Informatique II :

Cours de programmation (C++)
Programmation Orientée Objet (POO)

Jamila Sam Haroud

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

- Introduire les notions d'**encapsulation** et d'**abstraction**
- **Objets**, instances et **classes**
- Classes en C++
- Variables d'instance
- **Méthodes** d'instance
- Encapsulation et Interface : `public:` et `private:`
- L'objet `this` et le **masquage**

Objets : 1ère introduction

Vous avez appris à écrire des programmes de plus en plus complexes.
Il faut donc maintenant des outils pour **organiser ces programmes** de façon plus efficace.

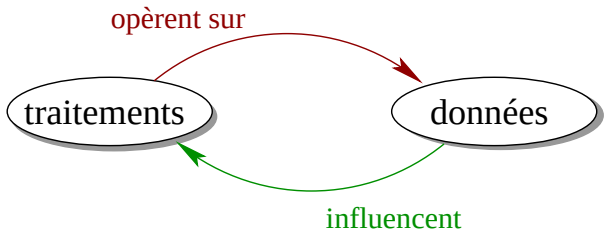
C'est l'un des objectifs principaux de la notion d'**objet**.

Les objets permettent de mettre en œuvre dans les programmes les notions :

- d'encapsulation
- d'abstraction
- d'héritage
- et de polymorphisme

Programmation impérative/procédurale [rappel]

Dans les programmes que vous avez écrits jusqu'à maintenant, les notions de variables/types de **données** et de **traitement** de ces données étaient séparées :



Notions d'encapsulation

Le **principe d'encapsulation** consiste à *regrouper* dans le même objet informatique («concept»), données et traitements qui lui sont spécifiques :

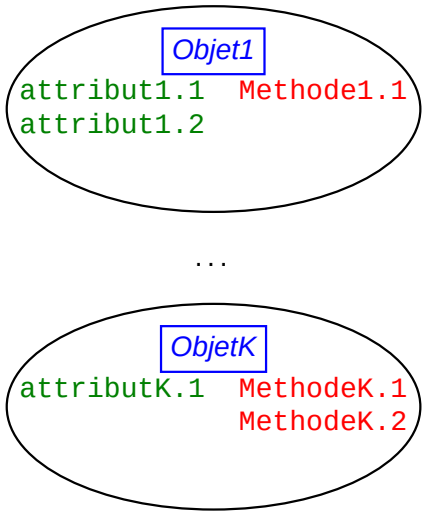
- Les données inclues dans un objet seront appelées les **attributs** de cet objet ;
- Les traitements/fonctions défini(e)s dans un objet seront appelées les **méthodes** de cet objet.

Résumé du premier principe de l'encapsulation :

OBJET
=
attributs
+
méthodes

Notion d'encapsulation (2)

Les objets sont définis par leurs attributs et leurs méthodes :



Notion d'abstraction (1)

Pour être véritablement intéressant, un objet doit permettre un certain degré d'**abstraction**.

Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :

- des caractéristiques communes à tous les éléments
- des mécanismes communs à tous les éléments

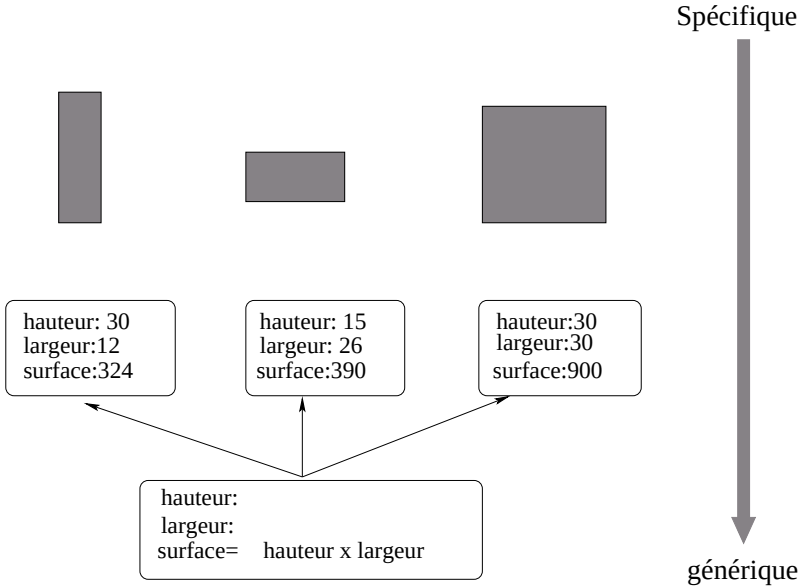
☞ description **générique** de l'ensemble considéré : se focaliser sur l'essentiel, cacher les détails.

