFn)(const void*, const void*));
```

That's quite a mouthful and uses `void*` three times – once for the input array and twice in the comparison function. The reason for the `void*` here is that C lacks language-level support for generic programming and consequently algorithms that need to operate on arbitrary data have to cater to the lowest common denominator – raw bits and bytes.

When using C's `qsort`, you have to be extremely careful to pass in all of the arguments correctly. When sorting an array of `ints`, you must take care to specify that `elemSize` is `sizeof(int)` and that your comparison function knows to interpret its arguments as pointers to `ints`. Passing in a comparison function which tries to treat its arguments as being of some other type (perhaps `char**s` or `double*s`) will cause runtime errors, and specifying the size of the elements in the array incorrectly will probably cause incorrect behavior or a bus error.

Contrast this with C++'s `sort` algorithm:


```
template <typename RandomAccessIterator, typename Comparator>
void sort(RandomAccessIterator begin, RandomAccessIterator end, Comparator c);
```

With C++'s `sort`, the compiler can determine what types of elements are stored in the range `[begin, end)` by looking at the type of the iterator passed as a parameter. The compiler can thus automatically figure out the size of the elements in the range. Moreover, if there is a type mismatch between what values the `Comparator` parameter accepts and what values are actually stored in the range, you'll get a compile-time error directing you to the particular template instantiation instead of a difficult-to-diagnose runtime error.

This example highlights the key weakness of `void*` – it completely subverts the C/C++ type system. When using a `void*` pointer, you are telling the compiler to forget all type information about what's being pointed at and therefore have to explicitly keep track of all of the relevant type information yourself. If you make a mistake, the compiler won't recognize your error and you'll have to diagnose the problem at runtime. Contrast this with C++'s template system. C++ templates are strongly-typed and the compiler will ensure that everything type-checks. If the program has a type error, it won't compile, and you can diagnose and fix the problem without having to run the program.

Whenever you're thinking about using a `void*` in C++ programming, make sure that it's really what you want to do. There's almost always a way to replace the `void*` with a template. Then again, if you want to directly manipulate raw bits and bytes, `void*` is still your best option.

One point worth noting: In pure C, you can implicitly convert between a `void*` pointer and a pointer of any type. In C++, you can implicitly convert any pointer into a `void*`, but you'll have to use an explicit typecast to convert the other way. For example, the C code

```
int* myArray = malloc(numElems * sizeof(int));
```

Does not compile in C++ since `malloc` returns a `void*`. Instead, you'll need to write

```
int* myArray = (int *)malloc(numElems * sizeof(int));
```

Or, even better, as

```
int* myArray = static_cast<int *>(malloc(numElems * sizeof(int)));
```

Using the C++ `static_cast` operator.

Tip 5: Prefer `vector` to raw arrays

Arrays live in a sort of nether universe. They aren't quite variables, since you can't assign them to one another, and they're not quite pointers, since you can't reassign where they're pointing. Arrays can't remember how big they are, but when given a static array you can use `sizeof` to get the total space it occupies. Functions that operate on arrays have to either guess the correct size or rely on the caller to supply it. In short, arrays are a bit of a mess in C and C++.

Contrast this with the C++ `vector`. `vectors` know exactly how large they are, and can tell you if you ask. They are first-class variables that you can assign to one another, and aren't implicitly convertible to pointers. On top of that, they clean up their own messes, so you don't need to worry about doing your own memory management. In short, the `vector` is everything that the array isn't.

In addition to being safer than regular arrays, `vectors` can also be much cleaner and easier to read. For example, consider the following C code:

```
void MyFunction(int size)
{
    int* arr = malloc(size * sizeof(int));
    memset(arr, 0, size * sizeof(int));

    /* ... */

    free(arr);
};
```

Compare this to the equivalent C++ code:

```
void MyFunction(int size)
{
    vector<int> vec(size);

    /* ... */
};
```

No ugly computations about the size of each element, no messy cleanup code at the end, and no `memset`s. Just a single line and you get the same effect. Moreover, since the `vector` always cleans up its memory after it goes out of scope, the compiler will ensure that no memory is leaked. Isn't that nicer?

Of course, there are times that you might want to use a fixed-size array, such as if you know the size in advance and can fit the array into a struct. But in general, when given the choice between using arrays and using vectors, the `vector` is the more natural choice in C++.

Tip 6: Avoid `goto`

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code)

– Edsger Dijkstra [Dij68]

The `goto` keyword been widely criticised since Dijkstra published “Go To Statement Considered Harmful” in 1968, yet still managed to make its way into C and consequently C++. Despite its apparent simplicity, `goto` can cause all sorts of programming nightmares because it is inherently *unstructured*. `goto` can jump pretty much anywhere and consequently can lead to unintuitive or even counterintuitive code. For example, here's some code using `goto`:

```
int x = 0;
start:
    if (x == 10) goto out;
    printf("%d\n", x);
    ++x;
    goto start;
out:
    printf("Done!\n");
```

This is completely equivalent to the *much* more readable

```
for(int x = 0; x < 10; ++x)
    printf("%d\n", x);
```

Despite `goto`'s bad reputation, modern C programming still has several places in which `goto` can still be useful. First, `goto` can be used as a sort of “super break” to break out of multiple levels of loop nesting. This use is still legitimate in C++, but is frowned upon stylistically. Second, `goto` can be used as a way of performing necessary cleanup in an error condition. For example:

```
/* Returns a string of the first numChars characters from a file or NULL in an
 * error case.
 */
char* ReadFromFile(const char* filename, size_t numChars)
{
    FILE* f;
    char* buffer;

    /* Allocate some space. */
    buffer = malloc(numChars + 1);
    if(buffer == NULL) return NULL;

    /* Open the file, abort on error. */
    f = fopen(filename, "rb");
    if(f == NULL)
        goto error;

    /* Read the first numChars characters, failing if we don't read enough. */
    if(fread(buffer, numChars, 1, f) != numChars)
        goto error;

    /* Close the file, null-terminate the string, and return. */
    fclose(f);
    buffer[numChars] = '\0';
    return buffer;

    /* On error, clean up the resources we opened. */
error:
    free(buffer);
    if(f != NULL)
        fclose(f);

    return NULL;
}
```

Here, there are several error conditions in which we need to clean up the temporary buffer and potentially close an open file. When this happens, rather than duplicating the cleanup code, we use `goto` to jump to the error-handling subroutine.

In pure C this is perfectly fine, but in C++ would be considered a gross error because there are much better alternatives. As mentioned in the chapter on exception handling, we could instead use a catch-and-rethrow strategy to get the exact same effect without `goto`, as shown here:

```

/* Returns a string of the first numChars characters from a file.
 * Throws a runtime_error on error.
 */
char* ReadFromFile(const char* filename, size_t numChars)
{
    FILE* f;
    char* buffer = NULL;

    try
    {
        /* Allocate some space. This will throw on error rather than returning
         * NULL.
         */
        buffer = new char[numChars + 1];

        /* Open the file, abort on error. */
        f = fopen(filename, "rb");
        if(f == NULL)
            throw runtime_error("Can't open file!");

        /* Read the first numChars characters, failing if we don't read enough. */
        if(fread(buffer, numChars, 1, f) != numChars)
            throw runtime_error("Can't read enough characters!");

