

LE C++ POUR LES **PROS**

OURS BLANC DES CARPATHES™

ISIMA–1999

Table des matières

1. LA FORME CANONIQUE DE COPLIEN	4
1.1 CONSTRUCTEUR PAR DÉFAUT	4
1.2 CONSTRUCTEUR DE RECOPIE	5
1.2.1 BUT DU CONSTRUCTEUR DE RECOPIES	5
1.2.2 PROTOTYPE DU CONSTRUCTEUR DE RECOPIE	5
1.2.3 QUAND EST IL NÉCESSAIRE DE CRÉER UN CONSTRUCTEUR DE RECOPIE ?	6
1.3 LE DESTRUCTEUR	6
1.4 OPÉRATEUR D’AFFECTATION	7
1.4.1 PROTOTYPE DE L’OPÉRATEUR D’AFFECTATION	7
1.4.2 DIFFICULTÉ SUPPLÉMENTAIRE PAR RAPPORT AU CONSTRUCTEUR PAR RECOPIE	7
1.5 EXERCICE RÉSOLU : UNE CLASSE CHAÎNE DE CARACTÈRES	7
1.5.1 STRUCTURE :	7
1.5.2 MISE EN ŒUVRE	8
1.5.3 CODE RÉSULTANT	8
1.6 EXERCICE RÉSOLU : DIFFÉRENCE ENTRE CONSTRUCTION ET AFFECTATION	10
1.7 LES CONSTRUCTEURS NE SUPPORTENT PAS LE POLYMORPHISME !	11
2. DE LA SURCHARGE DES OPÉRATEURS ET DE LA BONNE UTILISATION DES RÉFÉRENCES EN GÉNÉRAL	13
2.1 BUTS	13
2.2 CHOISIR ENTRE MÉTHODE ET FONCTION	13
2.2.1 UN EXEMPLE PARTICULIER : LES OPÉRATEURS DE REDIRECTION	13
2.2.2 UNE RÈGLE BIEN UTILE	14
2.2.3 RÉCAPITULATIF	15
2.3 QUELS OPÉRATEURS NE PAS SURCHARGER ?	15
2.4 QUELS PARAMÈTRES DOIVENT PRENDRE LES OPÉRATEURS	17
2.4.1 PRÉFÉRER LES RÉFÉRENCES CONSTANTES AUX OBJETS	17
2.4.2 POURQUOI NE PAS TOUJOURS RENVOYER UNE RÉFÉRENCE	19
2.5 QUE DOIVENT RENVOYER LES OPÉRATEURS ?	19
2.6 LE CAS PARTICULIER DES OPÉRATEURS D’INCRÉMENTATION	22
2.7 LE CAS PARTICULIER DES OPÉRATEURS NEW ET DELETE	24
2.7.1 BIEN COMPRENDRE LE FONCTIONNEMENT DE NEW ET DELETE	24
2.7.2 UNE VERSION SPÉCIALE DE NEW : OPERATOR NEW DE PLACEMENT	25
2.7.3 EXERCICE : UN EXEMPLE SIMPLISTE DE SURCHARGE DE NEW ET DELETE	25
3. LA GESTION DES EXCEPTIONS	26
3.1 SCHÉMA GÉNÉRAL D’INTERCEPTION DES EXCEPTIONS	26
3.2 LA LOGIQUE DE TRAITEMENT DES EXCEPTIONS	26
3.3 RELANCER UNE EXCEPTION	27
3.4 UTILISER SES PROPRES EXCEPTIONS	28
3.4.1 DÉCLARER SES EXCEPTIONS	28
3.4.2 UTILISER SES EXCEPTIONS	30
3.5 LES SPÉCIFICATEURS D’EXCEPTION	31
4. LES TRANSTYPAGES EN C++	32

4.1	NOUVELLE SYNTAXE DE L'OPÉRATEUR TRADITIONNEL	32
4.2	NOUVEAUX OPÉRATEURS DE TRANSTYPAGE	33
4.2.1	POURQUOI ?	33
4.2.2	SYNTAXE	33
4.2.3	FONCTIONNALITÉS	34
4.3	LES CONVERSIONS IMPLICITES	37
4.3.1	LES OPÉRATEURS DE CONVERSION IMPLICITES	37
4.3.2	LES CONSTRUCTEURS AVEC UN SEUL PARAMÈTRE	39
5.	EN GUISE DE CONCLUSION	40

Table des illustrations

Figure 3.1	Schéma général de gestion des exceptions en C++	26
Figure 3.1	Hiérarchie d'exceptions du TasMax	29
Figure 4.1	La lignée des objets graphiques.....	35
Programme 1.1	Forme canonique de Coplien.....	4
Programme 1.1	Constructeur par défaut et tableaux.....	5
Programme 1.1	Réalisation de la classe Chaîne	9
Programme 1.1	Exercice sur la forme canonique de Coplien.....	10
Programme 1.1	tentative d'appel polymorphique au sein d'un constructeur	11
Programme 1.2	fonction <code>main</code> de tentative d'appel polymorphique dans un constructeur.....	12
Programme 1.3	Résultat chimérique obtenu par appel polymorphique dans un constructeur	12
Programme 1.4	Résultat (décevant☹) réellement obtenu	12
Programme 2.1	la classe <code>Evaluable</code>	18
Programme 2.2	un opérateur de comparaison générique ?	18
Programme 2.3	la classe <code>Penalites</code>	18
Programme 2.4	Utilisation de l'opérateur <code><</code>	19
Programme 2.5	un opérateur de comparaison générique !	19
Programme 2.1	Déclaration d'une classe <code>Rationnel</code>	20
Programme 2.2	Opérateur de multiplication sur les rationnels	20
Programme 2.3	opérateur de multiplication 2 ^{ème} forme (erronée).....	22
Programme 2.4	opérateur de multiplication 3 ^{ème} forme (erronée).....	22
Programme 2.1	forme canonique des opérateurs de pré et post incrémentation	23
Programme 2.2	écriture de l'opérateur de post incrémentation	24
Programme 2.1	L'opérateur <code>new</code> de placement.....	25
Programme 3.1	Relancer une exception.....	27
Programme 3.1	Déclaration de la classe <code>std::exception</code>	28
Programme 3.2	Exemple d'une hiérarchie d'exceptions.....	30
Programme 3.1	Exemple de gestion des erreurs en provenance du Tas Max	31
Programme 3.1	Tentative de lancement d'une exception non prévue	31
Programme 3.2	Spécificateurs d'exceptions pour les méthodes du TasMax.....	31
Programme 4.1	exemple minimaliste de <code>static_cast</code>	33
Programme 4.1	exemple d'utilisation de <code>const_cast</code>	34
Programme 4.1	utilisation typique de <code>dynamic_cast</code>	36
Programme 4.1	Définition d'un opérateur de conversion implicite depuis <code>Rationnel</code> vers le type <code>double</code>	37
Programme 4.2	Utilisation d'un opérateur de conversion implicite	37
Programme 4.1	Bug insidieux dû à un opérateur de conversion implicite	38
Programme 4.1	Exemple d'erreur liée aux constructeurs à un seul paramètre.....	39
Tableau 2.1	Description des surcharge d'opérateurs.....	15
Tableau 2.1	Opérateurs du C++ ne pouvant être surchargés.....	16
Tableau 2.2	Opérateurs C++ qu'il vaut mieux ne pas surcharger.....	16
Tableau 4.1	Transtypes réalisables avec <code>const_cast</code>	34

1. La forme canonique de Coplien

La forme canonique, dite de *Coplien*, fournit un cadre de base à respecter pour les classes non triviales dont certains attributs sont alloués dynamiquement.

Une classe T est dite sous forme canonique (ou forme normale ou forme standard) si elle présente les méthodes suivantes :

```
class T
{
public:
    T ();                // Constructeur par défaut
    T (const T&);        // Constructeur de copie
    ~T ();               // Destructeur éventuellement virtuel
    T &operator=(const T&); // Operator d'affectation
};
```

Programme 1.1 Forme canonique de Coplien

Etudions chacune des composantes de la forme canonique :

1.1 Constructeur par défaut

Le constructeur par défaut est un constructeur sans argument ou dont chacun des arguments possède une valeur par défaut.

Il est utilisé, par exemple, lorsque l'on crée des tableaux d'objet. En effet, il n'est pas possible de passer un constructeur à l'opérateur **new[]** responsable de la création des tableaux.

En outre, de nombreuses bibliothèques commerciales ou non (Third Party Software), dont la bibliothèque standard du C++, nécessitent un constructeur par défaut pour la fabrication d'objets temporaires.

Si vous ne spécifiez aucun constructeur pour votre classe, alors le compilateur essaye de fournir un constructeur par défaut, qui, selon les environnements de développement, initialise à zéro chacun des membres de la classe ou les laisse vides. Au cas où votre classe agrégerait par valeur des objets, ceux-ci sont initialisés par leur constructeur par défaut s'il existe – autrement, le constructeur par défaut automatique ne peut pas être construit faute de moyen de construction des objets agrégés.

A partir du moment où vous spécifiez au moins un constructeur pour votre classe, le compilateur n'essaie plus de générer de constructeur par défaut automatique considérant que ceci est de votre ressort. De manière générale, il est très dangereux de ne pas fournir de constructeur pour une classe, sauf, peut-être, dans le cas de classes exception sans attribut.

Le Programme 1.2 montre un exemple où il est nécessaire de fournir un constructeur par défaut à la classe Toto.

<pre> class Toto { private: // declarations sans intérêt public: // Toto(int i); // Constructeur }; int main(int, char**) { Toto tableau[3]; // Erreur ! // il faudrait un // constructeur par // défaut ! } </pre>	<pre> class Toto { private: // declarations sans intérêt public: // Toto(int i=3); // Constructeur // par défaut ! }; int main(int, char**) { Toto tableau[3]; // Correct ! // tous les objets // sont construits // avec la valeur 3 } </pre>
--	--

Programme 1.2 Constructeur par défaut et tableaux

1.2 Constructeur de recopie

1.2.1 But du constructeur de recopies

Comme son nom l'indique, le constructeur de recopie est spécialisé dans la *création* d'un objet à partir d'un autre objet pris comme modèle. J'insiste sur le fait qu'il s'agit d'une création, donc à l'intérieur d'une déclaration d'objet. Toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation.

