# Une introduction à la programmation objets et au langage C++

J.F. Rabasse

12/98

**Présentation :**

Ce guide est une introduction à la conception orientée objets, principes, concepts, vocabulaire, et à la mise en œuvre via le langage de programmation C++.

Comme son nom l'indique, C++ est un surensemble du langage C et les deux partagent donc un important noyau commun. Ce guide ne traite que des spécificités de C++ par rapport à C. Il s'adresse donc à un lectorat ayant déjà la pratique (ou au moins une connaissance syntaxique raisonnable) du langage C.

Un autre guide, *Une introduction au langage C* (du même auteur et dans la même collection), est disponible.

Institut de Physique Nucléaire – S2I 91406 Orsay Cedex

Copyright © 1998 IPN

Rédaction, publication :
Institut de Physique Nucléaire -- S2I
91406 Orsay Cedex (France)

Document mis à jour le :
14/12/1998

Auteur :
Jean-François RABASSE

| | |
|---|---|
| IPN Orsay | Bat. 100 - A112 |
| Téléphone | (33) 01 69 15 51 54 |
| E-mail | rabasse@ipno.in2p3.fr |

# Sommaire



../.

# Table des matières

# 1 Introduction, mise en œuvre

Ce manuel veut se donner un double objectif : présenter les principes de la conception orientée objets[1] et toutes les notions afférentes, et d'autre part détailler la mise en œuvre à l'aide du langage C++.

En toute rigueur, ces deux points sont indépendants l'un de l'autre. La conception orientée objets est un cadre de programmation, une nouvelle manière de *penser* les applications informatiques, alors que C++ n'est qu'un outil de mise en œuvre parmi d'autres langages orientés objets, *SmallTalk*, *Java*...

Ceci étant, il est souvent malcommode d'exposer des concepts sans s'appuyer sur un matériel exemplaire. Ce manuel se voulant essentiellement pratique, les notions objets seront donc illustrées au fur et à mesure par leur expression syntaxique en C++ (voir la Note 1.1 en fin de chapitre).

Il appartiendra au lecteur de séparer mentalement les deux aspects, concepts d'une part, programmation d'autre part, pour pouvoir à terme mettre en œuvre un autre outil de programmation (*Java* par exemple).

La structure de ce manuel va d'ailleurs dans ce sens. Tous les chapitres, à partir du chapitre 2, détaillent les grands concepts de la programmation objets illustrés par leur mise en œuvre avec C++. Les annexes, elles, présentent un certain nombre de fonctionnalités C++, utiles ou indispensables, mais qu'on ne retrouvera pas nécessairement (ou pas sous la même forme) dans d'autres langages objets.

1. ou COO. En anglais OOD, *Objects Oriented Design*.

## 1.1 Le langage

Le langage C++, inventé par Bjarne STROUSTRUP vers 1983 et décrit par l'ouvrage [4], est une évolution *orientée objet* du langage C de Brian KERNIGHAN et Denis RITCHIE, [1].

Il s'est enrichi, au cours de la décennie 80, parallélement à la stabilisation et la normalisation de C (norme C ANSI de 1989).

C++ est actuellement en cours de normalisation, la référence syntaxique et fonctionnelle du langage étant l'ouvrage de STROUSTRUP remis à hauteur en 1992, *The C++ Programming Language*, $2^{nd}$ edition, [5].

Cet ouvrage est un « pavé », relativement indigeste mais exhaustif et de qualité. Le présent manuel étant une initiation, il est loin de comporter toutes les finesses et subtilités[2] de la programmation en C++, l'ouvrage de STROUSTRUP reste LA référence.

## 1.2 Mise en œuvre

Il n'existe pas de standard universel pour désigner les noms de fichiers source C++. Un certain nombre d'extensions de noms existent : `.cxx`, `.cc`, `.C`, `.cpp`, `.c++`, ..., selon les plateformes et compilateurs utilisés.

L'extension la plus universelle est `.cxx` et c'est celle qui sera utilisée dans tout ce guide (voir Note 1.2).

A l'IPN, on dispose des outils suivants :

- DEC C++ sur machines VAX/VMS
  ```
  cxx toto.cxx
  ```
- DEC C++ sur machines Alpha/DEC-Unix
  ```
  cxx -c toto.cxx
  ```
- Sun C++ sur machines Sun/Solaris
  ```
  CC -c toto.cxx
  ```
- GNU C/C++ sur machines Alpha/DEC-Unix et Sun/Solaris
  ```
  g++ -c toto.cxx
  ```
  (Sous réserve d'avoir activé l'environnement GNU C à l'aide de la commande : `use gcc`. Voir également la Note 1.3)
- GNU C++ sur certains systèmes Linux
  ```
  g++ -c toto.cxx
  ```

L'édition de liens du ou des modules objets se fait de manière classique :

- Linker VMS
  ```
  link/exec=toto.exe toto.obj
  ```

2. Oui ! C++ est subtil, n'en déplaise aux mécréants !

- Linkers UNIX

```
cxx -o toto toto.o
CC -o toto toto.o
g++ -o toto toto.o
```

**Nota :** on se souviendra que, sous UNIX, les minuscules et majuscules sont différentiées dans les lignes de commandes. Ainsi, avec les outils Sun, `cc` lance une compilation C et `CC` lance une compilation C++ !

## 1.3 Compatibilités

On remarquera que, contrairement à la compilation de sources C, la compilation en C++ n'utilise pas d'options ANSI ou autres.

La syntaxe de l'ANSI relative aux déclarations, prototypes de fonctions et autres, est obligatoire en C++. Un compilateur C++ est capable de compiler un code source écrit en C ANSI pur[3].

Certains compilateurs, c'est le cas du GNU C/C++, sont capables de traiter les deux dialectes, sur détection de l'extension de nom du fichier source (ou, parfois, via une option spécifique, c'est le cas du Turbo-C/C++ de Borland).

Il y a donc compatibilité ascendante, C vers C++, au niveau de l'écriture. Il n'est par contre pas possible de mélanger sans précautions, au sein d'un même programme, des modules objet compilés en C et en C++ (voir l'annexe B).

## 1.4 Spécificités du langage

Le langage C++ est un surensemble de C et repose sur les mêmes mécanismes d'écriture et de génération :

- Déclarations préalables et obligatoires.
- Prototypages des fonctions, selon les conventions de l'ANSI, obligatoires.
- Exploitation de toutes les fonctionnalités du préprocesseur C pour les inclusions de fichiers, code conditionnel, macro définitions, etc.

De fait, une connaissance pratique de C est indispensable pour aborder le C++. Si C++ apporte, par rapport à C, des notions nouvelles et fondamentales, la syntaxe elle est très peu différente.

La syntaxe du C++ est 95% de C et 5% d'ajouts, et ce guide ne traite que des notions nouvelles (et des 5% syntaxiques). En particulier, on ne reviendra pas sur « comment écrire une boucle, un test », « comment coder un calcul », « comment ouvrir un fichier », etc. Tout ceci est conforme au langage C, voir [1] ou [2].

3. Mais l'inverse est évidemment faux.

## 1.5 Extensions par rapport à C

Ce paragraphe s'adresse aux lecteurs pratiquant déjà le C et peut être sauté en première lecture.

### 1.5.1 Nouveaux concepts

C++ ajoute à C trois notions importantes et quelques améliorations, moins fondamentales mais intéressantes :

- Classes et objets : notion fondamentale et qui est le cœur même de la programmation orientée objets. Cette notion est développée aux chapitres 2 et 3.

- Surcharge de sélection : c'est la possibilité de définir des traitements (fonctions) à un niveau conceptuel plus élevé qu'en programmation classique, en reportant sur le compilateur le maximum de détails de mise en œuvre, en particulier les traitements associés aux types des arguments d'appel. Cette notion est développée au paragraphe 3.2.

- Héritages : c'est la possibilité de construire de nouveaux objets par réutilisation (dérivation) et modification d'objets existants (surcharge fonctionnelle). Cette notion est développée aux chapitre 4 et 6.

- Surcharges d'opérateurs : c'est une extension des règles d'écritures arithmétiques classiques à des objets non scalaires. Le principe est illustré en annexe C.

- Entrées/sorties par *streams* : c'est une amélioration élégante des mécanismes de lecture/écriture de données. L'annexe D documente ces mécanismes.

- Classe ou fonctions patrons (*templates*). Il s'agit d'un mécanisme d'aide à l'écriture des codes C++, permettant de définir, sous une forme semi-symbolique, des traitements similaires. Cette possibilité n'est disponible que depuis peu de temps dans les compilateurs C++. Elle est illustrée en annexe E.

- Gestion d'erreurs et exceptions : C++ a introduit, tout récemment, des principes sophistiqués de gestion des erreurs et problèmes en exécution. Ces mécanismes ne seront pas traités dans ce guide[4].