Notion d'abstraction (2)

Exemple : Rectangles

- la notion d'*« objet rectangle »* n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes généraux
  - ☞ propriétés et mécanismes **valables pour l'ensemble** des rectangles et non pas pour un rectangle particulier.
- Les notions de largeur et hauteur sont des propriétés générales des rectangles (**attributs**),
- Le mécanisme permettant de calculer la surface d'un rectangle (surface = largeur $\times$ hauteur) est commun à tous les rectangles (**méthodes**)

Abstraction (exemple)



Abstraction et Encaspuslation

Un intérêt de l'encapsulation est que cela permet d'**abstraire** :

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir deux **niveaux de perception** :

- Le niveau externe : partie « visible » (par les programmeurs-utilisateurs) de l'objet :
  - **prototype** des méthodes et attributs accessibles hors de l'objet
  - ☞ c'est l'**interface** de l'objet avec l'extérieur résultat du processus d'*abstration*
- Le niveau interne : (détails d')**implémentation** de l'objet
  - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou l'intérieur d'objets similaires)
  - **définition** de l'ensemble des méthodes de l'objet
  - ☞ c'est le **corps** de l'objet

Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation **Interface** = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser
   - ☞ Concentration sur les attributs/méthodes concernant l'objet (*abstraction*)

Exemple : L'interface d'une voiture

- Volant, accélérateur, pédale de freins, etc.
- Tout ce qu'il faut savoir pour la conduire (mais pas la réparer ! ni comprendre comment ça marche)
- L'interface ne change pas, même si l'on change de moteur... ...et même si on change de voiture (dans une certaine mesure) : abstraction de la notion de voiture (en tant qu'« objet à conduire »)

Pourquoi abstraire/encapsuler ?

L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une *meilleure visibilité* et une meilleure cohérence au programme, d'offrir une plus grande modularité.

L'intérêt de séparer les niveaux interne et externe est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

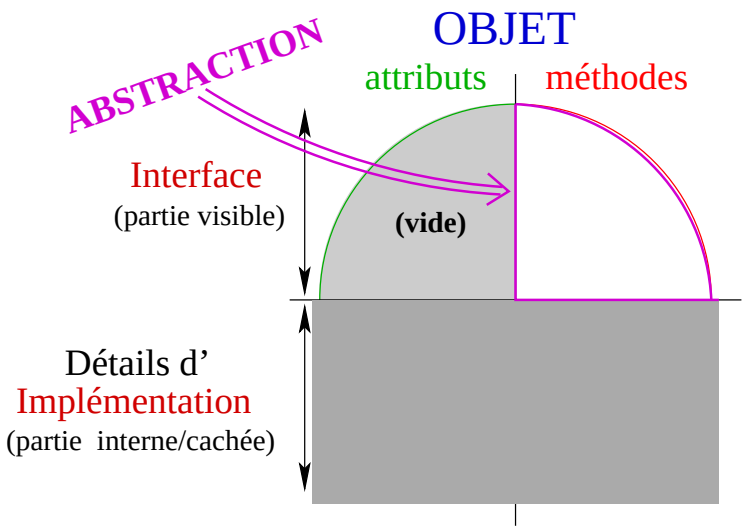
Les objets ne peuvent être utilisés qu'au travers de leur interfaces (niveau externe)
et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur

(même idée que la séparation prototype/définition d'une fonction)

Règle : (masquage) les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.