Le constructeur de recopie a également deux autres utilisations spécifiées dans le langage :

- Lorsqu'un objet est passé en paramètre par valeur à une fonction ou une méthode, il y a appel du constructeur par recopie pour générer l'objet utilisé en interne dans celle-ci. Ceci garantit que l'objet original ne peut être modifié ! L'autre solution consiste à utiliser une référence constante.
- Au retour d'une fonction / méthode renvoyant un objet, il y a création d'un objet temporaire par le constructeur de recopie, lequel est placé sur la pile pour usage par le code appelant.

Ces deux fonctionnalités seront étudiées en détail dans les paragraphes concernant le type de retour des surcharges d'opérateurs.

1.2.2 Prototype du constructeur de recopie

La forme habituelle d'un constructeur de recopie est la suivante :

T::T(const T&)

Le paramètre est passé en référence, ce qui assure de bonnes performances et empêche un bouclage infini. En effet, si l'argument était passé par valeur, il devrait être construit par le constructeur de recopie. Hors c'est précisément ce dernier que nous bâtissons : il y aurait un appel récursif infini !

D'autre part, la référence est constante, ce qui garantit que seules des méthodes constantes ne pouvant pas modifier les attributs seront appelables sur l'objet passé en argument. Notons toutefois que cette précaution est une chimère. En effet, le constructeur par recopie étant nécessairement une méthode d'instance appartenant à la classe T, il a visibilité sur les attributs de la classe et peut donc ainsi délibérément modifier les attributs de l'objet passé en argument sans passer par ses méthodes. Il appartient donc au programmeur de ne pas modifier l'objet argument.

1.2.3 Quand est il nécessaire de créer un constructeur de recopie ?

Si nous ne spécifions pas de constructeur par recopie, le compilateur en crée un automatiquement. Celui-ci est très simple : il recopie bit à bit, de manière « optimisée » l'objet source dans l'objet destination.

Ce comportement est absolument parfait pour les classes simples ne réalisant ni agrégation ni allocations dynamiques. Dans ce cas là, vous n'avez pas besoin de fournir un constructeur de recopie car vous ferez probablement moins bien que le compilateur lui même.

Dans tous les autres cas, vous aurez besoin de définir un constructeur de recopie :

Agrégation par valeur : il vous faudra recopier les objets agrégés par valeur, en appelant ... leur constructeur par recopie !

Agrégation par référence (ou pointeur) et allocation dynamique de mémoire : la copie optimisée ne générant que des copies de pointeurs, différents objets utiliseraient les même cases mémoire. D'une part, les modifications de l'un seraient répercutées sur l'autre, mais, plus grave, en l'absence de mécanismes de garde fou, les mêmes blocs de mémoire seraient libérés plusieurs fois par les destructeurs : il est nécessaire de faire une recopie des objets agrégés par référence (avec le constructeur de recopie ...) ou des blocs de mémoire (**new ...**)

1.3 Le destructeur

Le destructeur est là pour rendre les ressources occupées par un objet lorsqu'il arrive en fin de vie, soit qu'il soit alloué statiquement et sorte de portée, soit qu'il ait été alloué dynamiquement à l'aide de **new** et qu'on lui applique **delete** pour le supprimer.

Si un objet agrège statiquement d'autres objets, ceux-ci sont automatiquement détruits : il n'y a pas lieu de s'en soucier.

En revanche, s'il agrège dynamiquement d'autres objets (en particulier par l'intermédiaire de pointeurs), il sera indispensable de les détruire (par un appel à **delete**) sous peine de ne pouvoir disposer à nouveau de la mémoire avant le prochain reboot !

Pour conclure sur le destructeur, celui-ci sera nécessaire dès lors que votre classe réalise de l'agrégation par pointeur ou de l'allocation dynamique de mémoire.

1.4 Opérateur d'affectation

L'opérateur d'affectation est utilisé pour dupliquer un objet dans un autre en dehors des déclarations où ce rôle est dévolu au constructeur de recopie. De fait, il est nécessaire de créer un opérateur lorsqu'il est nécessaire de créer un opérateur de recopie : agrégation et allocation dynamique de mémoire ; dans tous les autres cas, le compilateur crée pour vous un opérateur d'affectation « par recopie bit à bit optimisée » qui frôle la perfection ☺.

1.4.1 Prototype de l'opérateur d'affectation

L'opérateur d'affectation présente typiquement la forme suivante :

$$T\& T::operator=(const T\&)$$

Le paramètre est passé par référence constante pour les raisons explicitées dans le cas du constructeur par recopie.

Cet opérateur renvoie une référence sur T afin de pouvoir le chaîner avec d'autres affectations. Rappelons que l'opérateur d'affectation est associatif à droite, en effet, l'expression $a=b=c$ est évaluée comme : $a=(b=c)$. Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable, aussi, il est naturel de renvoyer une référence.

1.4.2 Difficulté supplémentaire par rapport au constructeur par recopie

Un constructeur travaille toujours sur un matériau vierge : en effet, il est toujours appelé juste après l'allocation de mémoire. Tout au contraire, l'opérateur d'affectation est appelé sur un objet totalement instancié et, bien souvent, ayant déjà un lourd passé derrière lui. Aussi, toute opération d'affectation devra commencer par rétablir un terrain favorable à la recopie en détruisant les blocs de mémoire déjà affectés ainsi que les objets agrégés.

Pour résumer, l'opérateur d'affectation cumule des fonctionnalités limitées de destructeur et de constructeur par recopie.

Notons que la plupart des bibliothèques (dont la STL du C++) nécessitent la disponibilité de l'opérateur d'affectation pour la plupart des classes qu'elles manipulent.

1.5 Exercice résolu : une classe Chaîne de Caractères

Appliquons les principes précédents à la constitution d'une chaîne de caractères mise sous forme canonique de Coplien.

1.5.1 Structure :

Nous allons créer une chaîne toute simple, dotée de 3 attributs :

- Un tableau de caractères (**char ***) contenant la chaîne proprement dite.
- Un entier dénotant la capacité de la chaîne, *i.e.* le nombre de caractères qu'elle peut contenir à un instant donné. Il s'agit donc de la capacité physique du tableau de caractères.
- Un entier dénotant la longueur de la chaîne, *i.e.* le nombre de caractères significatifs qu'elle contient. Il s'agit de l'information renvoyée par l'application de **strlen** sur le tableau de caractères.

1.5.2 Mise en œuvre

Examinons chacun des membres de la forme canonique :

Constructeur par défaut : nous allons utiliser un constructeur qui crée une chaîne de longueur nulle mais susceptible de contenir 10 caractères.

Constructeur par recopie : il devra positionner correctement la capacité et la longueur mais également allouer de la mémoire pour le tableau de caractères et recopier celui de son modèle.

Destructeur : il sera chargé de rendre la mémoire du tableau de caractères

Opérateur d'affectation : après nettoyage de la mémoire, il devra allouer de la mémoire puis recopier la chaîne du modèle. Attention ! il ne faut surtout pas employer **strdup** car ce dernier utilise **malloc**, incompatible avec les mécanismes de **new** et **delete**.

1.5.3 Code résultant

Voici l'implémentation finale de notre classe chaîne. Bien entendu, la forme canonique ne fournit que le canevas de base nécessaire à une utilisation rationnelle de la mémoire qui ne comprend pas les méthodes nécessaires au bon fonctionnement de la chaîne.

```
#ifndef __Chaine_H__
#define __Chaine_H__

#include <string.h>

class Chaine
{
private:
    int    capacite_;
    int    longueur_;
    char *tab;
    enum {CAPACITE_DEFAULT=16}; // Capacité par défaut
                                   // et granularité d'allocation
public:
    // constructeur par défaut
    Chaine(int capacite=CAPACITE_DEFAULT) :
        capacite_(capacite),
        longueur_(0)
    {
        tab=new char[capacite_];
    }
};
```

```

    tab[0]=0;
}

// second constructeur
Chaine(char *ch)
{
    int lChaine=strlen(ch);
    int modulo=lChaine % CAPACITE_DEFAULT;

    if (modulo)
        capacite_=((lChaine / CAPACITE_DEFAULT)+1)*CAPACITE_DEFAULT;
    else
        capacite_=lChaine;
    longueur_=lChaine;
    tab=new char[capacite_];
    strcpy(tab,ch);
}

// constructeur par copie
Chaine(const Chaine &uneChaine) :
    capacite_(uneChaine.capacite_),
    longueur_(uneChaine.longueur_)
{
    tab=new char[capacite_];
    strcpy(tab, uneChaine.tab);
}

// accès aux membres
int capacite(void)
{
    return capacite_;
}

int longueur(void)
{
    return longueur_;
}

// destructeur
~Chaine(void)
{
    if (tab)
        delete [] tab; // ne jamais oublier les crochets lorsque
                        // l'on désalloue un tableau !
}

// Operateur d'affectation
Chaine & operator=(const Chaine& uneChaine)
{
    if (tab)
        delete [] tab;
    capacite_=uneChaine.capacite_;
    longueur_=uneChaine.longueur_;
    tab = new char[capacite_];
    strcpy(tab,uneChaine.tab);
    return *this;
}
};
#endif

```

Programme 1.3 Réalisation de la classe Chaine

1.6 Exercice résolu : différence entre construction et affectation

Enoncé :

Décrivez lignes de code suivantes (mettant en jeu les classes T et U) en séparant éventuellement les cas $T=U$ et $T \neq U$.

```
1) T t;  
2) U u;  
3) T z(t);  
4) T v();  
5) T y=t;  
6) z = t;  
7) T a(u);  
8) t = u;
```

Programme 1.4 Exercice sur la forme canonique de Coplien

Correction :

- T t** Construction d'un objet t de classe T avec appel du constructeur par défaut
- U u** Construction d'un objet u de classe U avec appel du constructeur par défaut
- T z(t)** Construction d'un objet z de classe T avec appel du constructeur de recopie sur l'objet t
- T v()** Déclaration d'une fonction v renvoyant un objet de classe T et ne prenant pas de paramètre.
- T y=t** Construction d'un d'objet y de classe T avec appel du constructeur de recopie sur l'objet t. Il ne peut en aucun cas s'agir d'une affectation car nous sommes dans une déclaration et l'objet y doit être construit ! En fait les lignes 3 et 5 sont équivalentes
- z = t** Affectation de l'objet t à l'objet z par l'opérateur d'affectation. Nous sommes bien ici dans le cadre d'une affectation : les objets t et z sont déjà construits et la ligne de code n'est pas une déclaration
- T a(u)** Dans le cas général ($T \neq U$), il s'agit de la construction d'un objet de classe T à partir d'un objet de classe U à l'aide d'un constructeur de la forme `T(const U&)`. Dans le cas dégénéré ($T=U$), il s'agit d'un appel au constructeur de recopie.
- t = u** Dans le cas général ($T \neq U$), il s'agit de l'affectation d'un objet de classe U à un objet de classe T à l'aide d'un opérateur de la forme `T& operator=(const U&)`. Dans le cas dégénéré ($T=U$), il s'agit d'une affectation à l'aide de l'opérateur d'affectation de la forme canonique.