### 1.5.2 Évolutions syntaxiques

Si la conception de programmes, en C++, n'a plus grand chose à voir avec la conception en C, la syntaxe elle est très peu modifiée par rapport au C ANSI. Les différences portent sur :

- Introduction d'un commentaire ≪jusqu'à fin de ligne≫, symbolisé par `//` :

```
int mini ; // Valeur mini
```

4. À l'heure actuelle, et y compris à l'IPN, tous les compilateurs ne supportent pas encore ces fonctionnalités.

en plus du commentaire « bloc » de C `/* ... */`, toujours reconnu.

- Introduction de nouveaux mots-clés correspondant aux nouvelles fonctionnalités : `class`, `public`, `private`, `protected`, `friend`, `virtual`, `this`, `template`, `operator`, `new`, `delete`, et au support d'erreurs quand il est disponible[4] : `try`, `throw`, `rethrow`, `catch`.

- Modification ou extension de la sémantique de certains mots-clés du C : `extern`, `static`, et de certains opérateurs : `&`, `>>`, `<<`. C'est sur ces derniers points que les habitué(e)s de C devront se méfier.

# Notes

**[1.1]** *Le choix de C++, pour une introduction à la conception objets, est du au fait que c'est aujourd'hui LE langage de développement orienté objets.*

*Des langages antérieurs, comme* SmallTalk, *sont trop peu répandus pour justifier un investissement. D'autres, comme le récent langage* Java, *sont plus spécialisés et prennent donc tout leur intérêt dans des environnements bien particuliers.*

**[1.2]** *En tout état de cause, il est fortement déconseillé d'utiliser des extensions par trop spécifiques à une plateforme. Des fichiers source C,* `toto.c`, *et C++,* `toto.C`, *valides sur une plateforme Unix vont provoquer des collisions de noms en cas de migration vers un système qui ne distingue pas la casse des noms de fichiers (VMS, MS-DOS) !*

**[1.3]** *Le compilateur GNU est un compilateur mixte C/C++. On pourra compiler des sources C++ en utilisant l'une ou l'autre des commandes* `gcc` *ou* `g++`.

*Pour l'édition de liens, par contre, l'invocation de* `g++` *permet de configurer correctement toutes les options nécessaires au* linker. *En particulier, lorsqu'on utilise des* packages *spécifiques C++, iostreams, nombres complexes, etc., les librairies seront invoquées implicitement. Avec la commande* `gcc`, *seule la librairie standard C est mise en œuvre.*

# 2 Objets et classes

LE CONCEPT OBJET est l'essence même de la programmation orientée objets[1]. La vision du monde (informatique) qu'a une application se ramène à la construction et la manipulation d'un ensemble d'objets. Partant du principe que tout, dans l'univers, peut être représenté par un ou des objets[2] la partie *principale* d'une application objets se résoud à : construire « l'objet application » (lequel met en œuvre autant de sous-objets que nécessaire) et le faire vivre jusqu'à la fin du programme.

## 2.1 L'objet logiciel

Un objet est une entité informatique, autonome et modulaire, disposant de son propre code de manipulation, les *méthodes*, et enfermant ses propres données, les *attributs*. On parle également de *fonctions membres* et *données membres*.

Les méthodes traduisent le : « que sait faire l'objet ? », les attributs traduisent le : « que connait l'objet ? ».

L'analogie avec un objet concret, par exemple un téléviseur, est immédiate : un téléviseur dispose de *méthodes* (boutons de face avant) permettant de l'utiliser, « mise en marche », « changement de canal », « réglage de volume », etc., et comporte en interne les *attributs* (composants électroniques) nécessaires à son fonctionnement.

On remarquera, et c'est également vrai pour l'objet informatique, que l'utilisation d'un objet suppose de savoir s'en servir mais ne nécessite pas la connaissance de son fonctionnement interne.

De plus, dans le cadre d'un développement (informatique ou industriel), la priorité est donnée à la spécification des méthodes, à ce que l'objet

1. On s'en doutait un peu !
2. Seul Dieu échappe sans doute à cette règle, et encore...

≪doit pouvoir faire≫, sans forcément savoir a priori comment le faire. L'implémentation interne sera ensuite la mise en œuvre des spécifications.

## 2.2 Classes et instances

Dans la terminologie objets, on appelle *classe* la famille, le type, la nature de l'objet. On appelle *instance* une occurence, une réalisation d'un objet de classe donnée.

≪Mon téléviseur≫ (instance) n'est pas le même objet physique que celui (autre instance) de mon voisin. Et ceci même si les deux appareils sont du même modèle (même classe).

En programmation traditionnelle, ces notions existent sous les appellations *type* et *variable*. Ainsi, en Fortran :

```
real    x, y
integer n
```

on déclare deux instances, nommées `x` et `y`, de la classe numérique réel et une instance, nommée `n`, de la classe numérique entier.

## 2.3 Mécanismes de spécification

A titre d'exemple (minimaliste) on désire, dans le cadre d'une application géométrique, disposer d'objets `Cercle`.

L'utilisateur potentiel de l'objet va spécifier les méthodes requises (ces méthodes sont des fonctions informatiques classiques, avec nom, arguments d'appel, type retour) :

- une méthode `Move`, appelée avec deux arguments réels, permet de déplacer l'objet en relatif.
- une méthode `Zoom`, appelée avec un argument réel, permet d'appliquer un facteur d'échelle à l'objet.
- une méthode `Area`, sans arguments, retourne la surface de l'objet.

A partir de ces spécifications, le développeur d'objet va choisir d'implémenter trois attributs numériques, la position `X, Y` du centre et le rayon du cercle.

**Nota :** dans une bonne conception objet, tout attribut doit pouvoir être justifié relativement à la mise en œuvre d'une ou plusieurs méthodes. C'est pourquoi cette étape s'effectue dans un deuxième temps.

## 2.4 Interface de classe

En C++, une interface de classe est une description symbolique de l'objet, destinée au compilateur. A ce titre, elle comporte la définition *publique* de la classe, méthodes utilisateurs, et l'implémentation *privée* des attributs (et/ou méthodes internes) nécessaires au fonctionnement.

Traditionnellement, une interface est écrite dans un fichier *header* séparé, portant le nom de la classe. Par exemple, ici, `cercle.h`

```
class Cercle
{
    // Methodes utilisateurs
public:
    void    Move(float deltax, float deltay);
    void    Zoom(float scale);
    float   Area();

    // Attributs implantation
private:
    float   cX, cY;      // Position centre
    float   cR;          // Rayon
};
```

**Nota:** cet exemple est volontairement réduit à son strict minimum et va être complété au cours de ce chapitre.

## 2.5 Instanciations

Le programme application désirant utiliser une classe doit inclure dans son code source le fichier interface, puis déclarer une ou plusieurs instances de cette classe et les manipuler :

```
#include <stdio.h>          // Librairie standard C
#include "cercle.h"         // Interface classe

int main()
{
    float S;                // Variable locale
    Cercle toto;            // Instance de cercle

    /* Deplace toto */
    toto.Move(150, 0);

    /* Affiche sa surface */
    S = toto.Area();
    printf("Surface = %f\n", S);
```

```
    /* Zoom et reaffiche */
    toto.Zoom(1.5);
    S = toto.Area();
    printf("Surface apres zoom = %f\n", S);

    return 0;
}
```

On remarquera la syntaxe des appels de méthodes. Le nom de la méthode est insuffisant par lui-même, il faut spécifier l'objet auquel on veut l'appliquer, i.e. le nom de l'instance.

On pourrait vouloir manipuler deux cercles :

```
    Cercle c1, c2;

    c1.Zoom(3);
    c2.Zoom(0.5);

    float S1, S2;
    S1 = c1.Area();
    S2 = c2.Area();
```

auquel cas la surface de `c1` n'a aucune raison d'être la même que la surface de `c2`. C'est le travail du compilateur d'assurer que les méthodes seront appelées sur les objets ad-hoc.

**Nota :** contrairement à C, C++ autorise les déclarations, scalaires ou objets, à n'importe quel endroit du code et non plus uniquement en en-tête de fonction. Cf. ci-dessus la déclaration de `S1` et `S2`.

## 2.6 Implémentation de classe

L'exemple ci-dessus montre que l'on est capable d'écrire du code source utilisant des objets sans se préoccuper de l'implantation. (On peut savoir utiliser un téléviseur sans être électronicien, sans savoir le construire !) L'interface et la connaissance de ce que font les méthodes est suffisante.