Encapsulation / Abstraction : Résumé

MIEUX :



Classes et Types

En programmation Objet :

- le résultat du processus d'abstraction s'appelle une **classe**
  classe = catégorie d'objets
- une classe définit un **type** (au sens du langage de programmation)
- une réalisation particulière d'une classe s'appelle une **instance**
  instance = objet

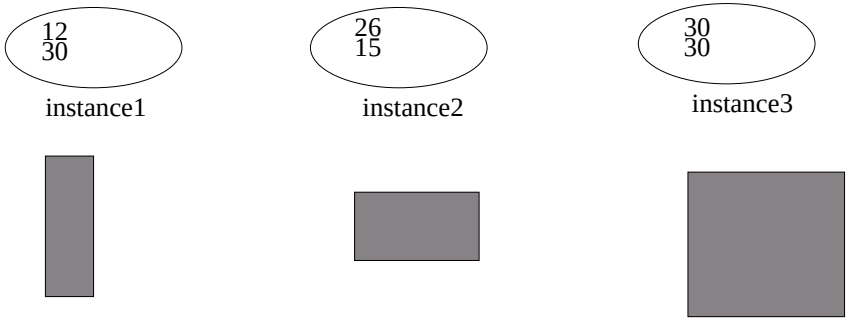
Classes v.s. instances

Rectangle	
Attributs:	méthodes:
largeur **hauteur**	**dessine()** **surface()**

classe (type abstrait)

Existence conceptuelle (écriture du programme)

Existence concrète (exécution)



Les classes en C++

En C++ une classe se déclare par le mot-clé `class`.

Exemple :

```
class Rectangle {
    ...
}; // ne pas oublier le ;
```

La déclaration d'une instance d'une classe se fait de la façon similaire à la déclaration d'une variable classique :

```
nom_classe nom_instance ;
```

Exemple :

```
Rectangle rect1;
```

déclare une instance `rect1` de la classe `Rectangle`.

Déclaration des attributs

La syntaxe de la déclaration des attributs est la même que celle des champs d'une structure.

```
type nom_attribut ;
```

Exemple : les attributs `hauteur` et `largeur`, de type `double`, de la classe `Rectangle` pourront être déclarés par :

```
class Rectangle {
    double hauteur;
    double largeur;
};
```


Accès aux attributs

L'accès aux valeurs des attributs d'une instance de nom `nom_instance` se fait comme pour accéder aux champs d'une structure :

```
nom_instance.nom_attribut
```

Exemple : la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

```
rect1.hauteur
```


Déclaration des méthodes

La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) {
  // corps de la méthode
  ...
}
```

sauf qu'on a **pas besoin de passer les attributs** de la classe comme arguments aux méthodes de cette classe
(on y revient dans une minute)

Exemple : une méthode `surface()` de la classe `Rectangle` pourrait être définie par :

```
class Rectangle {
  ...
  double surface() {
    return (hauteur * largeur);
  }
};
```


Déclaration des méthodes (2)

A noter que ce n'est pas parce qu'on n'a pas besoin de passer les valeurs des attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont jamais d'arguments.

Les méthodes **peuvent très bien avoir des arguments** : ceux qui sont nécessaires (et donc **extérieurs à l'instance**) pour exécuter la méthode en question !
Exemple :

```
class Couleur { ... };

class FigureColoree
{
        //...
        void colorie(Couleur c) { ... }
};

FigureColoree une_figure;
Couleur rouge;
//...
une_figure.colorie(rouge);
//...
```

Actions et Prédicats

En C++ on peut distinguer les méthodes qui modifient l'état de l'objet (« **actions** ») de celles qui ne changent rien à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot `const` **après** la liste des arguments de la méthode :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) const
```