        /* Close the file, null-terminate the string, and return. */
        fclose(f);
        buffer[numChars] = '\0';
        return buffer;
    }
    catch(...)
    {
        /* On error, clean up the resources we opened. */
        delete [] buffer;
        if(f != NULL)
            fclose(f);

        throw;
    }
}

```

Now that we're using exception-handling instead of `goto`, the code is easier to read and allows the caller to get additional error information out of the function.

An even better alternative here would be to use an `ifstream` and a `string` to accomplish the same result. Since the `ifstream` and `string` classes have their own destructors, we don't need to explicitly clean up any memory. This is shown here:

```

/* Returns a string of the first numChars characters from a file.
 * Throws a runtime_error on error.
 */
string ReadFromFile(const char* filename, size_t numChars)
{
    string buffer(numChars);

    /* Open the file, abort on error. */
    ifstream input(filename);
    if(input.fail())
        throw runtime_error("Can't open the file!");

    /* Read the first numChars characters, failing if we don't read enough. */
    input.read(&buffer[0], numChars);
    if(input.fail())
        throw runtime_error("Couldn't read enough data");

    return buffer;
}

```

This version is very clean and concise and doesn't require any `goto`-like structure at all. Since the object destructors take care of all of the cleanup, we don't need to worry about doing that ourselves.

My advice against `goto` also applies to `setjmp` and `longjmp`. These functions are best replaced with C++'s exception-handling system, which is far safer and easier to use.

Tip 7: Use C++'s `bool` type when applicable

Prior to C99, the C programming language lacked a standard `bool` type and it was common to find idioms such as

```
enum bool {true, false};
```

Or

```
#define bool int
#define true 1
#define false 0
```

Similarly, to loop indefinitely, it was common to write

```
while(1)
{
    /* ... */
}
```

Or

```
for(;;)
{
    /* ... */
}
```

Defining your own custom `bool` type is risky in C++ because a custom type will not interact with language features like templates and overloading correctly. Similarly, while both of the “loop forever” loop constructs listed above are legal C++ code, they are both less readable than the simpler

```
while (true)
{
    /* ... */
}
```

If you aren't already used to working with `bool`s, I suggest that you begin doing so when working in C++. Sure, you can emulate the functionality of a `bool` using an `int`, but doing so obscures your intentions and leads to all sorts of other messes. For example, if you try to emulate `bool`s using `ints`, you can get into nasty scrapes where two `ints` each representing `true` don't compare equal because they hold different nonzero values. This isn't possible with the `bool` type. To avoid subtle sources of error and to make your code more readable, try to use `bool` whenever applicable.

Tip 8: Avoid “`typedef struct`”

In pure C, if you define a struct like this:

```
struct pointT
{
    int x, y;
};
```

Then to create an instance of the struct you would declare a variable as follows:

```
struct pointT myPoint;
```

In C++, this use of `struct` is unnecessary. It is also bad practice, since veteran C++ programmers will almost certainly have to pause and think about exactly what the code means. Of course, most C programmers are also not particularly fond of this syntax, and to avoid having to spell out `struct` each time would write

```
typedef struct pointT_
{
    int x, y;
} pointT;
```

This syntax is still valid C++, but is entirely unnecessary and makes the code significantly trickier to read. Moreover, if you want to add constructors or destructors to the `struct` you would have to use the name `pointT_` even though externally the object would be called `pointT` without the underscore. This makes the code more difficult to read and may confuse class clients. In the interests of clarity, avoid this use of `typedef` when writing C++ code.

Tip 9: Avoid `memcpy` and `memset`

In pure C, code like the following is perfectly normal:

```
struct pointT
{
    int x, y;
};
struct pointT myPoint;
memset(&myPoint, 0, sizeof(pointT));
```

Here, the call to `memset` is used to initialize all the variables in the `pointT` to zero. Since C lacks constructors and destructors, this code is a reasonably good way to ensure that the `pointT` is initialized before use.

Because C++ absorbed C's standard library, the functions `memset`, `memcpy`, and the like are all available in C++. However, using these functions can lead to subtle but dangerous errors that can cause all sorts of runtime woes. For example, consider the following code:

```
string one = "This is a string!";
string two = "I like this string more.";
memcpy(&one, &two, sizeof(string)); // Set one equal to two - does this work?
```

Here, we use `memcpy` to set `one` equal to `two`. Initially, it might seem like this code works, but unfortunately this `memcpy` results in undefined behavior and will almost certainly cause a runtime crash. The reason is that the `string` object contains pointers to dynamically-allocated memory, and when `memcpy`ing the data from `two` into `one` we've made both of the string objects point to the same memory. After each pointer goes out of scope, both will try to reclaim the memory, causing problems when the underlying string buffer is doubly-deleted. Moreover, if this *doesn't* immediately crash the program, we've also leaked the memory `one` was originally using since all of its data members were overwritten without first being cleaned up.

If we wanted to set `one` equal to `two`, we could have just written this instead:

```
string one = "This is a string!";
string two = "I like this string more.";
two = one;
```

This uses the `string`'s assignment operator, which is designed to safely perform a deep copy.

In general, mixing `memcpy` with C++ classes is just asking for trouble. Most classes maintain some complex invariants about their data members and what memory they reference, and if you `memcpy` a block of data over those data members you might destroy those invariants. `memcpy` doesn't respect `public` and `private`, and thus completely subverts the encapsulation safeguards C++ tries to enforce.

But the problem runs deeper than this. Suppose that we have a polymorphic class representing a binary tree node:

```
class BinaryTreeNode
{
public:
    BinaryTreeNode();
    virtual ~BinaryTreeNode(); // Polymorphic classes need virtual destructors

    /* ... etc. ... */
private:
    BinaryTreeNode* left, *right;
};
```

We want to implement the constructor so that it sets `left` and `right` to `NULL`, indicating that the node has no children. Initially, you might think that the following code is safe:

```
BinaryTreeNode::BinaryTreeNode()
{
    /* Zero out this object. Is this safe? */
    memset(this, 0, sizeof(BinaryTreeNode));
}
```

Since the null pointer has value zero, it seems like this should work – after all, if we overwrite the entire object with zeros, we've effectively nulled out all of its pointer data members. But this code is a recipe for disaster because the class contains more than just a `left` and `right` pointer. In the chapter on inheritance, we outlined

how virtual functions are implemented using a virtual function table pointer that sits at the beginning of the class. If we use `memset` to clear out the object, we'll overwrite this virtual function table pointer with `NULL`, meaning that any attempt to call a virtual function on this object will result in a null pointer dereference and a program crash.*

The key problem with `memset` and `memcpy` is that they completely subvert the abstractions C++ provides to increase program safety. Encapsulation is supposed to prevent clients from clobbering critical class components and object layout is done automatically specifically to prevent programmers from having to explicitly manipulate low-level machinery. `memset` and `memcpy` remove these barriers and expose you to dangerous you could otherwise do without.

This is not to say, of course, that `memset` and `memcpy` have no place in C++ code – they certainly do – but their role is considerably less prominent than in pure C. Before you use low-level manipulation routines, make sure that there isn't a better way to accomplish the same goal through more “legitimate” C++ channels.

With that said, welcome to C++! Enjoy your stay!

* This example is based on a conversation I had with Edward Luong, who encountered this very problem when writing a large program in C++.

Appendix 1: Solutions to Practice Problems

Streams

Problem 2. There are two steps necessary to get `HasHexLetters` working. First, we transform the input number into a string representation of its hexadecimal value. Next, using techniques similar to that for `GetInteger`, we check to see if this string can be interpreted as an `int` when read in base 10. If so, the hexadecimal representation of the number must not contain any letters (since letters can't be interpreted as a decimal value), otherwise the representation has at least one letter in it.

One possible implementation is given here:

```
bool HasHexLetters(int value)
{
    /* Funnel the data into a stringstream, using the hex manipulator to represent
     * it in hexadecimal.
     */
    stringstream converter;
    converter << hex << value;

    /* Now, try extracting the string as an int, using the dec manipulator to read
     * it in decimal.
     */
    int dummy;
    converter >> dec >> dummy;

    /* If the stream failed, we couldn't read an int and we're done. */
    if(converter.fail()) return true;

    /* Otherwise, try reading something else from the stream. If we succeed, it
     * must have been a letter and we know that the integer has letters in its hex
     * representation.
     */
    char leftover;
    converter >> leftover;

    return !converter.fail();
}
```

STL Containers, Part I

Problem 5a. If we want to cycle the elements of a container, then our best options are the `deque` and the `queue`. Both of these choices let us quickly move the front element of the container to the back; the `deque` with `pop_front` and `push_back` and the `queue` with `push` and `pop`.

Cycling the elements of a `stack` is impossible without having some external structure that can store the `stack`'s data, so this is not a good choice. While it's possible to cycle the elements of a `vector` using `push_back` and `erase`, doing so is very inefficient because the `vector` will have to shuffle all of its elements down to fill in the gap at the beginning of the container. Remember, if you ever want to add and remove elements at the beginning or end of a `vector`, the `deque` is a better choice.

Problem 5b. For this solution we'll use an STL `queue`, since we don't need access to any element of the key list except the first. Then one solution is as follows:

```
string VigenereCipher(string toEncode, queue<int> values)
{
    for(int k = 0; k < toEncode.length(); ++k)
    {
        toEncode[k] += values.front(); // Encrypt the current character
        values.push(values.front());  // Add the current key to the back.
        values.pop();                 // Remove the current key from the front.
    }
}
```

STL Iterators

Problem 5. One possible implementation of the function is as follows:

```
vector<string> LoadAllTokens(string filename)
{
    vector<string> result;

    /* Open the file, if we can't, just return the empty vector. */
    ifstream input(filename.c_str());
    if(input.fail()) return result;

    /* Using the istream_iterator iterator adapter, read everything out of the
     * file. Since by default the streams library uses whitespace as a separator
     * character, this reads in all of the tokens.
     */
    result.insert(result.begin(),
                  istream_iterator<string>(input), istream_iterator<string>());

    return result;
}
```

STL Containers, Part II

Problem 2. We can solve this problem by loading all of the values in the map into a `map<string, int>` associating a value in the initial map with its frequency. We then can get the number of duplicates by adding up all of the entries in the second map whose value is not one (i.e. at least two elements have the same value). This can be implemented as follows:

```

int NumDuplicateValues(map<string, string>& input)
{
    map<string, int> counter;

    /* Iterate over the map updating the frequency count. Notice that we are
     * checking for the number of duplicate values, so we'll index into the map by
     * looking at itr->second. Also note that we don't check for the case where
     * the map doesn't already contain this key. Since STL containers initialize
     * all stored integers to zero, if the key doesn't exist a fresh pair will be
     * created with value zero.
     */
    for(map<string, string>::iterator itr = input.begin();
        itr != input.end(); ++itr)
        ++counter[itr->second];

    int result = 0;
    /* Now iterate over the entries and accumulate those that have at least value
     * two.
     */
    for(map<string, string>::iterator itr = input.begin();
        itr != input.end(); ++itr)
        if(itr->second > 1) result += itr->second;

    return result;
}

```

Problem 5. There are many good solutions to this problem. My personal favorite is this one:

```

void CountLetters(istream& input, map<char, int>& freqMap)
{
    char ch;
    while(input.get(ch)) ++freqMap[ch];
}

```

This code is dense and relies on several properties of the stream library and the STL. First, the member function `get` accepts as input a `char` by reference, then reads in a single character from the stream. On success, the function fills the `char` with the read value. On failure, the value is unchanged and the stream goes into a fail state. The `get` function then returns a reference to the stream object that did the reading, meaning that `while(input.get(ch))` is equivalent to

```

while(true)
{
    input.get(ch);
    if(!input) break;

    /* ... body of loop ... */
}

```

And since `!input` is equivalent to `input.fail()`, this one-liner will read in a character from the file, then break out of the loop if the read failed.

Once we've read in the character, we can simply write `++freqMap[ch]`, since if the key already exists we're incrementing the older value and if not a new key/value pair will be created with value 0, which is then incremented up to one.

Problem 6. As mentioned in the hint, the trick is to use the structure of lexicographic comparisons to construct a pair of strings lower- and upper-bounding all strings with the given prefix. For example, suppose that we want to find all words beginning with the prefix `anti`. Now, any word beginning with `anti` must compare lexicographically greater than or equal to `anti`, since the first four characters will match but the word beginning with `anti` must also be longer than `anti`. For example, `antigen` and `antidisestablishmentarianism` both compare greater than `anti` since they have the same prefix as `anti` but are longer.

The next observation is that any word that *doesn't* start with `anti` falls into one of two categories – those that compare lexicographically less than `anti` and those that compare lexicographically greater than `anti`. The first of these sets can be ignored, but how can we filter out words with non-`anti` prefixes that compare lexicographically greater than `anti`? The trick is to note that if the word doesn't have `anti` as a prefix, then somewhere in its first four letters it must disagree with `anti`. If we take the next lexicographically-higher prefix than `anti` (which is formed by incrementing the last letter), we get `antj`. This is the smallest possible prefix any word not starting by `anti` can have. Moreover, every word that starts with `anti` compares lexicographically less than `antj`, and so if we only look at words that compare lexicographically greater than or equal to `anti` and lexicographically less than `antj`, we have all of the words that start with `anti`. Using the set's `lower_bound` function, we can find which words in the set match these criteria efficiently (in $O(\lg n)$ time) using the following code:

```
void PrintMatchingPrefixes(set<string>& lexicon, string prefix)
{
    /* We'll assume the prefix is nonempty in this next step. */
    string nextPrefix = prefix;
    ++nextPrefix[nextPrefix.size() - 1];

    /* Compute the range to iterate over. We store these iterators outside of the
     * loop so that we don't have to recompute them every time.
     */
    set<string>::iterator end = lexicon.lower_bound(nextPrefix);
    for(set<string>::iterator itr = lexicon.lower_bound(prefix); itr != end; ++itr)
        cout << *itr << endl;
}
```

STL Algorithms

Problem 1. Printing out a vector is easy thanks to `ostream_iterator` and `copy`. Recall that an `ostream_iterator` is an iterator which prints out the values stored in it to `cout`, and that the `copy` algorithm accepts three inputs – two iterators defining a range to copy and one iterator representing the destination – then copies the input range to the output source. Thus we can print out a vector as follows:

```
void PrintVector(vector<int>& v)
{
    copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
}
```

Problem 4. By default, if you compare two strings to one another using `<`, the result is whether the first string lexicographically precedes the second string. Thus if we have a `vector<string>` called `v`, calling `sort(v.begin(), v.end())` will sort the input lexicographically. In our case, though, we want to “hack” the `sort` function so that it always puts “Me First!” at the front of the sorted ordering. To do this, we'll write a custom callback function that performs a default string comparison if neither string is “Me First!” but which skews the result otherwise. Here's one implementation:

```

const string kWinnerString = "Me First!";
bool BiasedSortHelper(string one, string two)
{
    /* Case one: Neither string is the winner string. Just do a default
     * comparison.
     */
    if(one != kWinnerString && two != kWinnerString)
        return one < two;