Pratiquement, il faudra tout de même assurer l'implémentation de l'objet et l'écriture effective du code des méthodes. Traditionnellement, un objet est encodé dans un fichier source portant le nom de la classe. Par exemple, ici, `cercle.cxx`

```
#include "cercle.h"      // Interface classe

/* Deplacement */
void Cercle::Move(float deltax, float deltay)
{
    cX += deltax;
    cY += deltay;
}
```

```
/* Echelle */
void Cercle::Zoom(float scale)
{
    cR *= scale;
}

/* Surface */
float Cercle::Area()
{
    return 3.14159 * cR * cR;
}
```

L'exemple est suffisament simpliste[3] pour que la programmation n'appelle pas de commentaire particulier. On notera simplement la syntaxe de définition d'une méthode :

*NomClasse* : : *NomMethode*

Dans le code des méthodes, les attributs sont invoqués sous leur nom symbolique, tel que défini dans la déclaration d'interface. En pratique, chaque instance d'une classe dispose de son propre jeu d'attributs et c'est au compilateur C++ de faire en sorte que les données convenables soient manipulées.

Ainsi, lorsqu'un code utilisateur manipule plusieurs instances :

```
Cercle c1, c2;

c1.Zoom(3.0);
c2.Zoom(0.5);
```

c'est bien l'attribut `cR` de l'instance `c1` qui sera multiplié par 3, et l'attribut `cR` de l'instance `c2` qui sera divisé par 2.

Le code d'implémentation d'un objet est donc une programmation générique, se référant à « l'objet courant pour l'appel ».

C'est : MA-GIQUE !

(Ou presque, voir le paragraphe A.3 en annexe.)

## 2.7 Visibilités

On aura remarqué, dans la description d'une interface de classe, paragraphe 2.4, la présente de deux *attributs de visibilité*, `public` et `private`.

Pour des raisons liées à l'implémentation des compilateurs C++, une interface de classe doit comporter la description exhaustive de l'objet, méthodes publiques mais aussi tous les attributs et méthodes privées. Les attributs de visibilité ont pour rôle de signaler au compilateur les invocations autorisées ou interdites :

- le code interne à l'objet, i.e. la programmation des méthodes de classe, a accès à tout, public ou privé !

3. Pour ne pas dire : bêbête !

- le code externe, en général le code utilisateur, n'a accès qu'aux invocations publiques. L'accès explicite à un attribut privé n'est pas autorisé :

```
Cercle c1;

c1.Zoom(2.0);     // Ok, methode publique

c1.cR *= 2.0;     // Erreur en compilation !
```

Ce mécanisme, appelé aussi *encapsulation des données*, est là pour obliger le code utilisateur à passer par les méthodes prévues par le concepteur. Ce n'est pas une contrainte mais plutôt une sécurité.

D'une part, le concepteur de l'objet qui assure la maintenance des fichiers relatifs à l'objet, fichier interface `machin.h` et fichier source `machin.cxx`, reste libre de modifier l'implémentation interne, de changer des noms d'attributs, d'en ajouter, d'en supprimer. Le code utilisateur reste valide (à une recompilation près) tant que l'interface publique est stable (et, en principe, elle doit l'être puisqu'elle résulte d'une spécification initiale).

D'autre part, et c'est souvent le cas en pratique, un objet important exige une certaine cohérence sur ses attributs. Parfois, une simple modification de valeur peut nécessiter un recalcul d'autres attributs. En obligeant le code utilisateur à passer par des « guichets », le concepteur est assuré de pouvoir maintenir un objet consistant.

Par défaut, dans une classe C++, tout est privé. Ces attributs sont à placer aux endroits ad-hoc de la définition de classe, on peut les alterner, la visibilité est valide pour « tout ce qui suit » jusqu'au prochain attribut.

**Nota :** d'autres mécanismes de protection existent, voir le chapitre 5.

## 2.8 Cycle de vie

Un objet logiciel a une vie et une mort[4].

Le langage offre la possibilité d'implémenter deux méthodes de classe particulières, le *constructeur* et le *destructeur*. La compilation assure que le constructeur, s'il est spécifié, sera appelé lors de la création d'une instance et avant la toute première utilisation, et que le destructeur, s'il est spécifié, sera appelé juste avant la destruction physique de l'objet.

Ainsi, dans l'exemple ci-dessus, on ne sait pas, lorsqu'on instancie un objet cercle, ce que valent ses attributs. Le rôle typique d'un constructeur est d'assurer une initialisation par défaut qui soit convenable.

Ces méthodes se déclarent sans type et avec un nom imposé, `NomClasse` pour un constructeur, `~NomClasse` pour un destructeur.

4. On en est tous là !

L'interface de classe (fichier `cercle.h`) deviendrait :

```
class Cercle
{
    // Constructeur, destructeur
public:
    Cercle();
    ~Cercle();

    // Methodes utilisateurs
public:
    void    Move(float deltax, float deltay);

    etc...
```

et l'implémentation (fichier `cercle.cxx`) comporterait :

```
/* Constructeur */
Cercle::Cercle()
{
    cX = cY = 0.0;    // A l'origine par defaut
    cR = 1.0;         // Rayon unite
}

/* Destructeur */
Cercle::~Cercle()
{
    // Rien a faire de special !
}
```

Le mécanisme d'initialisation par constructeur est, en pratique, indispensable, c'est le moyen d'assurer que tout objet commence sa vie avec une configuration plausible. Dans un environnement objets bien conçu, les mécanismes d'instanciation doivent toujours produire des objets utilisables en l'état et le code utilisateur ne devrait avoir à reconfigurer que lorsque les valeurs par défaut sont inacceptables.

Un autre mécanisme très intéressant est la possibilité qu'a le concepteur d'objet de proposer différents constructeurs de classe. Par exemple, on pourrait proposer un constructeur de cercles prenant une valeur initiale de rayon en argument.

Interface de classe :

```
class Cercle
{
    // Constructeurs, destructeur
public:
    Cercle();
    Cercle(float rayon);
    ~Cercle();

    ...
```

Implémentation :

```
/* Constructeur par defaut */
Cercle::Cercle()
{
    cX = cY = 0.0;    // A l'origine par defaut
    cR = 1.0;         // Rayon unite
}

/* Constructeur avec rayon */
Cercle::Cercle(float rayon)
{
    cX = cY = 0.0;    // A l'origine par defaut
    cR = rayon;       // Rayon utilisateur
}
```

Le code utilisateur pourra ensuite instancier des cercles en spécifiant tel ou tel constructeur :

```
Cercle c1;      // Construction par defaut, rayon 1.0

Cercle c2(5.5); // Construction avec argument rayon
```

L'appel correct de tel ou tel constructeur sera effectué par le compilateur C++, en fonction du contexte de déclaration et ce, de manière transparente pour le code utilisateur. On n'appelle jamais explicitement un constructeur ou un destructeur :

```
Cercle c1;      // Le constructeur est appele ici

c1.Cercle();    // Non !
```

L'implantation d'un destructeur (qui lui, ne peut être multiple ni comporter d'arguments) est moins fréquente. Dans l'exemple ci-dessus elle est inutile puisqu'on ne fait rien. Elle se justifie lorsqu'un objet nécessite des tâches de « ménage » lors de sa destruction : fermeture d'un fichier qui aurait été ouvert par l'objet, libération d'une zone mémoire allouée, etc. Comme le destructeur est également appelé automatiquement, on est ainsi sûr de ne rien oublier.

## 2.9 Gestion des objets

Le langage C dispose de trois mécanismes de durée de vie des données :

- statique ou global : la durée de vie des données est celle du programme.
- automatique ou local : la durée de vie est limitée à un contexte { ... } de programme.
- dynamique : les données sont crées (`malloc()`) et détruites explicitement (`free()`), leur manipulation s'effectue via un *pointeur*.

Les mêmes mécanismes sont utilisables en C++, pour les variables classiques et, a fortiori, pour des objets.

### 2.9.1 Gestion directe

```
Cercle c1;          // Global, statique

void toto()
{
    Cercle c2;      // Local
    ...
    if( ... ) {     // Nouveau contexte
       Cercle c3;   // Local
       ...
       ...
    }               // Fin de contexte, c3 est detruit
    ...
}                   // Fin de fonction, c2 est detruit
```

Un objet est créé (i.e. la mémoire est allouée puis le constructeur est appelé) lors de sa déclaration. Un objet global (`c1` dans l'exemple ci-dessus) est construit au lancement du programme.

Un objet est détruit (i.e. le destructeur est appelé puis la mémoire est libérée) lorsque qu'il devient *hors de portée*, lorsque l'on quitte le contexte qui le déclarait. Un objet global est détruit en fin de programme.

**Nota :** on voit ici toute la puissance apportée par la possibilité de programmer un destructeur lorsqu'un objet nécessite des tâches de « ménage », libération de ressources et autres : un simple `return`, n'importe où dans une fonction, fera quitter le contexte et invoquera implicitement toutes les opérations de nettoyage sur tous les objets locaux existants.