1.7 Les constructeurs ne supportent pas le polymorphisme !

Bien que ce dernier propos soit détaché de la librairie standard du C++, je crois bon de rappeler que les constructeurs ne supportent pas le polymorphisme. En particulier, il est impossible d'appeler automatiquement depuis le constructeur de la classe de base une méthode polymorphique.

Considérons par exemple le code suivant :

```
#include <iostream.h>

class T1
{
public:
    virtual void afficher(void)
    {
        cout << "Classe T1" << endl;
    }
    T1(void)
    {
        afficher();
    }
    virtual ~T1()
    {
        cout << "On detruit T1" << endl << "Affichage ";
        afficher();
    }
};

class T2 : public T1
{
public:
    virtual void afficher(void)
    {
        cout << "Classe T2" << endl;
    }

    T2(void) : T1()
    {
    }

    virtual ~T2()
    {
        cout << "On detruit T2" << endl << "Affichage ";
        afficher();
    }
};
```

Programme 1.5 tentative d'appel polymorphique au sein d'un constructeur

Le but ici est d'appeler la méthode `afficher` spécifique à chaque classe à l'intérieur de son constructeur. De même, on réalise la même opération à l'intérieur du destructeur.

Voici la fonction **main** associée :

```
int main(int, char**)
{
```

```

    cout << "Avant la construction d'un objet de classe T1" << endl;
    T1 t1;
    cout << "Avant la construction d'un objet de classe T2" << endl;
    T2 t2;
    return 0;
}

```

Programme 1.6 fonction main de tentative d'appel polymorphique dans un constructeur

On s'attendrait donc à un résultat du style :

```

Avant la construction d'un objet de classe T1
Classe T1
Avant la construction d'un objet de classe T2
Classe T2
On detruit T2
Affichage Classe T2
On detruit T1
Affichage Classe T1
On detruit T1
Affichage Classe T1

```

Programme 1.7 Résultat chimérique obtenu par appel polymorphique dans un constructeur

Hors, le résultat obtenu est le suivant :

```

Avant la construction d'un objet de classe T1
Classe T1
Avant la construction d'un objet de classe T2
Classe T1
On detruit T2
Affichage Classe T2
On detruit T1
Affichage Classe T1
On detruit T1
Affichage Classe T1

```

Programme 1.8 Résultat (décevant®) réellement obtenu

Le verdict est simple : le constructeur de T1 appelle toujours la méthode `T1::afficher` et n'applique pas le polymorphisme alors que le destructeur qui lui, est une méthode d'instance virtuelle le fait sans problème !

Ceci s'explique en fait très simplement : lors de la construction d'un objet de classe T1 par son constructeur la table des méthodes virtuelles n'est pas encore affectée à l'objet et les différentes méthodes appelées le sont toujours par liaison statique. C'est à la sortie du constructeur que la table des méthodes virtuelles est opérationnelle.

Bien entendu, il s'agit là d'un exemple d'école mais bien des personnes tentaient d'appeler une méthode d'initialisation automatique polymorphique des attributs à des valeurs par défaut de cette manière. Je vous laisse augurer de leur déception !

2. De la surcharge des opérateurs et de la bonne utilisation des références en général

Ce chapitre traite plus spécialement de la surcharge des opérateurs mais plus généralement de la bonne utilisation des références en C++. En effet, la plupart des règles exposées ici sur les opérateurs seront applicables directement aux autres méthodes ou fonctions.

2.1 Buts

La surcharge des opérateurs standard est assurément l'une des fonctionnalités de C++ les plus appréciées. En effet, il est indubitable que cela permet d'écrire du code ayant un aspect très naturel.

Hors, leur écriture est très piégeuse. En effet, certains doivent être surchargés en tant que fonctions, d'autres en tant que méthodes, leur type de retour doit être soigneusement étudié etc.

Il n'est donc pas inutile d'y revenir quelque peu !

2.2 Choisir entre méthode et fonction

Comment choisir si l'on doit définir un opérateur surchargé en tant que méthode ou bien en tant que fonction externe à la classe ?

2.2.1 Un exemple particulier : les opérateurs de redirection

Dans certains cas, la question ne se pose même pas. Supposons, par exemple, que l'on souhaite fournir pour une classe T une version des opérateurs << et >> permettant respectivement d'écrire le contenu de T sur un flux en sortie (**ostream**) et de lire depuis un flux en entrée (**istream**) le contenu de T.

Ces opérations ne peuvent absolument pas être codées en tant que méthodes de la classe T. Considérons les résultats désastreux que cela pourrait avoir. La déclaration / définition de cet opérateur pourrait être :

```
class T
{
public:
    ostream &operator<<(ostream &a)
    {
        // code omis ...
        return a;
    }
};
```

Avec les déclarations :

```
T t;
```

Au moment de l'utiliser, l'opérateur << devrait s'écrire, sous forme suffixe :

```
t.operator<<(cout);
```

Soit, sous forme infixe :

```
T << cout;
```

Ce qui ne correspond absolument pas à l'utilisation habituelle de cet opérateur. Le seul moyen d'utiliser une méthode pour le surcharger avec sa syntaxe habituelle serait donc de l'inclure dans la classe **ostream**, ce qui est inconcevable. Il est donc nécessaire de le définir sous la forme d'une fonction externe, éventuellement amie de la classe T si elle doit accéder aux attributs protégés, soit :

```
ostream &operator<<(ostream &a, const T &t)
```

Bien que cela pose moins de problèmes, il sera également judicieux de déclarer sous forme d'une fonction externe l'opérateur de redirection depuis un flux.

2.2.2 Une règle bien utile

En règle générale, nous appliquerons la règle suivante pour déterminer si un opérateur doit être surchargé en tant que méthode ou en tant que fonction externe (éventuellement amie).

*Un opérateur ne sera déclaré en tant que méthode que dans les cas où **this** joue un rôle privilégié par rapport aux autres arguments*

Donnons deux exemples qui nous permettront de juger du bien fondé de cette règle fondée sur des critères conceptuels. En effet, on ne considérera qu'une méthode comme légitime si son impact sur le paramètre par défaut est suffisant.

2.2.2.1 Le cas de l'opérateur d'affectation

Soit le cas de l'opérateur d'affectation que nous écrivons comme une méthode :

```
T& T::operator=(const T&a)
```

Etudions les rôles respectifs de l'argument par défaut **this** et de l'argument explicite a.

- L'argument a n'est accédé qu'en lecture et l'opération ne modifie en rien son statut.
- En revanche, **this** est considérablement modifié car, *a priori*, tous ces attributs sont modifiés. En outre, c'est une référence sur l'objet ***this** qui est renvoyée par l'opérateur.

Au regard de ces deux considérations, il est évident que **this** joue un rôle privilégié par rapport au second argument, il est donc logique que ce soit une méthode de la classe T.

2.2.2.2 Le cas de l'opérateur d'addition

Equipons dorénavant notre classe T d'un opérateur dyadique quelconque, par exemple, l'addition. Si nous décidons d'en faire une méthode, son prototype sera :

```
T T::operator+(const T&a)
```

Comme pour le cas précédent, étudions les rôles respectifs de l'argument par défaut **this** et de l'argument explicite a. Dans une addition, aucun des deux membres n'est modifié et le résultat n'est ni le membre de gauche, ni celui de droite : en fait les deux arguments jouent un rôle absolument symétrique.

De ce fait, **this** n'a pas un rôle privilégié par rapport à a et l'opérateur est alors transformé en fonction externe, soit :

```
T ::operator+(const T&a, const T&b)
```

2.2.3 Récapitulatif

Le tableau suivant récapitule quels opérateurs seront décrits en tant que méthode ou fonction externe :

Affectation, affectation avec opération (=, +=, *=, etc.)	Méthode
Opérateur « fonction » ()	Méthode
Opérateur « indirection » *	Méthode
Opérateur « crochets » []	Méthode
Incrémentation ++, décrémentation --	Méthode
Opérateur « flèche » et « flèche appel » -> et ->*	Méthode
Opérateurs de décalage << et >>	Méthode
Opérateurs new et delete	Méthode
Opérateurs de lecture et écriture sur flux << et >>	Fonction
Opérateurs dyadiques genre « arithmétique » (+, -, / etc.)	Fonction

Tableau 2.1 Description des surcharge d'opérateurs

2.3 Quels opérateurs ne pas surcharger ?

Pour commencer, il faut savoir que certains opérateurs ne peuvent pas être redéfinis car le langage vous l'interdit ! Il s'agit de

.	Sélection d'un membre
---	-----------------------

. *	Appel d'un pointeur de méthode membre
::	Sélection de portée
:?	Opérateur ternaire

Tableau 2.2 Opérateurs du C++ ne pouvant être surchargés

Vous noterez au passage, que si la sélection d'un membre d'un objet statique par l'opérateur . ne peut être surchargée, en revanche la sélection d'un membre d'un objet dynamique par l'opérateur flèche -> peut, elle, l'être ! Cela permettra, par exemple de créer des classes de pointeurs intelligents.

En outre, même si vous en avez la possibilité, il vaut mieux ne pas surcharger les opérateurs suivants :

,	Evaluation séquentielle d'expressions
!	Non logique
&&	Ou et Et logiques

Tableau 2.3 Opérateurs C++ qu'il vaut mieux ne pas surcharger

En effet, il faut respecter une règle fondamentale :

Les opérateurs surchargés doivent conserver leur rôle naturel sous peine de perturber considérablement leur utilisateur

En effet, le but des opérateurs surchargés est de faciliter l'écriture de code. Autrement dit, l'utilisateur de vos opérateurs ne doit pas avoir à se poser de question lorsqu'il va les utiliser, et ce n'est pas en modifiant le comportement des plus élémentaires que vous atteindrez ce but.

Par exemple, l'opérateur d'évaluation séquentielle est très bien comme il est. Et, à moins de vouloir régler son compte à un ennemi, vous n'avez aucun intérêt à le surcharger. La même règle s'applique directement au non logique.