Exemple :

```
class Rectangle {
  ...
  double surface() const {
    return (hauteur * largeur);
  }
  ...
};
```

Si jamais vous déclarez comme prédicat (`const`) une action, vous aurez à la compilation le message d'erreur :

```
assignment of data-member '...' in read-only structure
```


Accès aux méthodes

L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :

```
nom_instance.nom_methode(val_arg1,...)
```

Exemple : la méthode

```
void surface() const;
```

définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :

```
rect1.surface()
```

Autre exemple :

```
une_figure.colorie(rouge);
```


Portée des attributs

Remarque : Les attributs d'une classe constituent des variables directement accessibles dans toutes les méthodes de la classes (*i.e.* des variables globales à la classe). Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

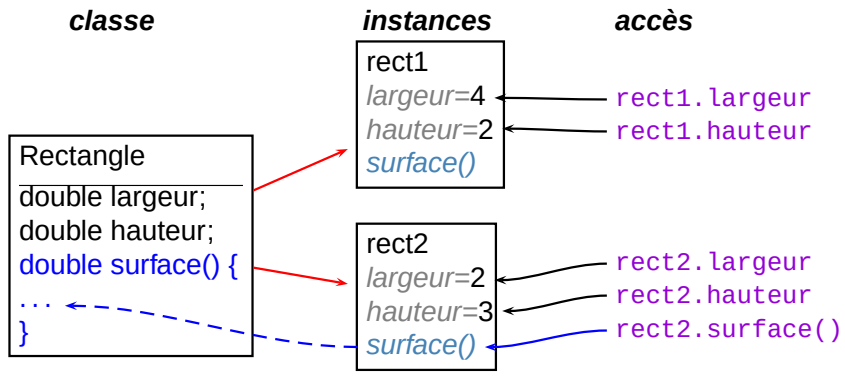
Par exemple, dans toutes les méthodes de la classe `Rectangle`, l'identificateur `hauteur` (resp. `largeur`) fait donc a priori référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de la classe.

Exemple : définition d'une méthode `surface` pour la classe `Rectangle`

```
class Rectangle {
   double surface() const {
     return (hauteur * largeur);
   }
   ...
};
```


Accès aux attributs et méthodes

Chaque instance a ses propres attributs : aucun risque de confusion d'une instance à une autre.



Encapsulation et interface

Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé `private` :

```
class Rectangle {
        double surface() const { ... }
private: // ICI
        double hauteur;
        double largeur;
}
```

Attribut d'instance **privée** = inaccessible depuis l'extérieur de la classe. C'est également valable pour les méthodes.

Erreur de compilation si référence à un(e) attribut/méthode d'instance privée :

```
Rectangle.cc:16: 'double Rectangle::hauteur' is private
```

Note : Si aucun droit d'accès n'est précisé, c'est `private` par défaut.

Encapsulation et interface (2)

À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé `public:`

```
class Rectangle {
public: // accessible partout
    double surface() const { ... }
private:
...
};
```

Dans la plupart des cas :

- **Privé** :
  - Tous les attributs
  - La plupart des méthodes
- **Publique** :
  - Quelques méthodes bien choisies (interface)

Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

- Tous les attributs sont privés ?
  - Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!
- Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...
  1. Manipulateurs (« méthodes set ») :
     - Modification
     - Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h;}

void setLargeur(double L) { largeur = L;}
```

  2. Accesseurs (« méthodes get ») :
     - Consultation
     - Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur;}

double getLargeur() const { return largeur;}
```


Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

Situation typique : (le nom d')un paramètre cache un (nom d')attribut

```
void setHauteur(double hauteur) {
    hauteur = hauteur;  // Hmm.... pas terrible !
}
```

Masquage (2)

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est un **pointeur sur l'instance courante**

`this` ≃ « mon adresse »

Syntaxe pour spécifier un attribut en cas d'ambiguïté :

```
this->nom_attribut
```