### 2.9.2 Gestion indirecte

La gestion de variables par pointeurs, familière aux programmeurs C, s'effectue de manière identique en C++ :

- déclaration d'un pointeur typé
- allocation d'une donnée

- manipulation classique via l'opérateur `->` ou l'opérateur d'indirection `*`, cf. exemple ci-dessous

- enfin, destruction explicite

```
float S;
Cercle *pc;       // Pointeur de Cercle

pc = new Cercle;  // Allocation

pc->Zoom(3.5);    // Appel methode

(*pc).Zoom(5.0);  // Autre syntaxe possible

S = pc->Area();   // Appel methode
...
delete pc;        // Liberation
```

La différence porte sur la mise en œuvre des allocations et libérations. En C++ on n'utilise JAMAIS les appels `malloc()` et `free()` !

L'opérateur `new` effectue l'allocation mémoire pour l'objet ET l'appel du constructeur. L'opérateur `delete` effectue l'appel du destructeur puis la libération mémoire. Dans le cas de constructeurs multiples, c'est la syntaxe de création qui lève les ambiguïtés :

```
Cercle *pc1, *pc2;

*pc1 = new Cercle;       // Constructeur par defaut

*pc2 = new Cercle(5.0);  // Constructeur "avec rayon"
```

A noter que ces opérateurs sont également utilisables sur des variables simples et existent sous une forme « tableaux » :

```
int   *pi;
float *pf;

pi = new int;        // Allocation scalaire
pf = new float[50];  // Allocation tableau
...
delete pi;           // Liberation scalaire
delete[] pf;         // Liberation tableau
```

Remarquer la notation `delete[]` pour détruire un tableau. On peut manipuler des tableaux de scalaires mais aussi des tableaux d'objets. Dans le cas de tableaux d'objets, les appels des constructeurs et destructeurs sont effectués sur tous les éléments du tableau.

**Nota :** tout comme en C, un pointeur déclaré dans une fonction est détruit (en tant que variable pointeur) lorsqu'on quitte le contexte, mais la mémoire éventuellement associée n'est pas libérée !

En C++, en cas de création d'objets par `new`, on devra assurer la destruction explicite par `delete`, à un moment ou un autre de l'exécution : destruction avant de quitter le contexte, ou sauvegarde du pointeur avant retour pour destruction ultérieure.

# 3 Appels et surcharges

## 3.1 Méthodes de classes

Contrairement aux autres langages de programmation où la portée d'un identificateur est unique pour une application donnée, une et une seule fonction ou routine peut s'appeler `toto` pour tout un programme, en C++ la portée d'un identificateur dépend du contexte d'appel.

Imaginons un autre type d'objet géométrique, le rectangle, qui disposerait des mêmes méthodes que le cercle :

```
class Rectangle
{
    // Methodes utilisateurs
public:
    void    Move(float deltax, float deltay);
    void    Zoom(float scale);
    float   Area();

    etc.
```

Dans le code utilisateur suivant :

```
    Cercle c1;
    Rectangle r1;

    c1.Zoom(3.5);
    r1.Zoom(4.0);
```

le compilateur lève les ambiguïtés de nom selon le contexte d'appel, l'écriture `c1.Zoom` désigne la méthode `Zoom` de la classe `Cercle`, appliquée à l'objet `c1`, alors que `r1.Zoom` désigne la méthode `Zoom` de la classe `Rectangle`, appliquée à `r1`.

La syntaxe d'implémentation des méthodes de classes (cf. 2.6), est d'ailleurs parlante, les points d'entrée sont `Cercle::Zoom` ou `Rectangle::Zoom`.

## 3.2 Surcharges de sélection

(L'appelation *surcharge de sélection* est propre à ce guide, voir la Note 3.1 en fin de chapitre.)

Un second mécanisme, très puissant, permet d'implanter dans une même classe plusieurs méthodes de même nom, dès l'instant que la liste d'arguments d'appel, nombre et nature, peut lever l'ambiguïté.

Par exemple :

```
class Machin
{
    void Calcul(int);            // Interface 1 entier, Ci
    void Calcul(int, int);       // Interface 2 entiers, Cii
    void Calcul(float, float);   // Interface 2 reels, Cff
    ...
```

Lors de l'utilisation, c'est le compilateur qui va choisir l'appel correct en fonction du contexte :

```
int n;
float x, y;
Machin M;

M.Calcul(n, 3);      // Appel Cii
M.Calcul(x, y);      // Appel Cff
M.Calcul(x, 1);      // Appel Cff, conversion 1 -> 1.0
M.Calcul(x, n);      // Appel Cff avec cast (float)n

M.Calcul(5);         // Appel Ci
M.Calcul(x);         // ERREUR !
M.Calcul((int)x);    // Cast explicite, appel Ci
```

Contrairement à d'autres langages où la même fonction peut exister sous tout un tas de noms différents selon les types d'arguments (voir en Fortran les `sin`, `dsin`, `imod`, `jmod`, `amod`, etc.), en C++, et sous réserve d'avoir prévu différentes configuration opératoires, on invoque tel ou tel traitement sous une appelation fonctionnelle générique en laissant au compilateur le soin de traiter les détails d'intendance !

Ce mécanisme a déjà été utilisé, sans explications, dans le cas du cercle et ses deux constructeurs, `Cercle()` et `Cercle(float)`. C'est exactement une mise en œuvre de ce mécanisme de sélection.

## 3.3 Arguments optionnels

Un dernier mécanisme permet de déclarer des méthodes avec des arguments optionnels et valeurs par défaut s'ils ne sont pas fournis lors de l'appel.

Dans l'exemple du cercle, on aurait pu n'implanter qu'un seul constructeur, celui avec un argument rayon optionnel. Si l'argument n'est pas fourni à l'appel, le compilateur utilisera la valeur par défaut.

L'interface est la suivante :

```
class Cercle
{
    Cercle(float rayon = 1.0);
    ...
```

et l'implémentation ne comporte qu'un constructeur :

```
/* Constructeur avec rayon */
Cercle::Cercle(float rayon)
{
    cX = cY = 0.0;     // A l'origine par defaut
    cR = rayon;        // Rayon utilisateur
}
```

A l'utilisation, le compilateur complètera l'appel automatiquement :

```
    Cercle c1(5.0);    // Rayon initial 5.0 explicite
    Cercle c2;         // c2(1.0) implicite
```

Dans cet exemple, l'intérêt est une simplification de l'implémentation, une seule méthode au lieu de deux.

- Un autre type d'utilisation, très utile en pratique, concerne l'évolution de développements C++. Supposons qu'un jour la librairie d'objets géométriques, `Cercle`, `Rectangle`, etc., évolue vers une version 3D.

  La classe `Cercle` devra être revue, avec trois attributs coordonnées du centre, `cX`, `cY`, `cZ`. La méthode effectuant un déplacement relatif deviendra :

  ```
  void Move(float dx, float dy, float dz);
  ```

  Doit-on remettre à hauteur toutes les applications 2D existantes ? Pas nécessairement, on peut décider que les anciennes applications 2D travailleront dans le plan `Z = 0` ; il suffit alors d'implanter l'interface ainsi :

  ```
  void Move(float dx, float dy, float dz = 0.0);
  ```

  pour que tous les codes existants utilisant l'ancienne interface, à deux arguments :

  ```
  Cercle c1;
  float dx, dy;

  ...
  c1.Move(dx, dy);
  ```

  restent recompilables en l'état !

# Notes

**[3.1]** *Il existe une grosse ambiguïté lexicale, dans la terminologie française C++. La terminologie anglo-saxonne utilise deux vocables :* overstrike, *traduit en français par* surcharge, *et* overload, *traduit en français par ...* surcharge *!*

*Sans vouloir relancer les éternelles polémiques sur la défense de la langue française (dont je suis un fervent partisan), il n'est pas acceptable de laisser en l'état de telles dégradations sémantiques. C'est pourquoi ce guide a choisi, arbitrairement, d'utiliser l'appelation* surcharge de sélection *pour la surcharge* overstrike, *présentée ici et consistant à définir des homonymes parmi lesquels le compilateur fera son choix.*

*La surcharge* overload, *présentée plus loin dans ce manuel, consistant à redéfinir par héritage une méthode d'une classe ancêtre, sera appelée* surcharge fonctionnelle.

# 4 Héritage

Après le concept de classe le concept d'héritage est la seconde grande notion en programmation orientée objets.

Il s'agit, dans le cadre de développements importants, de construire des ensembles de classes, selon une structure arborescente qui rappelle les mécanismes de taxinomie qu'on trouve en sciences de la nature : les chevaux sont des ongulés qui sont des mammifères qui sont des vertébrés qui sont des animaux. A chaque niveau, la classe apporte ses spécificités à un héritage commun, on parlera de ses *classes ancêtres*.