    /* Case two: Both strings are the winner string. Then return false because the
     * string isn't less than itself.
     */
    if(one == kWinnerString && two == kWinnerString)
        return false;

    /* Case three: one is the winner string, two isn't. Return true to bias
     * the sort so that the winner string comes first.
     */
    if(one == kWinnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't */
    return false;
}

```

The implementation of `BiasedSort` is then

```

void BiasedSort(vector<string>& v)
{
    sort(v.begin(), v.end(), BiasedSortHelper)
}

```

Problem 6: One implementation is as follows:

```

int count(vector<int>::iterator start, vector<int>::iterator end, int value)
{
    int result = 0;
    for(; start != end; ++start)
        if(*start == value) ++result;

    return result;
}

```

In the chapter on templates, you'll see how to generalize this function to operate over any type of iterators.

Pointers and References

Problem 5. If we allocate memory by writing `int* myIntPtr = new int`, C++ will only give us space to hold a single integer. However, the code `myIntPtr[0] = 42` is perfectly safe. Recall that `myIntPtr[0]` means to go to the address pointed at by `myIntPtr`, move forward zero elements, then return the value there. But this is equivalent to just looking up the object directly pointed at by `myIntPtr`, and in fact `myIntPtr[0] = 42` is completely equivalent to `*myIntPtr = 42`.

However, `myIntPtr[1] = 42` instructs C++ to access the element one after the element pointed at by `myIntPtr`. We don't own the memory after our single `int`, so this line writes 42 into a memory location. It probably will result in some sort of crash, either immediately or later on when the memory allocator gets confused.

C Strings

Problem 1. The code `myString = "String" + '!'` just *looks* right, doesn't it? It seems like we're concatenating an exclamation point on to the end of a string and then assigning the new string to `myString`. Had we been working with C++ `string` objects, this would be true, but remember that `"String"` is a C string and that `+` means pointer arithmetic rather than concatenation. In fact, what this code does is obtain a pointer to the string `"String"` somewhere in memory, then advance that pointer by the numerical value of the exclamation point character (thirty-three) and store the resulting pointer in `myString`. This results in a string that's pointing into random memory we don't own, resulting in (surprise!) undefined behavior.

The Preprocessor

Problem 5. This restriction exists because the preprocessor is a *compile-time* construct whereas functions are a *runtime* construct. That is, the code that you write for a function is executed only when the program already runs, and preprocessor directives execute before the program begins running (or has even finished compiling, for that matter). If a preprocessor directive were to execute a C++ function, the compiler would need to compile and run that function during preprocessing, which might cause dependency issues if the function referenced code that hadn't yet been preprocessed, or would have to defer preprocessing to runtime, defeating the entire purpose of the preprocessor.

Problem 9. The code for a not-reached macro is actually simpler than that for an assert macro because we don't need to verify any conditions and instead can immediately abort. We can begin by writing the code that actually performs the action associated with the not-reached statement:

```
void DoCS106LNotReached(string message, int line, string filename)
{
    cerr << "CS106LNotReached failed: " << message << endl;
    cerr << "Line number: " << line << endl;
    cerr << "Filename: " << filename << endl;
    abort();
}
```

```
#define CS106LNotReached(msg) DoCS106LNotReached(msg, __LINE__, __FILE__)
```

Next, we need to disable the macro in case `NO_CS106L_NOTREACHED` is defined. This can be done as follows:

```
#ifndef NO_CS106L_NOTREACHED
```

```
void DoCS106LNotReached(string message, int line, string filename)
{
    cerr << "CS106LNotReached failed: " << message << endl;
    cerr << "Line number: " << line << endl;
    cerr << "Filename: " << filename << endl;
    abort();
}
```

```
#define CS106LNotReached(msg) DoCS106LNotReached(msg, __LINE__, __FILENAME__)
```

```
#else
```

```
#define CS106LNotReached(msg) /* Nothing */
```

```
#endif
```

Problem 11. Adding a `NOT_A_COLOR` sentinel to the `Color` enumeration is much easier than it sounds. The X Macro code we have for generating the `Color` enumeration is currently

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

These preprocessor directives expand out to the full list of colors with a comma following the name of the last color. Thus all we need to do is change the code to look like this:

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
    NOT_A_COLOR
};
```

Now, the `enum` contains a constant called `NOT_A_COLOR` that follows all other colors.

Problem 13. We want to change the definition of the `Color` enumeration so that the names of the colors are pre-faced with `eColor_`. Thus the code we want to generate should look like this:

```
enum Color {
    eColor_Red,
    eColor_Green,
    /* ... */
    eColor_White
};
```

Recall that the original X Macro code we had for automatically generating the `Color` enumeration was

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

We can modify this to generate the above code by using the token-pasting operator `##` to concatenate `eColor_` before the name of each of the colors. The resulting code is

```
enum Color {
    #define DEFINE_COLOR(color, opposite) eColor_##color,
    #include "color.h"
    #undef DEFINE_COLOR
};
```

Introduction to Templates

Problem 1. `copy_if` accepts four parameters – two iterators defining an input range, one iterator defining the output range, and a predicate function determining whether we should copy a particular element. Since we can provide any sort of input iterator, any sort of output iterator, and a predicate that could theoretically accept any type, we'll templateize `copy_if` over the types of the arguments, as shown here:

```

template <typename InputIterator, typename OutputIterator, typename Predicate>
inline OutputIterator copy_if(InputIterator start,
                             InputIterator end,
                             OutputIterator where,
                             Predicate fn)
{
    /* ... */
}

```

Note that although the type of the predicate function depends on the type of iterators (that is, if we're iterating over `strings` we can't give a function accepting an `int`), we've templated the function with respect to the predicate. This gives the client more leeway in what predicates they can provide. For example, if the iterators iterate over a `vector<int>`, they could provide a predicate function accepting a `double`. Later when we cover functors you'll see a more general reason to template the predicate parameter.

Now all that's left to do is write the function body. Fortunately this isn't too tricky – we just keep advancing the `start` iterator forward, checking if the element iterated over passes the predicate and copying the element if necessary. The final code looks like this:

```

template <typename InputIterator, typename OutputIterator, typename Predicate>
inline OutputIterator copy_if(InputIterator start,
                             InputIterator end,
                             OutputIterator where,
                             Predicate fn)
{
    for(; start != end; ++start)
    {
        if(fn(*start))
        {
            *where = *start;
            ++where;
        }
    }
    return where;
}

```

As an FYI, you will sometimes see the code

```

*where = *start;
++where;

```

Written as

```

*where++ = *start;

```

This is a trick that relies on the fact that the `++` operator binds more tightly than the `*` operator. Thus the code is interpreted as `*(where++) = *start`, meaning that we advance the `where` iterator by one step, then store in the location it use to point at the value of `*start`. I personally find this syntax more attractive than the longhand version, but it is admittedly more confusing.