Pour ce qui concerne les opérateurs logiques dyadiques, il existe une autre règle encore plus subtile. En effet, en C++ comme en C, les évaluations sont dites à court circuit. En effet, si le premier membre d'un « et » est faux, il est inutile d'évaluer le reste de l'expression, le résultat sera faux quoiqu'il arrive. Réciproquement, l'évaluation d'une expression « ou » s'arrête dès que l'on rencontre un membre vrai.

Considérez maintenant la redéfinition des opérateurs || et &&. Leurs cas étant symétriques, nous n'allons étudier que le « ou ». Il s'écrira ainsi :

```
bool operator||(const bool a, const bool b)
```

(Notez au passage l'utilisation des valeurs directes des booléens et non pas de références : `sizeof(bool) < sizeof(@) !`)

... et souvenez vous ! les arguments d'une fonction sont toujours évalués *in extenso* avant l'appel de la fonction ! ainsi, il ne peut plus y avoir de court circuit et une expression telle que, par exemple :

```
if (p && *p>0)
```

qui pouvait être écrite sans danger en C++ ne peut plus l'être du fait de votre surcharge (la déréréférence du pointeur sera faite même si ce dernier est nul, première clause du `&&`).

2.4 Quels paramètres doivent prendre les opérateurs

Typiquement, les opérateurs doivent toujours prendre les paramètres susceptibles d'être modifiés par référence.

Les paramètres qui ne doivent pas être modifiés sont soit :

- Passés par référence constante
- Passés par valeur s'il s'agit de scalaires dont la taille est inférieure à celle d'un pointeur, en aucun cas, on ne passera un objet par valeur

2.4.1 Préférer les références constantes aux objets

Il faut toujours préférer une référence constante à un objet passé par valeur. Comme vous n'êtes pas obligés de me croire sur parole, je vais tacher de vous le démontrer en identifiant les points positifs et négatifs de la technique

2.4.1.1 Point négatif :

Bien entendu, si la fonction ou la méthode prenant une référence (fût-ce t'elle constante) est amie de la classe de l'objet passé par référence constante, vous ne pourrez empêcher un programmeur sale de modifier directement les attributs de ce dernier. Le seul moyen d'empêcher de tels effets de bord consiste à limiter au maximum l'utilisation des amis.

2.4.1.2 Points positifs :

Efficacité de l'appel : rappelons en effet que lorsqu'un objet est passé par valeur, il y a appel du constructeur de recopie pour générer une copie, qui elle est utilisée par la méthode / fonction appelée. Hors, une référence est fondamentalement un pointeur, et il sera plus rapide d'empiler un simple pointeur que d'appeler un constructeur de recopie aussi optimisé soit il.

Support du polymorphisme : supposons que vous créiez la classe `Evaluable`, dotée de la méthode `evaluation`, comme dans l'extrait de code suivant (les partisans des véritables langages à objets argueront, à raison, que cette classe aurait du être une interface !)

```

class Evaluable
{
protected:
    int (*f)(void);
    // détails omis
public:
    // Evaluation simple : utilisation de la fonction objectif
    virtual int evaluation(void) const
    {
        return (*f)();
    }
};

```

Programme 2.1 la classe Evaluable

La classe Evaluable est disposée à être dérivée et sa méthode Evaluation redéfinie afin de modifier le format de l'évaluation. Le but ultime est d'obtenir un comportement polymorphique sur la méthode Evaluation. Vous construisez maintenant un opérateur de comparaison de la manière suivante :

```

bool operator<(Evaluable gauche, Evaluable droite)
{
    return (gauche.evaluation() < droite.evaluation());
}

```

Programme 2.2 un opérateur de comparaison générique ?

Vous espérez ainsi pouvoir l'utiliser avec toutes les classes dérivées de Evaluable par polymorphisme sur la méthode Evaluation.

Par exemple, soit la classe Penalites, définie de la manière suivante :

```

class Penalites : public Evaluable
{
private:
    int facteurPenalites;
    int violation;
public:
    // Evaluation avec penalites quadratiques !
    virtual int evaluation(void) const
    {
        return (*f)() + facteurPenalites*violation*violation;
    }
};

```

Programme 2.3 la classe Penalites

Soit maintenant le code suivant :

```

int main(int, char**)
{
    Evaluable e1,e2;
    Penalites p1,p2;

    if (e1 < e2) // utilise Evaluable::evaluation
        // code omis
}

```

```
if (p1 < p2) // utilise egalement Evaluable::evaluation !!!!  
    // code omis  
}
```

Programme 2.4 Utilisation de l'opérateur <

L'on s'attendrait pour le moins à ce que la comparaison entre p1 et p2, objets de la classe Penalites utilise la méthode Penalites::evaluation. Hors, il n'en est rien ! du fait du passage par valeur, les objets p1 et p2 sont dégradés de Penalites vers Evaluable et le polymorphisme est perdu. Afin de retrouver le comportement souhaité, il suffit de passer les arguments par référence :

```
bool operator<(const Evaluable &gauche, const Evaluable &droite)  
{  
    return (gauche.evaluation() < droite.evaluation());  
}
```

Programme 2.5 un opérateur de comparaison générique !

Grâce aux références, l'on retrouve bien le comportement polymorphique tant attendu et le code du Programme 2.4 fonctionne comme prévu !

2.4.1.3 Conclusion

Au regard de ces explications, il est clair qu'il vaut toujours mieux utiliser une référence constant qu'un objet passé par valeur. Aussi, usez et abusez des paramètres passés par références constantes !

2.4.2 Pourquoi ne pas toujours renvoyer une référence

Après avoir traité le cas des paramètres positionnels des méthodes, étudions dorénavant le cas d'un paramètre particulièrement spécial : le retour de méthode / fonction.

Nous y apprendrons que s'il est quasi tout le temps recommandé d'utiliser des références constantes pour les paramètres, il faut le plus souvent retourner un objet.

2.5 Que doivent renvoyer les opérateurs ?

Il faut séparer deux grands cas dans les types de retour :

Les types de retour scalaires ne posent aucun problème particulier : le résultat est logé sur la pile ou dans un registre du processeur avant d'être récupéré par l'appelant.

Les types de retour objet sont plus complexes à gérer car l'on se retrouve souvent face à un dilemme : doit on retourner un objet ou une référence ?

A la question :

Quand peut on (et doit on !) absolument renvoyer une référence ?

Nous pourrions répondre :

« Il est nécessaire de renvoyer une référence lorsque l'opérateur (ou une méthode quelconque) doit renvoyer l'objet cible (i.e. ***this**) de manière à ce que celui-ci soit modifiable. C'est typiquement le cas des opérateurs dont le résultat *lvalue*, c'est à dire une expression que l'on peut positionner à gauche d'un signe d'affectation. Dans cette catégorie, on retrouve notamment tous les opérateurs d'affectation et assimilés ainsi que l'opérateur de préincrément. »

Dans tous les autres cas, il vaudra mieux retourner un objet.

Afin d'étayer notre propos, considérons une classe modélisant les nombres rationnels et déclarée comme suit :

```
class Rationnel
{
private:
    int numérateur;
    int denominateur;
public:
    Rationnel (int numVal, int denVal) :
        numérateur(numVal), denominateur(denVal)
    {
    }
    // déclarations omises
    int num() const
    {
        return numérateur;
    }
    int deno() const
    {
        return denominateur;
    }
};
```

Programme 2.6 Déclaration d'une classe Rationnel

Ecrivons dorénavant un opérateur de multiplication sur les rationnels, la première idée de codage est la suivante :

```
Rationnel operator*(const Rationnel &gauche, const Rationnel &droite)
{
    return Rationnel(gauche.num()*droite.num(),
                     gauche.deno()*droite.deno());
}
```

Programme 2.7 Opérateur de multiplication sur les rationnels

Ecrit tel quel, cet opérateur renvoie un objet Rationnel. Examinons maintenant la ligne de code suivante (qui suppose l'existence préalable de deux objets Rationnel a et b) :

```
Rationnel c(0,0);
// code omis
c=a*b;
```

Typiquement, voici le comportement de cette instruction telle que spécifié dans la documentation officielle du C++.

1. Appel de la fonction `operator*` avec passage par référence constantes des objets `a` et `b`.
2. Construction de l'objet temporaire résultat sur la pile dans l'espace automatique de **`operator*`** par le constructeur `Rationnel(int, int)`.
3. Instruction **`return`**. L'objet temporaire créé précédemment (par le constructeur `Rationnel(int, int)`) est recopié dans un espace temporaire, sur la pile, mais en dehors de l'espace automatique de **`operator*`** afin que ce nouvel objet ne soit pas détruit lors du retour de la fonction. Nous appellerons cet objet `retour_temp`.

Notez que dans ce cas, où la classe rationnelle n'a que des attributs scalaires, le constructeur par recopie automatiquement fourni par le compilateur (vous savez, celui qui fonctionne par recopie bit à bit « optimisée ») est parfait.

4. Sortie de la fonction **`operator*`**. L'objet temporaire `Rationnel(int, int)` construit par est détruit au contraire de l'objet `retour_temp` placé en dehors de la zone automatique de **`operator*`**.
5. L'opérateur d'affectation sur la classe `Rationnel` (une fois encore, l'opérateur automatiquement créé par le compilateur est suffisant) est appelé sur l'objet `c` avec pour paramètre `retour_temp`.
6. L'objet `retour_temp` devenu inutile est détruit.

Ce comportement peut paraître lourd mais il s'avère particulièrement sûr. En effet, il évite, par exemple, que l'objet temporaire créé par le constructeur `Rationnel(int, int)` ne soit détruit avant son utilisation à l'extérieur de la fonction.

En outre, certains compilateurs sont capables de réaliser de « l'optimisation de **`return`** ». Plutôt que d'utiliser un constructeur par recopie pour fabriquer l'objet `retour_temp`, en dehors de la zone de pile détruite au moment du retour, le compilateur créé directement l'objet `Rationnel(int, int)` dans cet emplacement privilégié s'épargnant ainsi un appel au constructeur de recopie.

Toutefois, le programmeur soucieux de performance est en mesure de se poser la question :

Si les références sont si efficaces pour passer les paramètres, pourquoi n'utiliserais-je pas une référence comme type de retour ?

L'opérateur de multiplication pourrait alors s'écrire :

```
Rationnel &operator*(const Rationnel &gauche, const Rationnel &droite)
{
    return Rationnel(gauche.num()*droite.num(),
```

```
} gauche.deno()*droite.deno());
```

Programme 2.8 opérateur de multiplication 2^{ème} forme (erronée)

Vous remarquez que l'on renvoie une référence sur un objet automatique, ce qui n'est pas sans rappeler une erreur fréquente des programmeurs en C qui renvoient l'adresse d'une variable automatique : les conséquences sont exactement les mêmes : l'affectation suivante va se faire sur un objet qui n'existe plus.