Pratiquement, la conception informatique d'une arborescence d'objets est loin d'être un exercice trivial. La réflexion et l'intuition jouent un rôle important mais on échappe rarement à un processus itératif consistant à détecter, en cours de développement, un oubli de conception.

Cela n'a rien de tragique, c'est même le lot commun du concepteur, et on arrive à rectifier une conception relativement facilement. Un cas de figure très fréquent est la détection, après coup, d'un *ancêtre commun* :

## 4.1 Comportement générique

Dans le cadre de la conception d'une petite librairie de figures géométriques, reprenons l'exemple du `Cercle` (pages 9 et 10), et ajoutons une classe `Rectangle`, disposant des mêmes méthodes application, à savoir `Move`, `Zoom` et `Area`.

L'implémentation interne devra, elle, comporter deux attributs de dimensions, largeur et hauteur, à la place du rayon.

La programmation naïve, analogique, conduit à définir l'interface suivante (fichier `rectangle.h`):

```
class Rectangle
{
    // Methodes utilisateurs
public:
    void    Move(float deltax, float deltay);
    void    Zoom(float scale);
    float   Area();

    // Attributs implantation
private:
    float   cX, cY;      // Position centre
    float   cL, cH;      // Largeur, hauteur
};
```

et l'implémentation suivante (fichier `rectangle.cxx`):

```
#include "rectangle.h"     // Interface classe

/* Deplacement */
void Rectangle::Move(float deltax, float deltay)
{
    cX += deltax;
    cY += deltay;
}

/* Echelle */
void Rectangle::Zoom(float scale)
{
    cL *= scale;
    cH *= scale;
}

/* Surface */
float Rectangle::Area()
{
    return cL * cH;
}
```

(On ajouterait un ou plusieurs constructeurs pour assurer l'initialisation des attributs.)

- La chose importante à remarquer, si l'on compare les classes `Cercle` et `Rectangle`, est que le support relatif à la position est strictement le même : deux attributs `cX`, `cY`, pour la position cartésienne, implantation identique de la méthode `Move`.

  Une règle incontournable, en programmation objets comme en programmation classique, voulant qu'on n'ait jamais de code identique en plusieurs exemplaires (programmer n'est pas *couper-coller*), on se trouve devant un cas de détection a posteriori d'ancêtre commun[1] !

Un tel oubli est rarement une fatalité mais résulte plutôt d'un manque de vigilance, les réponses sont souvent enfouies dans le vocabulaire de la spécification. C'est le cas ici, l'introduction de ce paragraphe parle de la conception d'une petite librairie de figures géométriques (sic).

De fait, avant de s'attacher à des notions précises comme cercle ou rectangle (ou triangle, trapèze, etc.), il est utile de s'intéresser à ce qu'est une figure géométrique.

## 4.2 Classe de base

On construira une classe, disposant de tout le support géométrique souhaité (ici 2D), sans préjuger de l'aspect exact de cette figure. On pourra imaginer un constructeur permettant de fixer la position initiale.

Interface (fichier `figure.h`) :

```
class Figure
{
public:
    // Constructeurs
    Figure();
    Figure(float xpos, float ypos);

    // Methode(s) utilisateurs
    void    Move(float deltax, float deltay);

    // Attributs implantation
private:
    float   cX, cY;      // Position
};
```

Implémentation (fichier `figure.cxx`) :

```
#include "figure.h"     // Interface classe

/* Constructeur par defaut */
Figure::Figure()
{
    cX = cY = 0.0;
}

/* Constructeur avec position initiale */
Figure::Figure(float xpos, float ypos)
{
    cX = xpos;
    cY = ypos;
}
```

1. Ou, en langue vulgaire : ≪ Enfer ! on a du oublier quelquechose ! ≫

```
/* Deplacement */
void Figure::Move(float deltax, float deltay)
{
    cX += deltax;
    cY += deltay;
}
```

## 4.3 Spécialisation par dérivation

On construira ensuite des classes dérivées, héritant de tout le support position et ajoutant simplement leur spécificités de forme.

Ainsi, l'interface cercle (fichier `cercle.h`) devient :

```
class Cercle : public Figure    // Declaration d'heritage
{
public:
    // Constructeurs
    Cercle();
    Cercle(float xpos, float ypos, float rayon = 1.0);

    // Methodes utilisateurs
    void    Zoom(float scale);
    float   Area();

    // Attributs implantation
private:
    float   cR;       // Rayon
};
```

On remarque que cette nouvelle classe ne comporte plus rien qui soit relatif à la position géométrique, le seul attribut est le rayon.

Dans la déclaration d'héritage figure un attribut de visibilité, ici `public`. Par défaut, en C++,

```
class Cercle : Figure
{
    ...
```

le mécanisme d'héritage « privatise » tout ce qui vient des ancêtres, même ce qui était public dans la classe ancêtre. C'est une sécurité, C++ pousse la paranoïa très loin.

Par contre, le fait de déclarer l'héritage public ne rend pas `public` ce qui est `private` dans la classe ancêtre, mais se borne à conserver les visibilités en l'état. Il s'agit d'une combinaison logique des visibilités, pas d'une redéfinition.

La terminologie objets utilise les mots *superclasse* et *sousclasse*. Comme le mécanisme est arborescent, ici `Figure` est LA *superclasse* de `Cercle`, alors que `Cercle` est UNE *sousclasse* de `Figure` (`Rectangle` pourra aussi être une autre *sousclasse* de `Figure`).

L'implémentation (fichier `cercle.cxx`) devient :

```
#include "figure.h"     // Interfaces classes
#include "cercle.h"

/* Constructeur par defaut */
Cercle::Cercle()
    : Figure()
{
    cR = 1.0;   // Rayon par defaut
}

/* Constructeur avec position et rayon */
Cercle::Cercle(float xpos, float ypos, float rayon)
   : Figure(xpos, ypos)
{
    cR = rayon; // Rayon specifie
}

...
```

(Les autres méthodes, `Zoom` et `Area`, sont strictement identiques à la première implémentation, voir page 11.)

- On remarquera la syntaxe particulière des constructions. La référence au constructeur de la superclasse figure avant l'accolade ouvrante du corps de fonction.

  La notation est parlante : puisqu'un `Cercle` est, au minimum, une `Figure`, plus des spécificités, la construction d'un `Cercle` consiste d'abord à construire la partie `Figure`, ensuite à configurer les spécificités. La construction est donc effectuée du haut vers le bas.

  Enfin, dans le cas de constructeurs multiples, chaque constructeur de sousclasse peut choisir celui des constructeurs superclasse à utiliser, en ≪ routant ≫ si besoin est certains des arguments. C'est le cas ici, pour les arguments de position initiale.

  **Nota :** lorsque les classes implémentent des destructeurs, optionnels, aucun mécanisme de chaînage n'est à spécifier. C'est le compilateur seul qui assurera l'appel des destructeurs éventuels et dans l'ordre inverse de la construction, sousclasses avant superclasses.

  **Nota :** la compilation correcte d'une classe dérivée suppose que le compilateur ait la connaissance de la classe et tous ses ancêtres. On doit donc inclure tous les fichiers interface nécessaires ! Ceci peut être fait, au choix, dans les fichiers sources ou dans les fichiers headers ; dans le cas d'inclusions dans les headers, il conviendra de se protéger des inclusions multiples, fréquentes en C++. Les techniques sont les mêmes qu'en C, voir en annexe le paragraphe A.5, page 50.

## 4.4 Exploitation de l'héritage

Si l'héritage est pris en compte explicitement, pour le travail de développements d'objets, il est par contre transparent au code utilisateur :

```
Cercle c1;

c1.Zoom(3.5);        // Methode propre a la classe
c1.Move(10, 2);      // Methode heritee
```

Si l'on compare aux exemples de la page 9, on voit que la mise en œuvre est inchangée. Bien que la classe `Cercle` ne dispose pas explicitement d'une méthode `Move`, le simple fait qu'un cercle soit d'abord une figure fait que `Cercle` hérite de la fonctionnalité `Figure::Move`.

Autrement dit, une sousclasse sait faire, au minimum, tout ce que savent faire ses ancêtres. On dispose là d'un mécanisme remarquablement puissant et concis (au prix de compilateurs remarquablement complexes[2]).

De la même manière, on pourra dériver de `Figure`, des sousclasses `Rectangle`, `Triangle`, etc., toutes exploitant les mécanismes de position qui sont, on le rappelle, écrits une seule fois dans un seul fichier source, `figure.cxx`.

Cette architecture se prête à merveille aux évolutions d'un développement :

Lorsque l'on souhaitera, un jour, ajouter à ce petit environnement exemple du code graphique, il faudra implanter dans chaque classe une méthode de dessin spécifique, un `Draw()` pour un cercle est différent d'un `Draw()` pour un rectangle. Par contre, tout le code support d'attributs graphiques (couleur, épaisseur de trait, type de trait, plein, pointillé, etc.) pourra être implanté, une fois et une seule, dans la classe `Figure`.