Problem 6. Recall that the code for `GetInteger` is as follows:

```
int GetInteger()
{
    while(true)
    {
        stringstream converter(GetLine());
        int result;

        converter >> result;
        if(!converter.fail())
        {
            char leftover;
            converter >> leftover;

            if(converter.fail()) return result;
            cout << "Unexpected character: " << leftover << endl;
        }
        else cout << "Please enter an integer." << endl;

        cout << "Retry: ";
    }
}
```

To templatzize this function over an arbitrary type, we need to change the return type, the type of `result`, and the error message about entering an integer. The modified code is shown here:

```
template <typename ValueType> ValueType GetValue(string type)
{
    while(true)
    {
        stringstream converter(GetLine());
        ValueType result;

        converter >> result;
        if(!converter.fail())
        {
            char leftover;
            converter >> leftover;

            if(converter.fail()) return result;
            cout << "Unexpected character: " << leftover << endl;
        }
        else cout << "Please enter " << type << endl;

        cout << "Retry: ";
    }
}
```

const

Problem 5. At first it seems like it should be safe to convert an `int **` into a `const int **`. After all, we're just adding more `consts` to the pointer, which restricts what we should be able to do with the pointer. How could we possibly use this to subvert the type system? The answer is the following chain of assignments that allow us to overwrite a `const int`:

```

const int myConstant = 137; // Legal
int * evilPtr; // Legal, uninitialized

/* This next line is not legal C++ because &evilPtr is an int** and badPtr is a
 * const int **. Watch what happens if we allow this.
 */
const int** badPtr = &evilPtr;

/* Dereference badPtr and assign it the address of myConstant. &myConstant is a
 * const int * and badPtr is a const int*, so the assignment is legal. However,
 * since badPtr points to evilPtr, this assigns evilPtr the address of the
 * myConstant variable, which is const!
 */
*badPtr = &myConstant;

/* This would overwrite a const variable. */
*evilPtr = 42;

```

This is a subtle edge case, but because it's possible C++ explicitly disallows it.

Problem 6. The initial interface for the CS106B/X Vector class is reprinted here:

```

template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size();
    bool isEmpty();

    ElemType getAt(int index);
    void setAt(int index, ElemType value);

    ElemType & operator[](int index);

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(ElemType elem));
    template <typename ClientDataType>
        void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
                    ClientDataType & data);

    Iterator iterator();
};

```

The first task in `const`-correcting this is marking non-mutating operations `const`. This is reasonably straightforward for most of the functions. The interesting case is the `operator[]` function, which returns a reference to the element at a given position. This function needs to be `const`-overloaded since if the `Vector` is `const` we want to hand back a `const` reference and if the `Vector` is non-`const` we want to hand back a non-`const` reference. The updated interface is shown here:

```

template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    ElemType getAt(int index) const;
    void setAt(int index, ElemType value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(ElemType elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};

```

Next, we'll update the class by passing all appropriate parameters by reference-to-const rather than by value. This results in the following interface:

```

template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    ElemType getAt(int index) const;
    void setAt(int index, const ElemType& value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(const ElemType& elem);
    void insertAt(int index, const ElemType& elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(const ElemType& elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(const ElemType& elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};

```

Finally, we'll change the return type of `getAt` to return a `const` reference to the element in the `Vector` rather than a full copy, since this reduces the cost of the function. The final version of the `Vector` is shown below:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    const ElemType& getAt(int index) const;
    void setAt(int index, const ElemType& value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(const ElemType& elem);
    void insertAt(int index, const ElemType& elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(const ElemType& elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(const ElemType& elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};
```

static

Problem 2. The `UniquelyIdentified` class can be implemented by having a static variable inside the class that tracks the most recently used ID, as well as a non-static variable for each instance that tracks the particular class's unique ID. This can be implemented as follows:

```
class UniquelyIdentified
{
public:
    UniquelyIdentified();

    int getUniqueID() const;
private:
    static int lastUsedID;
    const int currentID;
};

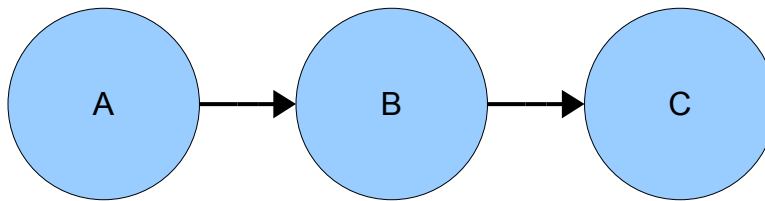
/* Remember, this must go outside the class! */
int UniquelyIdentified::lastUsedID = 1;

UniquelyIdentified::UniquelyIdentified() : currentID(lastUsedID)
{
    ++lastUsedID;
}
```

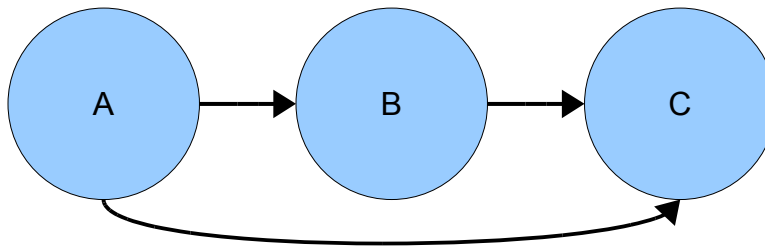
```
int UniquelyIdentified::getUniqueID() const
{
    return currentID;
}
```

Conversion Constructors

Problem 5. C++ will only apply at most one conversion constructor at a time to avoid getting into ambiguous situations. For example, suppose that we start off with types A, B, and C such that A is convertible to B and B is convertible to C. Then we can think of this graphically as follows:



Now, suppose we introduce another conversion into this mix, this time directly from A to C, as shown here:



If we try to implicitly convert from an A to a C, which sequence of conversions is correct? Do we first convert the A to a B and then convert that to a C, or just directly convert the A to a C? There's no clear right answer here, and to avoid confusions like this C++ sidesteps the issue by only applying one implicit conversion at a time.

Copy Constructors and Assignment Operators

Problem 1. The reason that this code is problematic is that at some point in the assignment operator, all of the resources that were allocated by the current class instance need to be cleaned up. However, we never allocated any resources, and none of the class's data members have been initialized. Trying to clean up garbage almost always results in a program crash. In general, you should not implement copy constructors in terms of assignment operators

Problem 2. It is illegal to write a copy constructor that accepts its parameter by value because this causes a circular dependency. Recall that the copy constructor is invoked whenever a function is passed by value into a function, so if the copy constructor itself took its parameter by value it would need to call itself to initialize the parameter by value, but then it would have to initialize the parameter in that function call by invoking itself, etc. This causes a compile-time error rather than a runtime error, by the way.

The assignment operator doesn't have this problem because if the assignment operator were to accept its parameter by value, the copy is made by the copy constructor, which is an entirely different function.

Operator Overloading

Problem 3. The other operators can be defined as follows:

```
A < B      ⇔      A < B
A <= B     ⇔      !(B < A)
A == B     ⇔      !(A < B || B < A)
A != B     ⇔      A < B || B < A
A >= B     ⇔      !(A < B)
A > B      ⇔      B < A
```

Problem 7. For reference, here's the broken interface of the `iterator` class:

```
class iterator
{
public:
    bool operator== (const iterator& other);
    bool operator!= (const iterator& other);

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

There are five problems in this implementation. The first two have to do with the declarations of the `==` and `!=` operators. Since these functions don't modify the state of the `iterator` (or at least, no sensible implementation should), they should both be marked `const`, as shown here:

```
class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

Next, take a look at the return type of `operator*`. Remember that `operator*` is called whenever the iterator is dereferenced. Thus if the iterator is pretending to point to an element of type `ElemType`, this function should return an `ElemType&` (a reference to the value) rather than an `ElemType*` (a pointer to the value). Otherwise code like this:

```
*myItr = 137;
```

Wouldn't compile. The updated interface now looks like this:

```

class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator operator++ ();

    ElemType& operator* () const;
    ElemType* operator-> () const;
};

```

Next, look at the return type of `operator++`. Recall that `operator++` is the prefix `++` operator, meaning that we should be able to write code like this:

```
++(++itr)
```

To increment the iterator twice. Unfortunately, with the above interface this code compiles but does something totally different. Since the return type is `iterator`, the returned object is a *copy* of the receiver object rather than the receiver object itself. Thus the code `++(++itr)` means to increment `itr` by one step, then to increment the *temporary iterator object* by one step. This isn't at all what we want to do, so we'll fix this by having `operator++` return a reference to an iterator, as shown here:

```

class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator& operator++ ();

    ElemType& operator* () const;
    ElemType* operator-> () const;
};

```

We've now fixed four of the five errors, so what's left? The answer's a bit subtle – there's nothing *technically* wrong with this interface any more, but we've left out an important function that makes the interface unintuitive. In particular, we've only defined a prefix `operator++` function, meaning that code like `++itr` is legal but not code like `itr++`. We should thus add support for a postfix `operator++` function. The final version of the `iterator` class thus looks like this:

```

class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator& operator++ ();
    const iterator operator++ (int);

    ElemType& operator* () const;
    ElemType* operator-> () const;
};

```

Functors

Problem 1. `for_each` returns the function passed in as a final parameter so that you can pass in a functor that updates internal state during the loop and then retrieve the updated functor at the end of the loop. For example, suppose that we want to compute the minimum, maximum, and average of a range of data in a single pass. Then we could write the following functor class:

```
class DataSample
{
public:
    /* Constructor initializes minVal and maxVal so that they're always updated,
    * sets the total to zero, and the number of elements to zero.
    */
    DataSample() : minVal(INT_MAX), maxVal(INT_MIN), total(0.0), count(0) {}

    /* Accessor methods. */
    int getMin() const
    {
        return minVal;
    }
    int getMax() const
    {
        return maxVal;
    }
    double getAverage() const
    {
        return total / count;
    }
    int getNumElems() const
    {
        return count;
    }

    /* operator() accepts a piece of input and updates state appropriately. */
    void operator()(int val)
    {
        minVal = min(minVal, val);
        maxVal = max(maxVal, val);
        total += val;
        ++count;
    }
private:
    int minVal, maxVal;
    double total;
    int count;
};
```

Then we can write a function which accepts a range of iterators (presumed to be iterating over `int` values) and then returns a `DataSample` object which contains a summary of that data:


```

template <typename InputIterator>
DataSample AnalyzeSample(InputIterator begin, InputIterator end)
{
    /* Create a temporary DataSample object and pass it through for_each. This
     * then invokes operator() once for each element in the range, resulting in a
     * DataSample with its values set appropriately. We then return the result of
     * for_each, which is the updated DataSample. Isn't that nifty?
     */
    return for_each(begin, end, DataSample());
}

```

Problem 3. AdvancedBiasedSort is similar to the BiasedSort example from the chapter on STL algorithms. If you'll recall, if we assume that the string that should be the winner is stored in a constant `kWinnerString`, the following comparison function can be used to bias the sort so that `kWinnerString` always comes in front:

```

bool BiasedSortHelper(const string& one, const string& two)
{
    /* Case one: Neither string is the winner string. Just do a default
     * comparison.
     */
    if(one != kWinnerString && two != kWinnerString)
        return one < two;

    /* Case two: Both strings are the winner string. Then return false because the
     * string isn't less than itself.
     */
    if(one == kWinnerString && two == kWinnerString)
        return false;

    /* Case three: one is the winner string, two isn't. Then return true to bias
     * the sort.
     */
    if(one == kWinnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't, so return false to bias
     * the sort.
     */
    return false;
}

```

To update this code so that *any* string can be set as the winner string, we'll need to convert this function into a functor that stores the string as a data member. This can be done as follows:

```

class BiasedSortHelper
{
public:
    explicit BiasedSortHelper(const string& winner) : winnerString(winner) {}
    bool operator() (const string& one, const string& two) const;
private:
    string winnerString;
};

```

```

bool BiasedSortHelper::operator()(const string& one, const string& two) const
{
    /* Case one: Neither string is the winner string. Just do a default
     * comparison.
     */
    if(one != winnerString && two != winnerString)
        return one < two;

    /* Case two: Both strings are the winner string. Then return false because the
     * string isn't less than itself.
     */
    if(one == winnerString && two == winnerString)
        return false;

    /* Case three: one is the winner string, two isn't. Then return true to bias
     * the sort.
     */
    if(one == winnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't, so return false to bias
     * the sort.
     */
    return false;
}

```

We can then implement `AdvancedBiasedSort` as follows:

```

void AdvancedBiasedSort(vector<string>& v, const string& winner)
{
    sort(v.begin(), v.end(), BiasedSortHelper(winner));
}

```

Problem 7. We want to compute the value of $\sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$ as applied to a range of data. This breaks down into four smaller tasks:

1. Compute the average of the data set.
2. Compute the value of the inner sum.
3. Divide the total by the number of elements.
4. Return the square root of this value.

We can use the `sqrt` function exported by `<cmath>` to perform the square root and `accumulate` for the rest of the work. We'll begin by computing the average of the data, as shown here:

```

template <typename ForwardIterator>
double StandardDeviation(ForwardIterator begin, ForwardIterator end)
{
    const ptrdiff_t numElems = distance(begin, end);
    const double average = accumulate(begin, end, 0.0) / numElems;
    /* ... */
}

```

Note that we've computed the size of the range using the `distance` function, which efficiently returns the number of elements between two iterators. We've stored the number of elements in a variable of type `ptrdiff_t`, which, as the name suggests, is a type designed to hold the distance between two pointers. Technically speaking

we should query the iterator for the type of value returned when computing the distance between the iterators, but doing so is beyond the scope of this text.

Now, let's see how we can use `accumulate` to evaluate the sum. Recall that the four-parameter version of `accumulate` allows us to specify what operation to apply pairwise to the current accumulator and any element of the range. In our case, we'll want to add up the sum of the values of $(x_i - \bar{x})^2$ for each element, so we'll construct a functor that stores the value of the average and whose `operator()` function takes the accumulator and the current element, computes $(x_i - \bar{x})^2$, then adds it to the current accumulator. This is shown here:

```
class StandardDeviationHelper
{
public:
    explicit StandardDeviationHelper(double average) : mean(average) {}
    double operator() (double accumulator, double currentValue) const
    {
        const double expr = currentValue - mean; //  $x_i - \bar{x}$ 
        return accumulator + expr * expr;        //  $(x_i - \bar{x})^2$ 
    }
private:
    const double mean;
};
```

Given this helper functor, we can now write the following:

```
template <typename ForwardIterator>
double StandardDeviation(ForwardIterator begin, ForwardIterator end)
{
    const ptrdiff_t numElems = distance(begin, end);
    const double average = accumulate(begin, end, 0.0) / numElems;
    const double sum =
        accumulate(begin, end, 0.0, StandardDeviationHelper(average));
    return sqrt(sum / numElems);
}
```

Problem 8. If we were given an arbitrary C string and told to set it to the empty string, we could do so by calling `strcpy(myStr, "")`, where `myStr` is the string variable. To do this to every string in a range, we can use `for_each` to apply the function everywhere and `bind2nd` to lock the second parameter of `strcpy` in place. This is shown here:

```
template <typename ForwardItr>
void ClearAllStrings(ForwardItr begin, ForwardItr end)
{
    for_each(begin, end, bind2nd(ptr_fun(strcpy), ""));
}
```

This works because `bind2nd(ptr_fun(strcpy), "")` is a one-parameter function that passes the argument as the first parameter into `strcpy` with the second parameter locked as the empty string.

Problem 9. Now that we're dealing with regular C++ strings, we can clear the strings by calling the `.clear()` member function on each of them. Recalling that the `mem_fun_ref` function transforms a member function into a one-parameter function that calls the specified function on the argument, we can write `ClearAllStrings` as follows:

```
template <typename ForwardItr>
void ClearAllStrings(ForwardItr begin, ForwardItr end)
{
    for_each(begin, end, mem_fun_ref(&string::clear));
}
```

Problem 11. The `replace_if` algorithm takes four parameters – two iterators defining a range of elements, a predicate function, and a value – then replaces all elements in the range for which the predicate returns true with the specified value. In our case, we're given a value as a parameter to the `CapAtValue` function and want to replace elements in the range that compare greater than the parameter with the parameter. Using `bind2nd` and the `greater` operator function, we have the following:

```
template <typename ForwardItr, typename ValueType>
void CapAtValue(ForwardItr begin, ForwardItr end, ValueType maxValue)
{
    replace_if(begin, end, bind2nd(greater<ValueType>(), maxValue), maxValue);
}
```

Introduction to Exception Handling

Problem 1. If you put a `catch(...)` clause at the top of a `catch` cascade, then none of the other `catch` handlers will ever catch exceptions thrown by the `try` block. C++ evaluates each `catch` clause in sequence until it discovers a match, and if the first is a `catch(...)` it will always choose that one first.

Problem 4. We want to modify the code

```
void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    /* This might throw an exception! */
    DoSomething(myStack);

    myStack.push(topElem);
}
```

So that if the `DoSomething` function throws an exception we're sure to put the element `topElem` back on the stack before propagating the exception. This can be done as follows:

```

void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    try
    {
        /* This might throw an exception! */
        DoSomething(myStack);
        myStack.push(topElem);
    }
    catch(...)
    {
        myStack.push(topElem);
        throw;
    }

    myStack.push(topElem);
}

```

Problem 5. We want to solve the same problem as in the above case, but by using RAII to manage the resource instead of manually catching and rethrowing any exceptions. We'll begin by defining an `AutomaticStackManager` class that takes in a reference to a `stack`, then pops the top and stores the result internally. When the destructor executes, we'll push the element back on. This looks like this:

```

template <typename ElemType> class AutomaticStackManager
{
public:
    explicit AutomaticStackManager(stack<ElemType>& s) :
        toManage(s), topElem(s.top())
    {
        toManage.pop();
    }

    ~AutomaticStackManager()
    {
        toManage.push(topElem);
    }
private:
    stack<ElemType>& toManage;
    const ElemType topElem;
};

```

Notice that we've templated this class with respect to the type of element stored in the `stack`, since there's nothing special about `string`. This is in general a good design philosophy – if you don't need to specialize your code over a single type, make it a template.

We can then rewrite the function as follows:

```

void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    AutomaticStackManager<string> autoCleanup(myStack);
    DoSomething(myStack);
}

```

Notice how much cleaner and shorter this code is than before – by having objects manage our resources we're sure that we won't leak any resources here. True, we had to write a bit of code for `AutomaticStackManager`, but provided that we use it in more than one circumstance the savings in code simplicity over the manual catch-and-rethrow approach are impressive.

Introduction to Inheritance

Problem 1. We don't have to worry that a pointer of type `Document*` points to an object of concrete type `Document` because `Document` is an abstract class and thus can't be instantiated. You do not need to worry about a pure virtual function call realizing that there's no actual code to execute since C++ will raise a compile-time error if you try to instantiate an abstract class.

Problem 4. One possible implementation for `DerivativeFunction` is shown here:

```

class DerivativeFunction: public Function
{
public:
    explicit DerivativeFunction(Function* toCall) : function(toCall) {}