Qu'à cela ne tienne me direz vous, je n'ai qu'à construire mon objet sur le tas, on obtient alors :

```
Rationnel &operator*(const Rationnel &gauche, const Rationnel &droite)
{
    return *(new Rationnel(gauche.num()*droite.num(),
                           gauche.deno()*droite.deno()));
}
```

Programme 2.9 opérateur de multiplication 3^{ème} forme (erronée)

Le problème précédent est en effet éliminé : la référence renvoyée n'est plus sur un objet temporaire. Toutefois, qui va se charger d'appeler **delete** sur l'objet créé dans l'opérateur pour rendre la mémoire ?

Dans le cas où l'on réalise une opération simple, ce n'est pas trop grave, on pourra toujours avoir :

```
c=a*b;
delete &c;
```

C'est très contraignant (et source rapide d'erreurs ...) mais ça marche ! En revanche considérez l'expression :

```
d=a*b*c;
```

Celle-ci se réécrit en fait :

```
d=operator*(a, operator*(b*c));
```

Notez qu'il y a construction d'un objet muet sans identificateur que vous n'avez absolument pas la possibilité de détruire. Il y a donc une réelle perte de mémoire.

Alors, quelle solution choisir ? il vaut mieux jouer la sécurité et adopter la première solution : celle qui renvoie des objets. Il est vrai que l'on risque d'utiliser un opérateur de copie supplémentaire mais il vaut mieux perdre un peu de temps que risquer de dilapider la mémoire ou travailler sur des objets en cours de destruction, ne croyez vous pas ?

2.6 Le cas particulier des opérateurs d'incrément

Nous discutons ici des opérateurs d'incrément au sens large, tout ce discours s'appliquant également aux opérateurs de décrémentation.

Les opérateurs d'incrémentation sont particuliers à bien des aspects. En particulier ils peuvent être utilisés en pré incrémentation (`++i`) ou en post incrémentation (`i++`). La première difficulté va donc être de séparer ces deux formes. Le C++ emploie pour cela une astuce assez sale. En effet, l'opérateur de pré incrémentation est déclaré sans paramètre, alors que l'opérateur de post incrémentation prend un paramètre **int** muet. En outre, reste le problème de leur type de retour.

Une fois de plus, nous nous basons sur le comportement de ces opérateurs sur le type canonique **int**.

Par exemple, l'expression

```
++i=5;
```

est tout à fait valide, au contraire de :

```
i++=5;
```

En effet, dans le second cas, il y a conflit dans l'ordre des opérations entre l'affectation et la post incrémentation. Pour résumer, il nous suffit de dire que `++X` est une *lvalue*, au contraire de `X++`. Ce qui nous indique clairement que l'opérateur de pré incrémentation doit renvoyer une référence sur ***this** alors que l'opérateur de post incrémentation doit renvoyer un objet constant.

Finalement, nous obtenons les déclarations suivantes pour une classe X quelconque :

```
class X
{
    // code omis
public:
    X & operator++(void); // pré incrémentation
    const X operator++(int); // post incrémentation
};
```

Programme 2.10 forme canonique des opérateurs de pré et post incrémentation

D'autre part, reprenons un manuel (un bon, de préférence ☺) de génie logiciel. Vous y trouverez probablement, sous quelque forme que ce soit, l'adage suivant :

Toute duplication de code est à proscrire

Nous allons l'appliquer à la rédaction de nos opérateurs d'incrémentation. En effet, il est évident de remarquer que l'opérateur de post incrémentation peut *toujours* s'écrire de la manière suivante une fois l'opérateur de pré incrémentation écrit :

```
const X X::operator++(int)
{
    const X temp=const_cast<const X>(*this); // copie de l'objet dans
                                              // son état actuel
    operator++; // pre incrementation
}
```

```
    return temp;
}
```

Programme 2.11 écriture de l'opérateur de post incrémentation

En effet, il suffit d'écrire une fois le code d'incrémentatation et ce dans l'opérateur de pré incrémentation, lequel renvoie toujours ***this**. Le code de l'opérateur de post incrémentation est alors trivial. On assiste alors au gigantesque gâchis réalisé par l'opérateur de post incrémentation :

1. Construction d'un objet temporaire par le constructeur de copie, lequel sera renvoyé par la méthode
2. Appel à l'opérateur de pré incrémentation pour réaliser effectivement l'incrémentatation
3. Renvoi de l'objet créé précédemment, avec, si le compilateur ne fait pas d'optimisation de retour, appel au constructeur de copie.

On voit bien que la post incrémentation est une opération beaucoup plus onéreuse que la pré incrémentation. Aussi, dès lors que l'on pourra faire soit l'une, soit l'autre, il faudra toujours utiliser la pré incrémentation.

2.7 Le cas particulier des opérateurs new et delete

Les opérateurs **new** et **delete** sont particulièrement spéciaux car ils gèrent l'allocation et la désallocation des objets. Avant de se poser la question de leur surcharge, nous allons d'abord nous intéresser à leur fonctionnement détaillé.

2.7.1 Bien comprendre le fonctionnement de new et delete

Pour plagier Scott Meyers (à qui ce poly doit beaucoup !, merci Scott pour tout le boulot réalisé sur le C++), il faut bien différencier deux notions fondamentales.

- L'opérateur **new**
- **operator new(size_t taille)**

Lorsque vous faites l'instruction :

```
X *x=new X(...)
```

Voici ce qui se passe :

1. L'opérateur **new** appelle **X::operator new(sizeof(X))** pour allouer la mémoire nécessaire à un objet de type X
2. L'opérateur **new** appelle ensuite le constructeur spécifié sur le bloc de mémoire renvoyé par **X::operator new**.

Ce comportement est immuable est vous ne pouvez absolument pas le changer car il est codifié dans la norme du C++. Ce que vous pouvez modifier, c'est le comportement de **operator new**. En effet, celui-ci agit comme **malloc**.

Point particulièrement intéressant, il existe un **operator new** par classe, lequel est affecté par défaut à **::operator new**, opérateur d'allocation de mémoire global. Vous pouvez également surcharger l'opérateur d'allocation de mémoire global, mais vous avez intérêt à savoir exactement ce que vous fabriquez car cet opérateur est utilisé pour allouer la mémoire de l'application elle même. Aussi, toute « boulette » se soldera immédiatement par une erreur système qui pourrait se payer au prix fort.

Le même raisonnement s'applique à **delete**. Il faut alors distinguer :

- L'opérateur **delete**, qui appelle le destructeur puis **operator delete**.
- **operator delete** qui rend la mémoire au système

2.7.2 Une version spéciale de new : operator new de placement

Pour chaque classe, il existe une version extrêmement simple de **operator new**. C'est ce que l'on appelle **operator new** et qui répond à la définition suivante :

```
X *X::operator new(void *a)
{
    return a;
}
```

Programme 2.12 L'operator new de placement

Comme vous pouvez le constater, cet opérateur est extrêmement simple : il ne fait que renvoyer la valeur qui lui est passée en paramètre. Il est néanmoins très utile lorsque l'on a réservé à l'avance de la mémoire et que l'on souhaite appliquer les constructeurs dans un second temps.

Considérez par exemple le programme suivant qui alloue des objets dans un tableau de caractères prévu à l'avance. Cette technique peut être utile dans un environnement où les appels à **new** sont indésirables (surcoût important, faible fiabilité, etc.)

2.7.3 Exercice : un exemple simpliste de surcharge de new et delete

Concevez un exemple d'allocation et de désallocation de mémoire dans un grand tableau alloué en tant que tableau de **char**. Il n'est pas demandé de gérer la fragmentation. En fait, il faut surcharger **new** et **delete** de manière à leur conférer par défaut le comportement de l'exemple précédent.

3. La gestion des exceptions

Les exceptions ont été rajoutées à la norme du C++ afin de faciliter la mise en œuvre de code robuste.

3.1 Schéma général d'interception des exceptions

Le schéma général s'appuie sur la description de blocs protégés suivis de code de prise en charge des exceptions appelés traite exception ou gestionnaire d'exception.

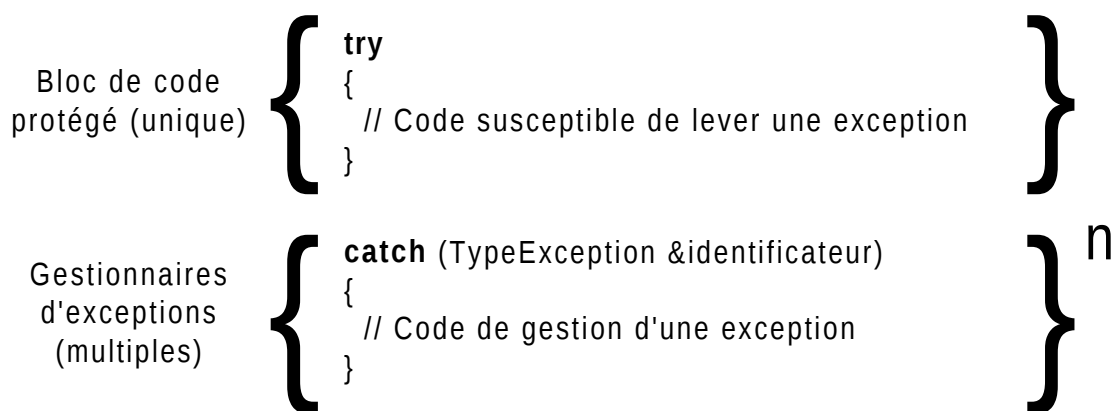


Figure 3.1 Schéma général de gestion des exceptions en C++

Il est important de faire d'ores et déjà quelques commentaires importants :

- Un gestionnaire utilise toujours une référence sur un type d'exception. Lorsque l'exception correspondante est levée, il se sert de l'identificateur pour accéder aux données membres ou aux méthodes de l'exception.
- Il existe un gestionnaire universel qui est introduit par la séquence `catch (...)`. Il convient toutefois de ne l'utiliser que dans des circonstances limitées c'est à dire lorsque vous maîtrisez l'ensemble des exceptions qui risquent d'être levées par le bloc de code incriminé. En effet, ce gestionnaire intercepte n'importe quelle exception.