Enfin, ce mécanisme peut se répéter indéfiniment. Toute sousclasse est utilisable comme superclasse pour une dérivation. Dans notre exemple il aurait été plus rigoureux (mathématiquement parlant) de dériver de `Figure` une classe `Ellipse` à deux attributs, `cA` et `cB`, puis de dériver de celle-ci la classe `Cercle`. Cette dernière se bornerait à assurer l'égalité des demi-axes à la construction, `cA = cB = R`.

Même remarque pour une dérivation de `Rectangle` en `Carre`.

## 4.5 Accès privilégiés

Les mécanismes de protection d'accès (cf 2.7), gérés en tout ou rien par les attributs de visibilité `public` ou `private`, manquent un peu de souplesse.

En particulier, il peut arriver que pour l'écriture d'une classe dérivée, on ait besoin d'accéder à des attributs ou méthodes non publics de la superclasse (ou d'une ancêtre de la superclasse).

La révision C++ de 1992 (ref. [5]) a introduit un troisième attribut de visibilité, `protected`, qui est un intermédiaire entre les deux autres et qui rend

2. Tant pis pour eux ! Qu'ils bossent !

accessible à tout code d'une méthode de classe dérivée tout en verrouillant l'accès depuis du code extérieur.

On pourra ainsi moduler assez finement ce qui est en accès public, dans une classe, ce qui est strictement privé, ce qui n'est pas public mais utile à certaines classes dérivées. Voir Note 4.1 en fin de chapitre.

## 4.6 Objets composites

Un objet peut tout à fait contenir, parmi ses attributs, d'autres objets. Par exemple, on pourrait imaginer une figure composite comportant deux cercles concentriques :

```
class Machin : public Figure
{
public:
    Machin(float xpos, float ypos);

private:
    Cercle aC1;
    Cercle aC2;
    ...
};
```

Par défaut, à la construction d'un objet composite, tous les sous-objets sont construits avec leur constructeur par défaut (i.e. sans arguments) s'il existe :

```
Machin::Machin(float xpos, float ypos)
    : Figure(xpos, ypos)
{
    ...
}
```

Dans cet exemple, les instances de cercle aC1 et aC2 sont initialisés via le constructeur `Cercle()`.

On peut spécifier un constructeur différent, par exemple avec des arguments, avant le corps de fonction :

```
Machin::Machin(float xpos, float ypos)
    : Figure(xpos, ypos),
      aC1(xpos, ypos, 2.0), aC2(xpos, ypos, 0.5)
{
    ...
}
```

On notera la différence d'écriture : le constructeur superclasse est invoqué par son nom générique, `Figure`, alors que les constructions des attributs, ou objets membres, utilisent le nom spécifique des instances.

Une alternative consiste à laisser les constructions par défaut puis à configurer les sous-objets explicitement :

```
Machin::Machin(float xpos, float ypos)
     : Figure(xpos, ypos)
{
     aC1.Move(xpos, ypos);
     aC1.Zoom(2.0);
     aC2.Move(xpos, ypos);
     aC2.Zoom(0.5);
}
```

Pour les destructeurs, on n'a strictement rien à faire, le compilateur C++ génére le code nécessaire pour que les destructeurs des sous-objets soient invoqués lors de la destruction de l'objet principal.

# Notes

**[4.1]** *En règle générale, au cours d'un développement, il est conseillé de jouer la « sécurité maximale » (le mode par défaut, en C++) en déclarant* `private` *tout ce qui ne concerne pas strictement l'interface publique telle que spécifiée.*

*Par la suite, on modifiera au cas par cas les visibilités en fonction des besoins de telle ou telle classe dérivée et en choisissant le meilleur mécanisme (voir également le paragraphe 5.3).*

*Surtout pour des développements importants, c'est toujours une faute de déclarer a priori tout* `public`*, uniquement pour « ne plus avoir à s'embêter ensuite avec les protections » !*

# 5 Accès, protections

LE BUG est la plaie de l'informatique ! Les causes de *bugs*[1] sont multiples, proportionnelles aux capacités d'imagination de l'être humain.

Cela dit, une des causes majeures est due aux effets de bords et pollutions de données. L'aphorisme sous-jacent est que moins on touche aux données, dans un programme, moins on risque de les modifier par inadvertance.

Le niveau de fiabilité d'un langage de programmation est directement lié aux capacités du-dit langage à cacher le plus de choses possibles et tous les langages inventés depuis quarante ans ont cherché à développer ces mécanismes, ce qui fait que le classement du moins fiable au plus fiable suit à peu près l'ordre historique :

Fortran, C, Pascal, C ANSI, C++

## 5.1 Passages d'arguments

Comme C, C++ passe les arguments d'appels par valeur (i.e. copie locale au code appelé de la valeur d'appel). Si l'argument d'appel est une variable, celle-ci n'est donc jamais modifiée dans le contexte appelant, quelles que soient les avanies que peut subir l'argument dans le code appelé.

Il existe toujours des cas de programmation où l'on a besoin de passer une *variable de retour*, pratiquement dès qu'un traitement doit fournir plus d'un résultat et que donc la fonction informatique, avec sa valeur de retour unique, n'est pas suffisante.

1. Désolé pour les puristes, je ne sais pas me résoudre à écrire : *bogue* comme le voudrait notre Druon national...

Le langage C, disposant de pointeurs de données, résoud ce problème en utilisant des adresses explicites :

```
void toto(int* arg)
{
    *arg = 10;
}

....
   /* Appel */
   int var;

   toto(&var);
```

C++ dispose des mêmes mécanismes mais également, et c'est nouveau par rapport à C, d'un *passage par référence*, analogue à ce que fait Fortran en standard, utilisant la syntaxe suivante :

```
void toto(int& arg)
{
    arg = 10;     // Et non pas:  *arg !
}

....
   /* Appel */
   int var;

   toto(var);     // Et non pas: &var !
```

L'effet est analogue au passage explicite d'adresses, mais on est affranchi des écritures d'indirection, là encore c'est le compilateur qui se débrouille.

La seule obligation se situe au niveau du prototype de l'appel. La coexistence d'un double mécanisme, par valeur ou par référence, suppose une déclaration explicite, utilisant le symbole `&` :

```
void toto(int val, int& ref);
```

**Nota :** les Pascaliennes et Pascaliens ne manqueront pas de s'exclamer :

≪ Bon sang ! Mais c'est

`procedure toto(val : integer, var ref : integer);!` ≫

▶ Attention : une donnée passée par référence n'est pas un pointeur sur donnée mais se manipule, syntaxiquement, comme une donnée locale :

```
void toto(Cercle& a, Cercle* b)
{
    a.Zoom(10);     // Invocation directe
    b->Zoom(10);    // Invocation via pointeur
}
```

```
// Appel
   Cercle c1;
   Cercle c2;

   toto(c1, &c2);
```

Même si, au final, le résultat est le même, la syntaxe est différente.

## 5.2 Protection en écriture

Un mécanisme de protection, introduit dans le C ANSI, utilise un qualifieur `const` pour indiquer que quelque chose est consultable mais ne peut être modifié.

Par exemple :

```
void toto(int *p1, const int *p2)
{
    int a;
    a = *p1;    // Correct
    a = *p2;    // Correct
    *p1 = 0;    // Correct
    *p2 = 0;    // ERREUR !
```

En développement, on devrait toujours utiliser ce mécanisme à chaque fois que c'est possible. Cela permet d'éliminer, dès la phase de compilation, un certain nombre d'écritures erronées ou suspectes.

C++ a étendu ce mécanisme aux passages par références. On peut déclarer des fonctions ainsi :

```
void toto(const int& arg);
```

Pour des arguments scalaires, c'est sans aucun intérêt, si l'on veut passer une donnée non modifiable, on la passe par valeur, c'est le défaut en C, C++.

Par contre, dans le cas d'arguments objets, seuls les passages par pointeur ou par référence sont utilisables, avec toutes les conséquences, bonnes ou mauvaises, que cela implique ! On pourra alors sécuriser du code en utilisant des pointeurs ou des références `const`, non modifiables par le code appelé.

Par exemple :

```
void toto(const Cercle& arg)
{
    ...
```

Le défaut de ce mécanisme est que, à part utiliser l'argument tel quel pour un autre appel, en général le code appelé ne pourra rien faire avec l'objet en question. Le compilateur, ne voulant pas prendre le risque de modifier quoi que ce soit de l'objet interdira toute invocation de méthode !

Sauf ...

Sauf si l'objet a été soigneusement conçu. Et c'est le rôle du concepteur de l'objet de distinguer les méthodes qui peuvent modifier l'objet de celles qui sont *sûres*.