    virtual double evaluateAt(double where) const
    {
        const double kEpsilon = 0.00001; // Small Δx
        return (function->evaluateAt(where + kEpsilon) -
                function->evaluateAt(where - kEpsilon)) / (2 * kEpsilon);
    }
private:
    Function *const function;
}

```

Here, the constructor accepts and stores a pointer to an arbitrary `Function`, and the `evaluateAt` function invokes the virtual `evaluateAt` function of the stored function at the proper points to approximate the derivative.

Bibliography

- [Str94]: Bjarne Stroustrup, *The Design and Evolution of C++*. Addison-Wesley: 1994.
- [Str09]: Bjarne Stroustrup. "Stroustrup: FAQ" URL: http://www.research.att.com/~bs/bs_faq.html#decline. Accessed 7 Jul 2009.
- [Ste07]: Alexander Stepanov. "Short History of STL" URL: <http://www.stepanovpapers.com/history%20of%20STL.pdf>. Accessed 2 Jul 2009.
- [Intel]: Intel Corporation. "965_diagram.gif" URL: http://www.intel.com/assets/image/diagram/965_diagram.gif. Accessed 7 Jul 2009.
- [Pic96]: NeilFred Picciotto. "Neil/Fred's Gigantic List of Palindromes" URL: <http://www.derf.net/palindromes/old.palindrome.html>. Accessed 10 Jul 2009.
- [Str09.2]: Bjarne Stroustrup. "Stroustrup: C++" URL: <http://www.research.att.com/~bs/C++.html>. Accessed 24 Jun 2009.
- [Spo08]: Joel Spolsky, *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress: 2008.
- [GNU]: GNU Project. "strfry" URL: http://www.gnu.org/s/libc/manual/html_node/strfry.html#strfry. Accessed 4 Aug 2009.
- [Sut98]: Herb Sutter. "Advice from the C++ Experts: Be Const-Correct" URL: <http://www.gotw.ca/publications/advice98.htm>. Accessed 10 Jun 2009.
- [MCO]: Mars Climate Orbiter Mishap Investigation Board Phase I Report. November 10, 1999. URL: ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
- [Mey05]: Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley: 2005.
- [Str09.3]: Bjarne Stroustrup. "C++0x FAQ" URL: <http://www.research.att.com/~bs/C++0xFAQ.html#think>. Accessed 10 Jul 2009.
- [Dij68]: Edsger Dijkstra. "Go To Statement Considered Harmful." *Communications of the ACM*, Vol. 11, No. 3, Mar 1968, pg. 147-148

Index

A

- abort**, 172, 278
- adaptable functions, 375, 381
 - operator functions, 380
- address-of operator, 138
- assert**, 170, 229
- assignment operators, 263
 - copy-and-swap, 277
 - disabling, 275
 - for derived classes, 452
 - implementation of, 268
 - return value of, 272
 - self-assignment, 270
- auto_ptr**, 398
 - and exception-safety, 401

B

- BCPL, 5
- binary_function**, 377
- bind1st**, 378
 - implementation, 385
- bind2nd**, 378, 416
- bitwise equivalence, 273
- Bjarne Stroustrup, 5
 - and pure virtual functions, 437
 - and the preprocessor, 167
- Boost C++ libraries, 23
- busy loop, 63

C

- C strings,
 - in-memory layout, 148
 - strcat**, 150
 - strchr**, 153
 - strcmp**, 151
 - strcpy**, 150
 - strlen**, 149
 - strncpy**, 153
 - strstr**, 152
 - terminating null, 148
- c_str()**, 26
- C++0x, 481
 - lambda expressions, 489
 - rvalue references, 485
 - sizeof...**, 494
 - static_assert**, 494
 - type inference, 481

- unique_ptr**, 488
 - variadic templates, 490
- Caesar cipher, 52
- casting operators, 457
 - const_cast**, 505
 - dynamic_cast**, 458
 - reinterpret_cast**, 505
 - static_cast**, 64, 66, 457, 507
- catch**, 393
 - catch(...)**, 397
- cerr**, 26
- cin**, 25
- clock**, 63
- clock_t**, 64
- CLOCKS_PER_SEC**, 63
- complex**, 426
- const**, 205
 - bitwise **constness**, 211
 - const** and pointers, 206
 - const** and **static**, 252
 - const** member functions, 207, 217
 - const** references, 209, 216
 - const-correctness**, 215, 282
 - semantic **constness**, 211
- conversion constructors, 255, 285
- copy constructors, 263
 - disabling, 275
 - for derived classes, 452
 - implementation of, 265
- copy semantics, 399, 485
- cout**, 25
- ctime**, 266

D

- default arguments, 241
- defined** (preprocessor predicate), 162
- delete**, 142
- delete[]**, 144
- deterministic finite automaton, 100
- dimensional analysis, 331
- distance**, 532
- dumb pointer, 315

E

exception handling, 393
 and resource leaks, 395
 catch-and-rethrow, 396
 resource acquisition is initialization, 400
explicit, 258, 285
external polymorphism, 474

F

free, 506
friend, 305
functors, 366
 as comparison objects, 371
 as parameters to functions, 369
fundamental theorem of software engineering, 465

G

getline, 34, 61, 214
goto, 508
 Go To Statement Considered Harmful, 508
grid, 345

H

higher-order programming, 373

I

ifstream, 26, 58, 203, 213
implicit conversions, 255
include guard, 163
inheritance, 429
 abstract classes, 437
 base classes, 430
 copy functions for derived classes, 452
 derived classes, 430
 disallowing copying with, 452
 function overrides, 438
 is-a relationship, 430
 polymorphism, 436
 private inheritance, 453
 pure virtual functions, 437
 slicing, 456
inline functions, 166
invalid_argument, 394, 423
isalpha, 128
ispunct, 118
isspace, 39, 129

L

lexicographical ordering, 357
lvalue, 137, 294

M

macros, 164
 dangers of, 165
make_pair, 94, 105, 233
malloc, 506
mem_fun, 377
mem_fun_ref, 377
member initializer list, 237
memory segments, 148
mixin classes, 456
monoalphabetic substitution cipher, 125
move semantics, 399, 485
mutable, 212

N

namespace, 18
new, 142
new[], 144
nondeterministic finite automaton, 107
not1, 379
 implementation, 381
not2, 379
NULL, 141
numeric_limits, 373

O

object construction, 237
 in derived classes, 446
ofstream, 26
operator overloading, 291
 as free functions, 302
 compound assignment operators, 297
 element selection operator, 296, 352
 function call operator, 366, 469
 general principles, 293
 increment and decrement operators, 304
 mathematical operators, 300, 339
 member selection operator, 310, 320
 pointer dereference operator, 309, 320
 relational operators, 305, 356
 stream insertion operator, 307
 unary minus operator, 299

P

pair, 94, 104, 232
palindromes, 127
pointers, 137
 pointer arithmetic, 151
 pointer assignment, 139
 pointer dereference, 139
 void*, 506
principle of least astonishment, 292

printf, 504
protected, 440
proxy objects, 353
ptr_fun, 377
 implementation, 384
ptrdiff_t, 532

Q

qsort, 506

R

rand, 66, 254
RAND_MAX, 66
reference counting, 315
references, 136
row-major order, 346
rule of three, 265, 286
rvalue, 294

S

scanf, 504
segmentation fault, 149
semantic equivalence, 273
sibling access, 267
Simula, 5
singleton class, 278
smart pointer, 315
sqrt, 532
srand, 66, 254
standard deviation, 389
static, 247
 static data members, 247
 static member functions, 251
STL algorithms, 113
 accumulate, 113, 372
 binary_search, 116
 copy, 118, 267, 288, 423
 count, 156
 count_if, 365
 equal, 117, 128, 202
 fill, 123
 find, 116
 find_if, 313, 419
 for_each, 119, 389
 generate, 389
 includes, 117
 lexicographical_compare, 358
 lower_bound, 117, 372
 random_shuffle, 116
 remove, 118
 remove_copy, 118

remove_copy_if, 118, 389
 remove_if, 118, 128
 replace_if, 390
 reverse, 128
 rotate, 116
 set_difference, 117
 set_intersection, 117
 set_symmetric_difference, 117
 set_union, 117
 sort, 115, 389
 swap, 277
 swap_ranges, 420
 transform, 119, 129, 202, 373, 416, 418
STL containers, 43
 deque, 49, 56
 map, 95, 104, 227
 multimap, 96, 109
 multiset, 96
 queue, 44, 430, 436
 set, 91, 104
 stack, 44
 vector, 45, 54, 213, 346, 507
STL iterators, 81
 begin(), 82
 const_iterator, 210
 defining ranges with, 82
 end(), 83
 iterator adapters, 86
 back_insert_iterator, 86
 back_inserter, 87
 front_insert_iterator, 88
 front_inserter, 88
 insert_iterator, 88
 inserter, 88, 117
 istream_iterator, 88, 130, 213
 istreambuf_iterator, 88, 203
 ostream_iterator, 86, 288
 ostreambuf_iterator, 88
 iterator categories, 84
 iterator, 81
 reverse_iterator, 84, 128
stream extraction operator, 25
stream failure, 32
stream insertion operator, 25

stream manipulators, 27

- boolalpha**, 31
- dec**, 32
- endl**, 27
- hex**, 32
- left**, 29
- noboolalpha**, 31
- oct**, 32
- right**, 29
- setfill**, 30
- setw**, 29, 31

stringstream, 35, 130

syntax sugar, 293

system, 64

T

templates, 183

- implicit interface, 198
- integer template arguments, 338
- template classes, 183
 - member functions of, 185
- template functions, 195
- template member functions, 199
 - of template classes, 200
- type inference, 195, 384, 472
- typename**, 183, 200

temporary object syntax, 288, 348, 369

this, 249

- and assignment operators, 271
- and static member functions, 251
- as an invisible parameter, 250

throw, 393

- lone **throw** statement, 397

time, 66, 254

time_t, 266

tolower, 119

try, 393

typedef, 91

- typedef struct**, 512

U

unary_function, 376

undefined behavior, 143

union-find data structure, 225

using namespace std, 17

V

valarray, 313

Vigenère cipher, 52

virtual, 434

- in constructors and destructors, 449
- virtual destructors, 441

= 0, 434

virtual function table, 443, 474

X

X Macro trick, 173

Z

zero-overhead principle, 133

__DATE__, 169

__FILE__, 169

__LINE__, 169

__TIME__, 169

?: operator, 70, 166

(stringizing operator), 169

(token-pasting operator), 170

#define, 158

- dangers of, 159

#elif, 162

#else, 162

#endif, 162

#if, 162

#ifdef, 163

#ifndef, 163

#include, 157

#undef, 173

<algorithm>, 113

<cassert>, 170

<cctype>, 118, 123

<cmath>, 123, 532

<complex>, 426

<cstdlib>, 142

<cstdio>, 503

<cstdliblib>, 64, 66, 172, 254

<cstring>, 149

<ctime>, 63, 254, 266

<fstream>, 26

<functional>, 375

- implementation of, 381

<iomanip>, 29

<iostream>, 17

<iterator>, 86

<limits>, 373

<memory>, 398

<numeric>, 113

<sstream>, 36

<stdexcept>, 394

<string>, 19

<utility>, 94

<valarray>, 313