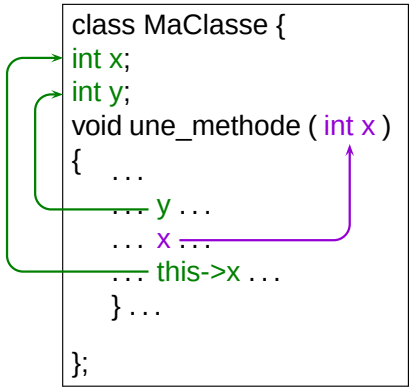
Exemple :

```
void setHauteur(double hauteur) {
   this->hauteur = hauteur;  // fonctionne !
}
```

L'utilisation de `this` est obligatoire dans les situations de **masquage**
(mais évitez ces situations !)

Portée des attributs (résumé)

La portée des attributs dans la définition des méthodes peut être résumée dans le schéma suivant :



Classes = super `struct`

Pour résumer à ce stade, une classe c'est une `struct`

- qui contient aussi des fonctions (« méthodes »)
- dont certains champs (internes) peuvent être cachés (`private:`)
- et dont d'autres constituent l'interface (`public:`)

Un exemple complet de classe

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

// definition de la classe
class Rectangle {
public:
  // definition des methodes
  double surface() const { return (hauteur * largeur); }
  double getHauteur() const { return hauteur; }
  double getLargeur() const { return largeur; }
  void setHauteur(double hauteur) { this->hauteur = hauteur; }
  void setLargeur(double largeur) { this->largeur = largeur; }

private:
  // declaration des attributs
  double hauteur;
  double largeur;
};
```

Un exemple complet de classe (2)

```
//utilisation de la classe

int main() {
  Rectangle rect;
  double lu;
  cout << "Quelle hauteur? "; cin >> lu;
  rect.setHauteur(lu);
  cout << "Quelle largeur? "; cin >> lu;
  rect.setLargeur(lu);

  cout << "surface = " << rect.surface() << endl;

  return 0;
}
```


Opérateur : :

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ **meilleure lisibilité du code**, **modularisation**

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur `::` de résolution de portée :

- La déclaration de la classe contient les *prototypes des méthodes*
- les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(arg1, arg2, ...)
{...}
```


Opérateur : : (2)

Par exemple, la déclaration de la classe `Rectangle` pourrait être (dans `rectangle.h`) :

```
class Rectangle
{
public:
        // prototypes des methodes
        double surface() const;

        double hauteur() const;
        double largeur() const;

        void hauteur(double);
        void largeur(double);

        // declaration des attributs
private:
        double hauteur_;
        double largeur_;
};
```


Opérateur : : (3)

Accompagné des définitions « externes » des méthodes (dans `rectangle.cc`) :

```
double Rectangle::surface() const
{
   return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}
... // idem pour largeur
```

Objets et Classes en C++

`class MaClasse { ... };` déclare une classe.
`MaClasse obj1;` déclare une instance (objet) de la classe `MaClasse`

Les attributs d'une classe se déclarent comme des champs d'une structure : `class MaClasse { ... type attribut; ... };`

Les méthodes d'une classe se déclarent comme des fonctions, mais dans la classe elle-même :

```
class MaClasse { ... type methode(type1 arg1, ...); ... };
```

Encapsulation et interface :

```
class MaClasse {
private: // attributs et methodes privees
...
public: // interface : attributs et methodes publiques
...
};
```

L'attribut particulier `this` est un pointeur sur l'instance courante de la classe. Exemple d'utilisation : `this->monattribut`

Approche modulaire

Jusqu'à maintenant vos programmes était écrit en une seule fois, dans un seul fichier.

Cette approche n'est pas réaliste pour des programmes plus conséquents, qui nécessitent partage de composants, maintenance séparée, réutilisation, ...

On préfére une **approche modulaire**

c'est-à-dire une approche qui
décompose la tâche à résoudre en sous-tâches
implémentées sous la forme de **modules génériques** (qui pourront être **réutilisés** dans d'autres contextes).