Dans l'exemple de la classe `Cercle`, la méthode `Zoom` modifie l'attribut `cR`, le rayon, alors que la méthode `Area` se borne à faire un calcul utilisant des données de l'objet mais sans le modifier. Cette méthode peut donc être utilisée sans risques sur un objet *constant*.

On introduira le qualifieur `const` dans l'interface :

```
class Cercle : public Figure
{
    ...
    float   Area() const;
```

et également dans l'implémentation :

```
float Cercle::Area() const
{
    return 3.14159 * cR * cR;
}
```

Moyennant quoi, on pourra alors concevoir du code recevant en argument des références non modifiables :

```
void toto(const Cercle& arg)
{
    float S;

    S = arg.Area();      // Autorise, methode const

    arg.Zoom(5.0);       // ERREUR !
    ...
```

## 5.3 Codage en ligne

Les évolutions des architectures processeurs, *pipeline*, *superscalaire*, ont conduit les compilateurs à implanter des mécanismes de *codage en ligne*, consistant à compiler non plus des appels à des fonctions, mais à répliquer sur place, à l'endroit de l'appel, le code de ces fonctions. Le but est de faciliter au maximum une exécution en séquence, sans branchements, sans déroutages.

Les langages ont suivi le mouvement et le C ANSI a introduit un qualifieur `inline` permettant d'écrire des fonctions en spécifiant un codage en ligne.

En C++, on pourra ainsi implanter, directement dans le fichier interface (et non plus dans le fichier source, la définition en ligne doit être disponible lors de la compilation de tous les modules application), des méthodes (courtes) :

```
class Cercle : public Figure
{
    ...
    void Zoom(float scale);
    ...
};

inline void Cercle::Zoom(float scale)
{
    cR *= scale;
}
```

On peut faire mieux, par convention du C++, toute méthode encodée directement dans la déclaration de classe est `inline` !

```
class Cercle : public Figure
{
    ...
    void Zoom(float scale)  { cR *= scale; }
    ...
};
```

On conserve ainsi une interface « propre », de type fonction, tout en ayant les avantages en performances d'un codage direct.

- Ce mécanisme est utilisé intensivement pour implanter ce qu'on appelle des *accesseurs*, à savoir des méthodes permettant la manipulation des attributs d'un objet.

  Par exemple, on ajoute à notre petite librairie un support graphique utilisant un attribut « couleur de la figure ». Cet attribut est un code numérique, index dans une palette par exemple.

  Le code application doit pouvoir définir l'index et accéder à sa valeur courante :

```
class Figure
{
    ...
    // Support couleur
private:
    int   iColor;
public:
    void  SetColor(int color)  { iColor = color; }
    int   GetColor() const     { return iColor; }
};
```

On a déjà signalé l'intérêt d'encapsuler les données, au paragraphe 2.7, l'accès `inline` permet d'interdire à quiconque la manipulation explicite de l'attribut, déclaré `private`, tout en ayant les mêmes performances en exécution.

Il n'existe AUCUNE (mauvaise) raison d'implanter des attributs publics !

Le jour où, pour telle ou telle raison, la modification du code couleur nécessitera des traitements auxiliaires (reconfiguration à faire au niveau de l'interface graphique utilisée), il suffira de remplacer la méthode en ligne `SetColor` par une méthode encodée de manière classique, dans le fichier source `figure.cxx` et effectuant tout le travail nécessaire.

Vu de l'extérieur, les codes applicatifs qui utilisent la méthode `SetColor` seront inchangés (à une recompilation près).

**Nota :** on notera la présence du qualifieur `const` sur la méthode `GetColor`. Celle-ci ne modifiant pas l'objet, pourra donc être utilisée à partir d'une référence constante.

- Enfin, ce mécanisme permet de moduler très subtilement les accès. On peut imaginer une classe, comportant un attribut consultable par tout le monde mais modifiable uniquement par la classe ou les classes dérivées, mais non par du code externe :

```
class Machin
{
    // On ne touche pas !
private:
    int  iTruc;

    // Tout le monde peut lire !
public:
    int  GetTruc() const    { return iTruc; }

    // Les sousclasses seules peuvent modifier
protected:
    void SetTruc(int truc)  { iTruc = truc; }
```

## 5.4 Touche à mon pote !

Dans certains cas, on pourra avoir besoin d'accéder à des membres ou méthodes privés ou protégés d'une classe, depuis une autre classe (non dérivée, sinon l'attribut `protected` suffit) ou depuis une fonction externe.

On déclarera alors, dans l'interface de classe, telle ou telle classe amie ou telle ou telle fonction amie :

```
class Truc
{
    friend class Pote;              // Classe amie
    friend void  toto(Truc& arg);   // Fonction amie
private:
    int mon_attrib;
};
```

Moyennant quoi, tous les accès sont possibles, sur des objets de la la classe `Truc`, depuis le code de la fonction `toto` :

```
void toto(Truc& arg)
{
   arg.mon_attrib = 0;   // Attribut prive !
   ...
}
```

ou depuis n'importe quelle méthode de la classe `Pote`, puisque l'ensemble de la classe est amie.

Ce mécanisme casse TOUS les autres types de protection ! Il ne doit donc être utilisé que dans certains cas bien précis et pour contourner certaines rigidités d'implantation. Un exemple de tels cas est présenté en annexe C, page 62.

**Nota :** la déclaration amie doit figurer obligatoirement dans l'interface de classe. C'est voulu ! Le concepteur d'une classe est seul responsable de sa sécurité et les fichiers source et *header* lui appartiennent. C'est le concepteur et lui seul qui décide des visibilités, des amis.

## 5.5 Conclusion

Ce chapitre avait pour but de détailler tous les mécanismes de protection disponibles en C++.

Il faut casser un mythe : tout cela n'est pas QUE du détail !

C'est sans aucun intérêt tant que l'on se limite à la programmation d'un « Hello world ! », et lorsqu'un développement, commencé tout petit, fini par devenir énorme (plusieurs dizaines ou centaines de classes, imbriquées dans des arborescences complexes, possédant chacune des dizaines de méthodes) il est trop tard pour « commencer à penser à la fiabilité du code » !

Un concepteur de classe doit cultiver sa paranoïa, une bonne charte de développement pourrait être :

- a priori, tout est verrouillé, `private`, attributs et méthodes.
- toute méthode qui laisse l'objet intact est déclarée `const`
- l'accès à un attribut se fera par un accesseur de type `GetXXX`, qui est toujours `const`
- le positionnement d'un attribut se fera par un accesseur de type `SetXXX`.
- les visibilités des méthodes (ou accesseurs) seront `public` uniquement si elles font partie de l'interface de spécification de la classe, sinon `private`
- on assouplira la règle précédente en changeant la visibilité `private` en `protected` pour les méthodes qui s'avèrent nécessaires à l'implantation de classes dérivées

Pour être honnête, il faut dire que le *zéro bug* n'existe pas, mais c'est une asymptote et tout doit être tenté pour s'en rapprocher, au moins en ce qui concerne les effets de bords et les pollutions. Pour les *bugs* à caractère algorithmique, il n'existe pas d'aide spécifique langage.

# 6 Polymorphisme, abstraction

Le polymorphisme est le troisième grand concept de la programmation objets. L'idée de base est de pouvoir manipuler des objets dont on ne connait pas la nature exacte, plus précisement dont on connait simplement un ancêtre réel, on parlera de *classe support*, ou un ancêtre théorique, sans existence informatique[1], on parlera alors de *classe abstraite*.

En poussant plus loin, ce concept permet d'écrire aujourd'hui du code qui manipule des objets connus et des objets inconnus, qui ne seront développés que dans le futur, et ce sans même avoir à recompiler le code en question.

On peut poser le problème sur l'exemple suivant, utilisant une petite librairie de figures géométriques[2].

On se propose de gérer un groupe de plusieurs figures, des cercles, des rectangles, et pouvoir effectuer des opérations d'ensemble : déplacer tout le monde, zoomer tout le monde, calculer un barycentre à partir des surfaces, etc.

De plus, on souhaite que le travail en question reste valide, lorsque dans quelques mois, la librairie se sera enrichie d'autres figures, triangles, trapèzes, et tout ce qui aura été imaginé d'ici là et qu'on ignore aujourd'hui.

La technique la plus classique, pour implanter un groupe quelconque, passe par une table de pointeurs d'objets qu'on crée dynamiquement et qu'on remplira ensuite :

```
int   total = 20;           // Nombre d'objets
XXX** table;                // Table des pointeurs

table = new (XXX*)[total];  // Allocation
remplir(table, total);      // Remplissage, ouf !
```

1. C'est un *must* informatique que d'écrire des programmes qui manipulent des choses qui n'ont pas d'existence informatique !
2. Ça alors ! On s'y attendait si peu...

Tout cela est bel et bon, à un détail près : `XXX` ? Une table de pointeurs de quoi ?