Hors certains environnements de programmation encapsulent toute la fonction `main` à l'intérieur d'un bloc **try** et fournissent des gestionnaires spécifiques à certaines exceptions. Si vous utilisez **catch (...)** à l'intérieur d'un tel bloc, vous risquez de court-circuiter les mécanismes de gestion de telles interruptions avec des conséquences parfois dramatiques.

Aussi, il vaut mieux, comme nous le verrons plus loin, utiliser des hiérarchies bien pensées de vos exceptions.

3.2 La logique de traitement des exceptions

Lorsqu'une exception est levée dans votre programme, que se passe-t-il ?

- Si l'instruction en faute n'est incluse dans un bloc **try**, il y a appel immédiat de la fonction **terminate**.

Comme son nom l'indique, cette dernière provoque la terminaison immédiate du programme par l'appel à la fonction **abort** (l'équivalent d'un SIG_KILL sous Unix) sans aucun appel de destructeur ou de code de nettoyage ; ce qui s'avère la plupart du temps désastreux.

Notons également que **terminate** est appelée par défaut par la fonction **unexpected**.

Pour terminer par une note optimiste, il est possible de modifier le fonctionnement de **terminate** en utilisant la fonction **set_terminate**.

- En revanche, si l'instruction incriminée est incluse dans un bloc **try**, le programme saute directement vers les gestionnaires d'exception qu'il examine séquentiellement dans l'ordre du code source.
 - ✧ Si l'un des gestionnaires correspond au type de l'exception, il est exécuté, et, s'il ne provoque pas lui-même d'interruption ou ne met fin à l'exécution du programme, l'exécution se poursuit à la première ligne de code suivant l'ensemble des gestionnaires d'interruption. En aucun cas il n'est possible de poursuivre l'exécution à la suite de la ligne de code fautive.
 - ✧ Si aucun gestionnaire ne correspond au type de l'exception, celle-ci est propagée au niveau supérieur de traitement des exceptions (ce qui signifie le bloc **try** de niveau supérieur en cas de blocs **try** imbriqués) jusqu'à arriver au programme principal qui lui appellera **terminate**.

3.3 Relancer une exception

Dans certains cas, il est impossible de traiter localement une exception. Toutefois, le cas n'est pas jugé suffisamment grave pour justifier l'arrêt du programme. Le comportement standard consiste alors à relever l'exception incriminée (après, par exemple, un avertissement à l'utilisateur) dans l'espoir qu'un bloc de niveau supérieur saura la traiter convenablement. Ceci se fait en utilisant l'instruction **throw** isolée comme le montre l'exemple suivant :

```
try
{
}
catch (TypeException &e)
{
    // code de traitement local
    throw; // Relance l'exception
}
```

Programme 3.1 Relancer une exception

3.4 Utiliser ses propres exceptions

3.4.1 Déclarer ses exceptions

Selon la norme, les exceptions peuvent être de n'importe quel type (y compris un simple entier). Toutefois, il est pratique de les définir en tant que classes. Afin de ne pas polluer l'espace global de nommage, il est possible d'encapsuler les exceptions à l'intérieur des classes qui les utilisent.

En outre, et sans vouloir empiéter sur le cours concernant la librairie standard du C++ (STL), il me semble adéquat de dériver toute exception du type **std::exception** proposé en standard par tous les compilateurs modernes ☺.

En effet, cette classe propose une méthode nommée **what** qui peut s'avérer des plus précieuses. Examinons le code suivant :

```
class exception
{
    // code omis
    virtual const char * what () const throw ()
    {
        return << pointeur vers une chaîne quelconque >> ;
    }
};
```

Programme 3.2 Déclaration de la classe std::exception

De toute évidence, la méthode **what** est destinée à être surchargée par chaque classe dérivée de **std::exception** afin de fournir à l'utilisateur un message le renseignant sur la nature de l'erreur levée.

En outre, cela permet de récupérer votre exception avec une clause :

```
catch (std::exception &e)
```

plutôt qu'en utilisant l'infâme :

```
catch(...)
```

Considérons par exemple une classe nommée TasMax et implémentant les fonctionnalités d'un Tas max (on s'en doutait un peu ☺). Les erreurs peuvent être multiples :

- Erreurs d'allocation
- Erreurs d'accès aux éléments, lesquelles seront principalement :
 - ✧ Tentative de dépilage d'un élément sur un tas vide
 - ✧ Tentative d'empilage d'un élément sur un tas déjà plein

Afin de se simplifier l'existence, il est agréable d'utiliser une classe mère de toutes les classes d'exception que l'on peut lancer. La hiérarchie devient alors :

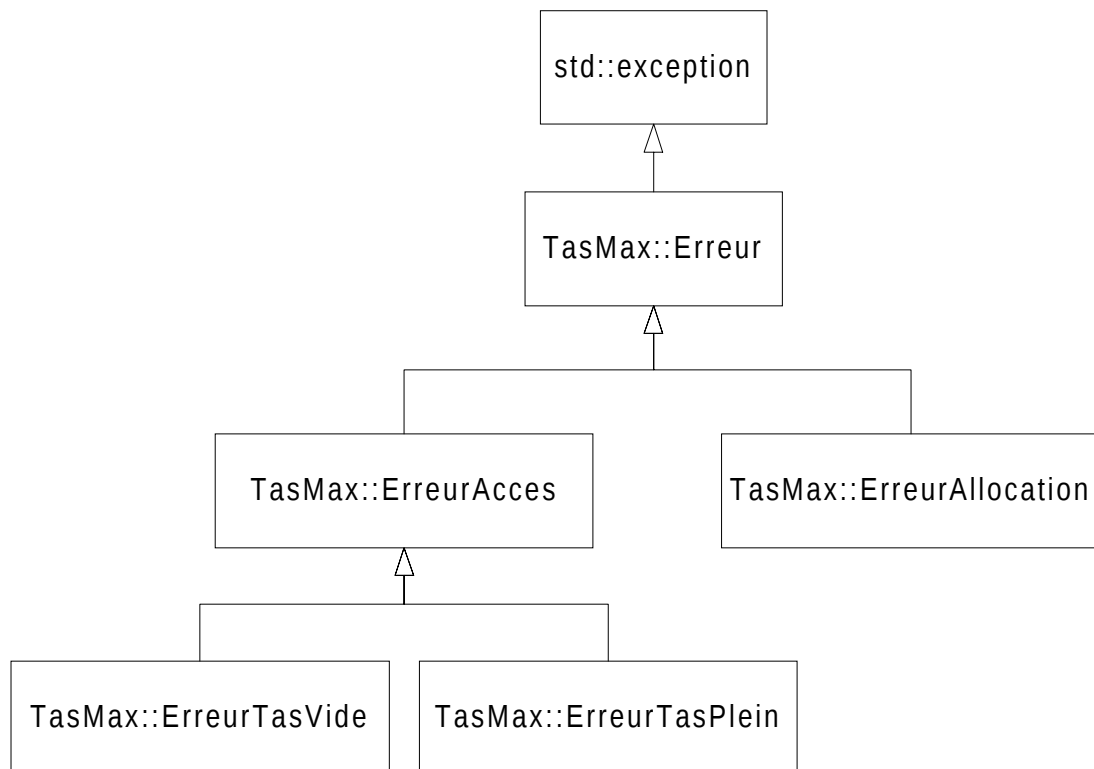


Figure 3.2 Hiérarchie d'exceptions du TasMax

Le code associé peut être :

```
class TasMax
{
public:
    class Erreur : public std::exception
    {
        const char *what(void) const throw()
        {
            return "Exception generale sur un tas max";
        }
    };

    class ErreurAcces : public TasMax::Erreur
    {
        const char *what(void) const throw()
        {
            return "Exception d'accès sur un tas max";
        }
    };

    class ErreurTasVide : public TasMax::ErreurAcces
    {
        const char *what(void) const throw()
        {
            return "Exception de depilement d'un tas max vide";
        }
    };

    class ErreurTasPlein : public TasMax::ErreurAcces
```

```

{
    const char *what(void) const throw()
    {
        return "Exception d'empilement sur un tas max plein";
    }
};

class ErreurAllocation : public TasMax::Erreur
{
    const char *what(void) const throw()
    {
        return "Impossible d'allouer de la memoire dans le tas max";
    }
};

// reste du code omis
};

```

Programme 3.3 Exemple d'une hiérarchie d'exceptions

Vous remarquerez au passage que ce code utilise des noms complètement qualifiés pour dériver les classes imbriquées. Ceci n'est pas requis par la norme du C++ mais permet parfois de lever certaines ambiguïtés, en particulier lorsque les identificateurs sont très simples.

3.4.2 Utiliser ses exceptions

Il convient de bien utiliser les gestionnaires d'exception, en particulier l'instruction :

```
catch (TypeException &e)
```

intercepte non seulement les exceptions de type **TypeException**, mais également de toutes les classes dérivées de **TypeException**. Aussi, il est important de toujours spécifier les gestionnaires des classes dérivées avant ceux des classes générales. Par exemple, si vous voulez intercepter toutes les erreurs dérivant de `TasMax::Erreur` mais plus spécifiquement `TasMax::ErreurTasVide` vous écririez :

```

try
{
    // bloc de code protégé
}
catch (TasMax::ErreurTasVide &e)
{
    // code spécifique à cette exception
}
catch (TasMax::Erreur &e)
{
    cerr << "Le Tas Max a provoqué l'exception : " << e.what() << endl;
    cerr << "Fin du programme" << endl;
    exit(1);
}
catch (std::exception &e)
{
    cerr << "Exception inconnue : " << e.what() << endl;
    cerr << "Fin du programme" << endl;
}

```

```
    exit(1);  
}
```

Programme 3.4 Exemple de gestion des erreurs en provenance du Tas Max

Vous noterez au passage l'utilisation de l'identificateur `e` permettant d'appeler les méthodes des exceptions.

Si vous aviez placé le gestionnaire de `TasMax::Erreur` avant celui de `TasMax::ErreurTasVide`, ce dernier n'aurait jamais été invoqué car `TasMax::Erreur` et une super classe de `TasMax::ErreurTasVide` et aurait donc intercepté l'exception.

3.5 Les spécificateurs d'exception

Lecteur attentif, vous n'aurez pas manqué de vous interroger sur la signification de la clause `throw()` à la fin du prototype de la méthode **what** de la classe **std::exception** (voir Programme 3.2).