Chaque module correspond alors à une tâche ponctuelle, à un ensemble cohérent de données, à un concept de base, etc...

Réutilisabilité

La conception d'un programme doit tenir compte de deux aspects importants :

- la **réutilisation** des objets/fonctions existants : bibliothèques logicielles (« libraries » en anglais);
  (les autres/passé $\longrightarrow$ nous/présent)
- la **réutilisabilité** des objets/fonctions nouvellement créés.
  (nous/présent $\longrightarrow$ les autres/futur)

Remarque : vous pouvez vous-même créer vos **propres bibliothèques**.
(mais non abordé dans ce cours)

Conception modulaire

Concrètement, cela signifie que les types, structures de données et fonctions correspondant à un « concept de base » seront **regroupés dans un fichier** qui leur est propre.

Par exemple, on définira la structure `qcm` et ses fonctions dans un fichier à part de son utilisation.

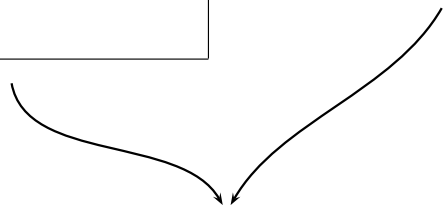
☞ séparation des déclarations des objets de leur utilisation effective (dans un `main()`).

Concrètement, cela crée donc plusieurs fichiers séparés qu'il faudra regrouper (« lier ») pour faire un programme.

Exemple

```
struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...

void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
}
```

```
int demander_nombre(int min, int max);

int demander_nombre(int a, int b) {
  ...
}
```



```
int main ()
{
   qcm maquestion;

   ...
   poser_question(maquestion);
}
```


Compilation séparée

☞ Pouquoi faire cela ?

- Pour rendre **réutilisable** (exemple `demander_nombre`) : évite de réinventer la roue à chaque fois
- Pour **maintenir** plus **facilement** : pas besoin de tout recompiler le jour où on corrige une erreur dans `demander_nombre`
- Pour pouvoir **développer** des programmes **indépendemment**, c'est-à-dire même si le code source n'est pas disponible
- **Distribuer des bibliothèques** logicielles (morceaux de code) sans en donner les codes sources (protection intellectuelle).

☞ Mais comment alors faire un tout (un programme complet) ? Comment `main()` connait-il le reste ?

Compilation séparée

La partie déclaration est la partie **visible** du module que l'on écrit, qui va permettre son utilisation (et donc sa réutilisation).

C'est elle qui est utile aux autres fichiers pour utiliser les objets déclarés.

La partie définition est l'implémentation du code correspondant et n'est pas directement nécessaire pour l'utilisateur du module. Elle peut être **cachée** (aux autres).

Compilation séparée

De ce fait, il est nécessaire (en conception modulaire) de séparer ces parties en deux fichiers :

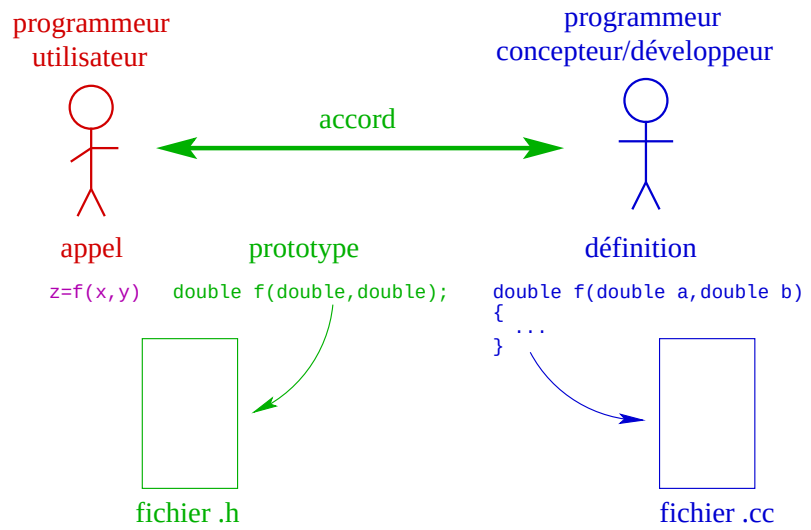
- les fichiers de déclaration (fichiers « *headers* »), avec une extension `.h`).