C'est ce que l'on appelle un spécificateur d'exception et renseigne l'utilisateur sur le type des exceptions que peut renvoyer une méthode. Ceci paraît très attractif, on sait immédiatement à quel type d'exceptions l'on doit s'attendre au moment de les intercepter, et, le compilateur vous empêche de lancer une exception non prévue par votre spécificateur. Par exemple, le code suivant est interdit :

```
void TasMax::depiler(void) throw (TasMax::ErreurTasVide)  
{  
    ...  
    throw TasMax::Erreur(); // exception non prévue dans la spécification  
}
```

Programme 3.5 Tentative de lancement d'une exception non prévue

Mais, car il y a un **mais**, le spécificateur d'exception interdit aux autres méthodes ou fonctions appelées d'invoquer des exceptions non prévues. Hors, ce point est difficile à vérifier lors de la compilation. Plus grave, à l'exécution, si une exception non prévue par le spécificateur est lancée, la norme prévoit que la fonction **unexpected** soit invoquée, laquelle, appelle immédiatement **terminate** avec les conséquences que nous avons déjà vues précédemment. Il est toutefois possible de modifier ce comportement en modifiant **unexpected** grâce à la fonction **set_unexpected**.

Aussi, les spécificateurs d'exception doivent ils être réservés au code que vous maîtrisez totalement et plus spécifiquement aux méthodes pour lesquelles vous êtes en mesure de prévoir le déroulement complet. Par exemple, dans le cas du Tas Max, vous pourriez écrire :

```
TasMax(int taille=10) throw (TasMax::ErreurAllocation);  
void empiler(int valeur) throw (TasMax::ErreurTasPlein);  
int meilleurElement(void) throw (TasMax::ErreurTasVide);  
void depiler(void) throw (TasMax::ErreurTasVide);
```

Programme 3.6 Spécificateurs d'exceptions pour les méthodes du TasMax

4. Les transtypages en C++

Au passage du C au C++, les opérateurs de transtypage (ou conversion de type) ont subi un véritable lifting, et pas toujours pour le bien du programmeur !

En effet, si la nouvelle syntaxe de l'opérateur traditionnel et les nouveaux opérateurs de transtypage explicites sont une vraie bénédiction, les opérateurs de conversion implicites sont démoniaques !

Examinons une par une ces nouvelles fonctionnalités ...

4.1 Nouvelle syntaxe de l'opérateur traditionnel

Mettez vite Procol Harum sur la platine 33 tours de vos parents, coiffez vous d'une perruque hippie, faites passer un ~~Censuré~~ dans l'assistance et souvenez vous de l'opérateur de conversion (**cast**) du langage C que nous aimons tous tellement. Sa syntaxe est la suivante :

(nouveau type)(expression à transtyper)

où les parenthèses autour de l'expression à transtyper peuvent être omises si celle-ci est très simple et si vous êtes pris de pulsions suicidaires. La nouvelle syntaxe est la suivante :

type(expression à transtyper)

C'est beaucoup plus simple et en outre, à l'avantage de vous obliger à utiliser **typedef** pour les types pointeur.

En effet, considérez le transtypage :

```
Y *y;  
X *x=(X *)y
```

Pour l'écrire avec la nouvelle syntaxe, il vous faut passer par un **typedef** :

```
typedef X* PX;  
Y *y;  
X *x=PX(y)
```

Bien que simplificatrice, la nouvelle syntaxe n'était pas satisfaisante à bien des points de vue car elle ne fait que simplifier l'écriture des transtypages et ne corrige aucun des défauts de l'opérateur traditionnel (par ordre croissant de sévérité) :

- Elle est dure à retrouver dans un code source, car ce n'est ni plus ni moins qu'un nom de type et une paire de parenthèses.
- Elle permet de mixer dangereusement les objets constants et non constants
- Elle permet de réaliser n'importe quel type de promotion, ou transtypage de pointeur descendant ou encore *downcast*.

Rappelons rapidement ce qu'est un *downcast*. Il s'agit d'une conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée. Si l'opération est légitime, cela ne posera pas de problème spécifique. En revanche, si l'opération est illégitime, vous essayerez probablement d'accéder à des informations inexistantes avec tous les problèmes des mémoire conséquents.

Les nouveaux opérateurs vont permettre de résoudre ces problèmes.

4.2 Nouveaux opérateurs de transtypage

4.2.1 Pourquoi ?

Ces nouveaux opérateurs sont destinés à résoudre les problèmes soulevés par l'opérateur traditionnel hérité du langage C. Le tableau suivant indique le nom des nouveaux opérateurs ainsi que leur principale fonctionnalité. Notez bien que leur syntaxe commune permet de résoudre le problème de la recherche des transtypes dans le code source

static_cast	Opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère const ou volatile .
const_cast	Opérateur spécialisé et limité au traitement des caractères const et volatile
dynamic_cast	Opérateur spécialisé et limité au traitement des <i>downcast</i> .
reinterpret_cast	Opérateur spécialisé dans le traitement des conversions de pointeurs peu portables. Ne sera pas abordé ici

4.2.2 Syntaxe

La syntaxe, si elle est un peu lourde, a l'avantage de ne pas être ambiguë :

*op_***cast**<expression type>(expression à transtyper)

Où, vous l'aurez compris, *op* prend l'une des valeurs (**static**, **const**, **dynamic** ou **reinterpret**).

Cette syntaxe a l'avantage de bien séparer le type que l'on veut obtenir, qui est placé entre signes < et >, de l'expression à transtyper, placée, elle, entre parenthèses.

Par exemple, pour transformer un **double** en **int**, on aurait la syntaxe suivante :

```
int    i;
double d;
i=static_cast<int>(d)
```

Programme 4.1 exemple minimaliste de static_cast

Nous allons maintenant étudier, un par un, les trois principaux nouveaux opérateurs de transtypage, seul **reinterpret_cast** dont l'usage est encore entaché d'ambiguïté sera passé sous silence.

4.2.3 Fonctionnalités

4.2.3.1 L'opérateur static_cast

C'est l'opérateur de transtypage à tout faire qui remplace dans la plupart des cas l'opérateur hérité du C. Toutefois il est limité dans les cas suivants :

- Il ne peut convertir un type constant en type non constant
- Il ne peut pas effectuer de promotion (*downcast*)

4.2.3.2 L'opérateur const_cast

Il est spécialisé dans l'ajout ou le retrait des modificateurs **const** et **volatile**. En revanche, il ne peut pas changer le type de données de l'expression. Ainsi, les seules modifications qu'il peut faire sont les suivantes :

X	→	const X
const X	→	X
X	→	volatile X
volatile X	→	X
const X	→	volatile X
volatile X	→	const X

Tableau 4.1 Transtypages réalisables avec `const_cast`

Le code suivant illustre l'intérêt de **const_cast**, le résultat de la compilation (avec **gcc**) est placé en commentaire.

```

X t1(...);
const X t2(...);
X *t11=&t1;
const X *t22=&t2;

t11=&t2; // warning: assignment to `X *' from `const X *' discards const
t11=static_cast<X *>(&t2); // error: static_cast from `const X *' to `X *'
t11=const_cast<X *>(&t2); // Ok !

```

Programme 4.2 exemple d'utilisation de `const_cast`

4.2.3.3 L'opérateur `dynamic_cast`

Les *downcast* posent un problème particulier car leur vérification n'est possible qu'à l'exécution. Aussi, contrairement à **`static_cast`** et **`const_cast`** qui ne sont que des informations à l'adresse du compilateur, **`dynamic_cast`** génère du code de vérification.

Supposons que vous disposez d'un conteneur agrégeant par pointeur différents objets issus d'une lignée d'objets graphiques représentée par la figure suivante :

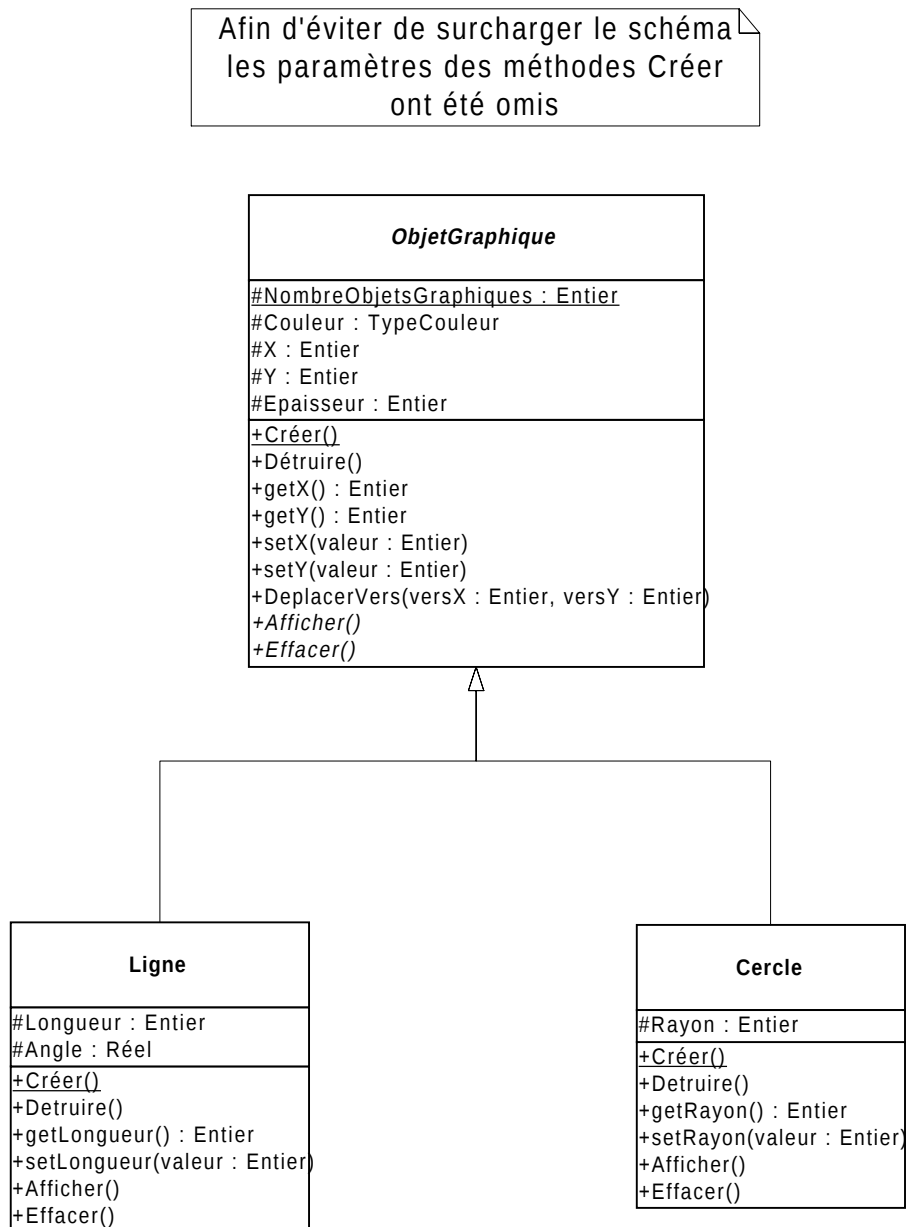


Figure 4.1 La lignée des objets graphiques

Si le conteneur est basé sur le type `ObjetGraphique *`, nous n'avons pas la possibilité de connaître le type exact d'un objet issu du conteneur à un instant quelconque.