  Ce sont ces fichiers qu'on inclut en début de programme par la commande `#include`

- les fichiers de définitions (fichiers sources, avec une extension `.cc`)

  Ce sont ces fichiers que l'on compile pour créer du code exécutable.

Compilation séparée



Compilation séparée : exemple

```
struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...

void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
}
```

```
#include "qcm.h"
void affiche(const qcm& question)
{
   ...
}

int poser_question(const qcm& question)
{
   ...
}
```

qcm.cc

```
struct qcm { ... };

void affiche(const qcm& question);
int poser_question(const qcm& question);
...
```

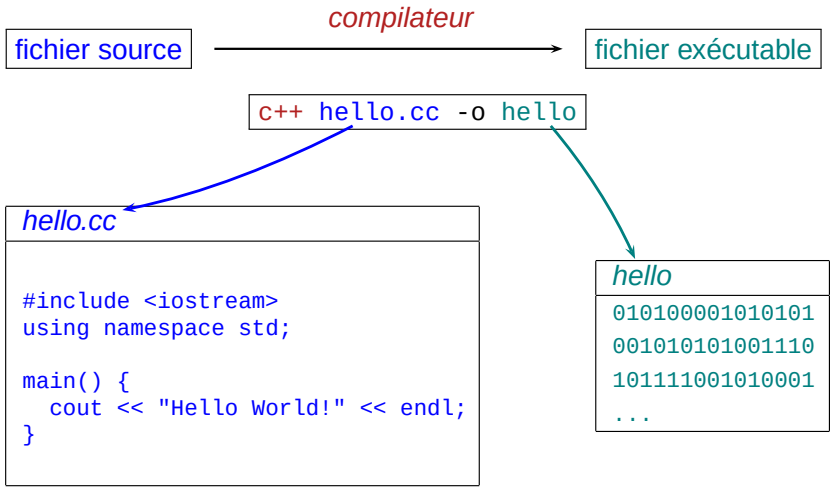
qcm.h

Compilation séparée (2)

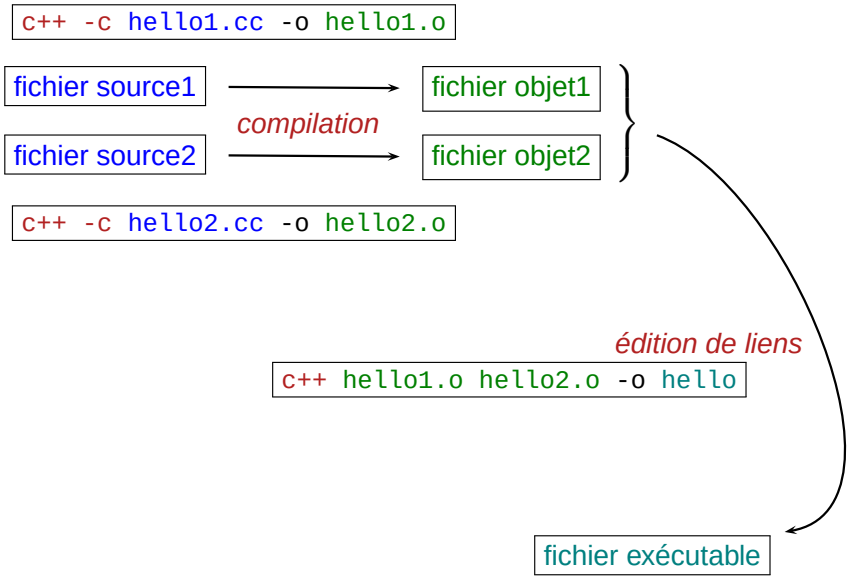
La séparation des parties déclaration et définition en deux fichiers permet une **compilation séparée** du programme complet :