Sortons un objet du conteneur. Décidons qu'il s'agit d'un `Cercle` et essayons de lui appliquer la méthode `setRayon` typique de la classe `Cercle`. Cette opération nécessite la promotion du pointeur depuis `ObjetGraphique*` vers `Cercle*`. Nous allons nous assurer de la légitimité de l'opération grâce à **dynamic_cast** par le code suivant :

```
ObjetGraphique *g= ... / récupération d'un pointeur dans un conteneur
Cercle          *c=dynamic_cast<Cercle *>(g)
```

Programme 4.3 utilisation typique de dynamic_cast

A l'exécution, le programme va s'assurer que l'opération est légitime en utilisant des informations de types codées dans les objets et connues sous le nom de RTTI (Run Time Type Information). Cette fonctionnalité sera étudiée dans les chapitres consacrés à la Librairie Standard du C++.

4.2.3.4 Exercices sur les nouveaux opérateurs de conversion

Enoncé : Commenter les opérations de transtypage suivantes :

```
ObjetGraphique *o;
const Ligne     ligne;
Cercle          cercle;
const Cercle    *pCercle;
Ligne           *pLigne;

1) pCercle = const_cast<const Cercle *>(&cercle);
2) pLigne  = dynamic_cast<Ligne *>(pCercle);
3) o = static_cast<ObjetGraphique *>(&ligne);
```

Solution :

- 1) Pas de problème : `&cercle` de type `Cercle*` devient un pointeur constant par l'utilisation de **const_cast**
- 2) Erreur : il n'est pas possible d'ajouter ou de supprimer le caractère **const** avec **dynamic_cast**. Deux solutions sont possibles :
 - a) Convertir `pCercle` en pointeur non constant puis traverser la hiérarchie avec **dynamic_cast**, soit :
`pLigne=dynamic_cast<Ligne *>(const_cast<Cercle *>(pCercle))`
 - b) Faire le **const_cast** à la fin, c'est-à-dire :
`pLigne=const_cast<Ligne *>(dynamic_cast<const Ligne*>(pCercle))`
- 3) Même problème que précédemment : **static_cast** ne permet pas d'ajouter ou de retirer le caractère constant d'un pointeur. En outre, afin de s'assurer que la conversion est légitime, il eut mieux valu utiliser **dynamic_cast**. On retrouve alors les deux solutions :

```
o=const_cast<ObjetGraphique*>(dynamic_cast<const ObjetGraphique*>(&ligne));
```

```
o=dynamic_cast<ObjetGraphique *>(const_cast<Ligne *>(&ligne)) ;
```

4.3 Les conversions implicites

Une conversion implicite est un transtypage effectué par le compilateur dans votre dos, l'exemple le plus simple concerne les conversions implicites entre types numériques.

Il y a toutefois deux cas nettement plus graves où le C++ est en mesure d'effectuer des conversions implicites : les opérateurs de conversion et les constructeurs à un seul paramètre.

4.3.1 Les opérateurs de conversion implicites

4.3.1.1 Syntaxe et utilisation des opérateurs de conversion implicites

Les opérateurs de conversion implicites sont des adjonctions relativement récentes au C++. Ils permettent de réaliser une conversion d'un objet vers un autre type. Ces opérateurs sont nécessairement définis en tant que méthodes et leur syntaxe est la suivante :

operator type_desire(void)

Par exemple si nous reprenons l'exemple de la classe `Rationnel`, nous pourrions créer un opérateur de conversion vers le type **double** de la forme suivante :

```
class Rationnel
{
    // ...
    operator double (void)
    {
        return double(numérateur)/dénominateur;
    }
};
```

Programme 4.4 Définition d'un opérateur de conversion implicite depuis `Rationnel` vers le type `double`

Muni de cet opérateur, il est possible d'employer un rationnel dans des expressions arithmétiques réelles sans effort supplémentaire de programmation :

```
{
    double a=0.0;
    double c;
    Rationnel r(1,2);
    c = a + r;
}
```

Programme 4.5 Utilisation d'un opérateur de conversion implicite

La dernière ligne de cet exemple utilise l'opérateur de conversion implicite de `Rationnel` vers **double** pour convertir `r` en nombre réel, ainsi l'opérateur `+` sur les **double** peut s'appliquer.

Dans le même ordre idée, la plupart des bibliothèques orientées objet (dont OWL et VCL d'Inprise) encapsulant l'API Windows proposent une conversion implicite de leur classe Fenêtre vers le type **HANDLE** de Windows.

Ces conversions sont assurément très pratiques car elles permettent d'éviter un effort de programmation supplémentaire au cours de la programmation. Elles ont toutefois des côtés particulièrement détestables :

- Le code obtenu lors de leur utilisation est certes plus compact mais peut induire en erreur sur le type réel des objets un programmeur chargé de sa maintenance.
- Il faut toujours se méfier de ce que fait le compilateur automatiquement car cela peut occasionner des erreurs pernicieuses comme le montre la section suivante.

4.3.1.2 Exemple de problème soulevé par leur utilisation

Dans l'exemple suivant, nous allons voir qu'une simple faute de frappe peut se transformer en bug très délicat à détecter. Nous reprenons la classe Rationnel munie de son opérateur de conversion explicite vers les réels et utilisons le code suivant :

```
Rationnel r1(1,2);
Rationnel r2(1,3);

// Tentative de comparaison des numérateurs
// Le code devrait être (r1.num() == r2.num())

if (r1.num() == r2) // faute de frappe
{
    .. reste du code omis ...
}
```

Programme 4.6 Bug insidieux dû à un opérateur de conversion implicite

Le but recherché était de comparer `r1.num()` avec `r2.num()`. Malheureusement, et par mégarde, vous avez remplacé `r2.num()` par `r2`. Hors, le compilateur ne signale pas d'erreur, le programme ne plante pas mais donne un résultat faux.

Voici donc ce qui se déroule :

- a) Le compilateur détecte la comparaison de `r1.num()` qui est un **int** avec `r2`, objet de classe `Rationnel`. A priori, la comparaison est illicite et devrait déclencher une erreur de compilation
- b) Le compilateur détecte la possibilité de convertir `r2` en **double**, ce qui permettrait d'effectuer une comparaison tout à fait valable entre un **double** et en entier
- c) Finalement, le compilateur génère pour vous un appel à l'opérateur de conversion implicite, effectue une promotion de `r2.num()` depuis **int** vers **double**, et code une comparaison sur les **double**.

Ce genre de bug non détecté par le compilateur est particulièrement pernicieux à détecter et doit être évité autant que possible

4.3.1.3 Parade

Il n'y a pas de parade à proprement parler. À mon avis, il est toujours dangereux d'utiliser des opérateurs de conversion implicite. Il vaut mieux utiliser une méthode pour faire ce travail.

À ce sujet, il est intéressant de constater que les concepteurs de la STL du C++ utilisent une méthode nommée **c_str** pour convertir une chaîne de classe **string** vers un **char***, cet exemple est à méditer.

4.3.2 Les constructeurs avec un seul paramètre

Il est écrit dans la norme du C++ que les constructeurs avec un seul paramètre peuvent être utilisés à la volée pour construire un objet temporaire.

En outre il faut se souvenir qu'un constructeur avec plusieurs paramètres tous avec des valeurs par défaut ou dont seul le premier n'a pas de valeur par défaut peut se transformer immédiatement en opérateur de conversion implicite.

4.3.2.1 Exemple de problème soulevé par les constructeurs avec un seul paramètre

Dans l'exemple suivant, la classe **Chaîne** dispose d'un constructeur qui prend un entier en paramètre, soit :

```
class Chaîne
{
public:
    Chaîne (int param) ... // Crée une chaîne de param caractères non
                        // initialisés
};
```

Supposons également que la classe **Chaîne** soit munie d'un opérateur de comparaison **==**.

Examinons alors le code suivant :

```
Chaîne c1= new Chaîne("tklp");
Chaîne c2= new Chaîne("tkop");

for (int i=0;i<c1.size();i++)
{
    if (c1==c2[i]) // Erreur de frappe ! ce devrait être c1[i]
    ...
}
```

Programme 4.7 Exemple d'erreur liée aux constructeurs à un seul paramètre

En théorie, l'on s'attendrait à ce que la faute de frappe soit détectée par le compilateur. Hors il faut prendre les éléments suivants en compte :

- Le membre de gauche de la comparaison est une `Chaine` et il existe un opérateur de comparaison sur la classe `Chaine`
- `c2[i]` est un caractère qui est donc assimilable à un entier.

Il existe un constructeur à un paramètre qui construit une `Chaine` à partir d'un entier

Il va donc se produire le scénario suivant :

- Création d'un objet `Chaine` temporaire à l'aide du constructeur `Chaine::Chaine(int)` avec `c2[i]` comme paramètre
- Application de l'opérateur `==` sur `c1` et l'objet temporaire
- Destruction de l'objet temporaire

Ce qui n'est absolument pas le comportement recherché et conduit à des erreurs pernicieuses et délicates à détecter.

4.3.2.2 Parade

Ici, la parade est très simple. En effet, il suffit de préfixer le nom du constructeur par le mot clef **explicit**. Ainsi, un constructeur ne peut jamais être utilisé en tant qu'opérateur de conversion implicite et l'exemple ci-dessus aurait produit une erreur de compilation

Retenez bien le principe : tout constructeur susceptible d'être appelé avec un seul paramètre doit être déclaré **explicit**.

5. En guise de conclusion

Je voudrais dédicacer ce manuel de survie à la promotion 1999 de l'ISIMA au sein de laquelle je compte de nombreux amis. Bien plus que de simples étudiants, ils ont su me donner le goût de l'enseignement. Je sais ce que je leur dois et leur souhaite bonne chance dans la vie.

Bon courage à tous -Bruno