- phase 1 : production de fichiers binaires (appelés fichiers **objets**) correspondant à la compilation des fichiers sources (`.cc`) contenant les parties définitions (et dans lesquels on inclut (`#include`) les fichiers « *headers* » (`.h`) nécessaires) ;
- phase 2 : production du fichier exécutable final à partir des fichiers objets.

RAPPEL : Compilation d'un programme C++



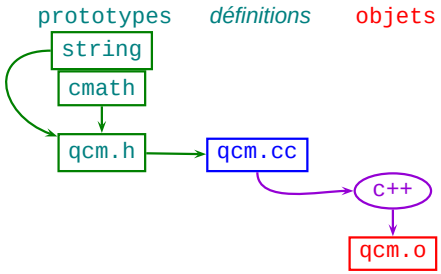
Compilation séparée d'un programme C++



Compilation séparée : phase 1

Pour créer un **fichier objet** (identifié par une extension `.o`), on utilise une commande de compilation du type :

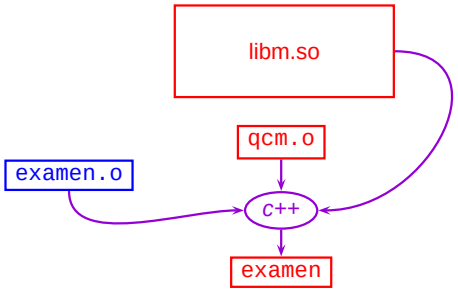
```
c++ qcm.cc -c -o qcm.o
```



Compilation séparée : phase 2

Le fichier exécutable est produit par une compilation qui **intègre** (« lie ») les fichiers objets nécessaires :

```
c++ examen.o qcm.o -lm -o examen
```



Outils de construction de projets

Compiler "à la main" un projet rédigé sur plusieurs fichiers est laborieux.
Il existe de nombreux outils pour automatiser ce traitement :

Par exemple :

- les outils intégrés de développement de projets (IDE) : Code::Blocks, KDevelop, Anjuta, NetBeans, Eclipse, ...
- CMake, `http://www.cmake.org/`,
- SCons, `http://www.scons.org/`,
- Jam (BJam, KJam, ...)
- the GNU Build Tools, alias « autotools » (`automake`, `autoconf` and `libtool`), cf `http://sourceware.org/autobook/`, `http://autotoolset.sourceforge.net/tutorial.html`

☞ Nous travaillerons avec SCons : **Etudiez le tutoriel de la série 4**

Ce que j'ai appris aujourd'hui

- quels sont les principes de base de la programmation orientée-objet : encapsulation, abstraction, héritage et polymorphisme ;
- que l'on peut « encapsuler » données et traitements en utilisant des classes d'objets ;
- qu'il faut clairement dissocier l'interface (« publique ») de la représentation interne (« privée ») ;
- comment réaliser les éléments de base de la POO en C++ :
  - attributs et méthodes (d'instances) ;
  - encapsulation et interface : `public:`, `private:`, « méthodes get » (accesseurs) et « méthodes set » (manipula-teurs) ;
  - l'attribut `this` et le (dé)masquage.

☞ je peux maintenant mieux concevoir mes programmes grâce à la *programmation orientée objet*

La suite

- Exercices de cette semaine :
  - Premiers programmes orientés objets
  - Prise en main de l'environnement graphique (SFML)
  - Prise en main de SCons (outil de compilation séparée)
- Le prochain cours :
  - Suite des concepts de POO : constructeurs et destructeurs
  - Surcharge des opérateurs
- Ensuite :
  - Héritage
  - Polymorphisme
  - Héritage multiple