## 6.1 Compatibilité hiérarchique

C++ assure une compatibilité ascendante des objets, à savoir que tout objet, connu par pointeur ou par référence, est utilisable en lieu et place d'un de ses ancêtres, puisque tout objet *est, au minimum,* une de ses superclasses.

Si une classe `Cercle` dérive d'une classe `Figure`, on peut écrire des choses comme :

```
Figure* PF = new Cercle;
```

L'inverse est évidemment faux :

```
Cercle* PC = new Figure;
```

Le compilateur C++ rejettera une telle écriture, et même si l'on voulait tricher en utilisant un `cast` :

```
Cercle* PC = (Cercle*)(new Figure);
```

la sanction sera immédiate en exécution[3].

Ceci nous permet donc de manipuler tout ce qui dérive d'un ancêtre commun, ici des `Figure` :

```
int   total = 20;              // Nombre d'objets
Figure** table;                // Table des pointeurs

table = new (Figure*)[total];  // Allocation
remplir(table, total);         // Remplissage, ouf !
```

et le code de remplissage pourra placer ce qu'il veut :

```
void remplir(Figure** table, int total)
{
    table[0] = new Cercle;
    table[1] = new Rectangle;
    table[2] = new Machin;
    ...
```

(`Machin` est une `Figure`, cf. page 27 !)

Disposant d'une table de pointeurs d'objets, on est maintenant capable de déplacer l'ensemble des figures, par une simple boucle :

```
int ii;

for(ii = 0; ii < total; ii++) table[ii]->Move(150, 10);
```

Ceci exploite le fait que le déplacement est un héritage de la superclasse `Figure`, laquelle dispose de la méthode `Move`, même si, dans les faits, nos

3. ≪ Mettez à un lapin la peau d'un chacal, vous ne ferez pas croire que c'est un chacal à un autre chacal ! ≫ (Proverbe touareg).

pointeurs sont des pointeurs de `Cercle` ou de `Rectangle`, ou de...

Par contre, la même technique n'est plus utilisable si l'on souhaite appliquer un facteur d'échelle au groupe : la classe `Figure` ne dispose pas de méthode `Zoom`, et pour cause, la description géométrique d'une figure est indéfinie.

## 6.2 Méthodes virtuelles, classes support

La première technique consiste à implanter une méthode *virtuelle* dans la superclasse :

```
class Figure
{
     ...
     virtual void Zoom(float scale);
     ...
```

Cette méthode pourra être codée dans le module source d'implémentation :

```
void Figure::Zoom(float scale)
{
}
```

ou mieux (vu le volume de code mis en œuvre !), implantée en ligne dans l'interface :

```
class Figure
{
     ...
     virtual void Zoom(float scale) { }
     ...
```

Une méthode *virtuelle*, i.e. déclarée avec un qualifieur `virtual`, est une méthode de classe qui pourra être remplacée complètement, *surchargée*, par toute méthode de même nom et même interface d'appel (arguments) d'une classe dérivée. Ce mécanisme de remplacement se fait lors de la création de la classe dérivée.

Cette seconde surcharge, *overload* en anglais, que nous appelons *surcharge fonctionnelle* ici, est très différente de la surcharge de sélection, détaillée au paragraphe 3.2. La première consistait à proposer au compilateur diverses variantes d'une méthode, en fonction des types d'arguments.

Ici, si l'on écrit maintenant une boucle de zoom :

```
     int ii;

     for(ii = 0; ii < total; ii++) table[ii]->Zoom(8);
```

le compilateur ne connait qu'une méthode `Zoom` dans la classe `Figure`. Il n'empêche que, si le pointeur de `Figure` en `table[0]` est en réalité un pointeur de `Cercle`, c'est bien le code `Cercle::Zoom` qui sera exécuté. Si `table[1]` est en réalité un pointeur de `Rectangle`, c'est le code `Rectangle::Zoom` qui sera exécuté.

On dispose là d'un mécanisme puissant permettant, à une sousclasse de remplacer, lors de sa construction, une méthode d'une de ses classes ancêtres, par une autre mieux adaptée à sa nature. Les seules contraintes sont que la classe ancêtre, qui devient ainsi une *classe support* (support de surcharge), autorise le remplacement en déclarant sa méthode `virtual`, et que la méthode de classe dérivée ait strictement la même interface.

En effet, une classe dérivant de `Figure`, qui déclarerait une méthode :

```
void Zoom(int scale);
```

créerait une surcharge de sélection, et non plus une surcharge fonctionnelle.

**Nota :** il est inutile de redéclarer `virtual` la méthode surchargée dans les classes dérivées, la déclaration dans la classe de base suffit pour toutes les dérivations possibles.

**Nota :** l'implantation ci-dessus, si elle est correcte, n'est pas des plus heureuses : on déclare une méthode fictive, sans objet sur la classe en question, et on l'implante par un code vide pour ensuite la remplacer par autre chose de plus intelligent !

De plus, et si par exemple la fonction de remplissage voulait s'amuser à créer des `Figure` :

```
void remplir(Figure** table, int total)
{
    table[0] = new Cercle;
    table[1] = new Figure;
    ...
```

Rien ne l'interdit et on aura un fonctionnement stupide !

Pratiquement, la surcharge fonctionnelle sert à implanter dans une classe de base des algorithmes généraux, utilisables tels quels dans la majorité des cas. À charge pour une classe dérivée particulière, qui nécessite un traitement non standard, de surcharger avec son propre algorithme spécialisé.

De plus, il est possible dans une surcharge d'utiliser le code de la méthode originale de la classe ancêtre, s'il présente un intérêt :

```
class Parent
{
    ...
    virtual void Hello(int world);   // Methode virtuelle
    ...
class Enfant : public Parent         // Heritage
{
    ...
    void Hello(int world);           // Surcharge
    ...
```

Si l'on appelle la méthode `Hello` sur un objet `Enfant`, ou même, par polymorphisme, à partir d'un pointeur de `Parent` qui est en réalité un `Enfant`, c'est bien la méthode `Enfant::Hello` qui est invoquée.

Cette méthode peut utiliser explicitement le code de sa superclasse :

```
void Enfant::Hello(int world)
{
    Parent::Hello(world);  // Appel explicite
    ...                    // Complements
}
```

La syntaxe `Parent::Hello` s'appelle *résolution de portée*. En effet, par défaut, un identificateur de méthode ou d'attribut s'applique à la classe courante et écrire un appel à `Hello` dans la méthode `Enfant::Hello` va provoquer un appel récursif infini vite suivi du désagréable :

```
Bus error !
```

## 6.3 Classes abstraites

Notre problème peut se traiter de manière plus élégante, et plus réjouissante intellectuellement parlant !

On veut disposer d'une méthode `Zoom` dans la classe `Figure` pour pouvoir zoomer nos objets, connus par pointeurs, comme si c'était des `Figure`. Donc méthode virtuelle.

Mais on sait qu'il s'agit d'un artifice puisqu'une `Figure` ne se zoome pas. Un `Cercle` oui, un `Rectangle` oui, mais pas une `Figure`. On va alors préciser que la classe `Figure` possède bien une méthode `Zoom`, mais qu'en fait, cette méthode n'existe pas[4] !

Ce qui, en langue C++, se dit :

```
class Figure
{
    ...
    virtual void Zoom(float scale) = 0;
    ...
```

La syntaxe est rude, méthode = 0, aucun doute le C++ est bien un digne héritier de C ! Elle n'existe pas, ce qui ne l'empêche pas d'être `virtual`[5].

On vient de créer une *classe abstraite*. Une classe abstraite sait faire (potentiellement) tout un tas de choses, possède telle ou telle méthode, mais si une au moins des méthodes de la classe n'existe pas, la classe toute entière ne peut avoir d'existence légale.

On ne pourra jamais instancier, directement ou indirectement, une telle classe :

```
    Figure F1;         // ERREUR
    Figure* PF;
    PF = new Figure;   // ERREUR
```

Par contre, on peut la dériver, et si la ou les méthodes vides sont virtuelles, et surchargées par les sousclasses, ces sousclasses deviennent tout à fait instanciables.

4. Jusque là, tout le monde suit ?
5. Tout le monde suit toujours ?

Grâce à ce mécanisme, on n'a plus, comme dans la première implantation, à encoder une méthode qui ne fait rien, on ne court plus le risque d'utiliser un objet `Figure` puisque toute création est devenue impossible, et on conserve l'intérêt de manipuler nos objets sous une interface banalisée, le pointeur de `Figure`.

Enfin, lors de l'ajout de nouveaux objets figures, dans la librairie, il faudra évidemment remettre à hauteur le code de la fonction de remplissage si l'on souhaite créer aussi des `Triangle` ou autres, mais le code de manipulation de la table, boucles `Move`, `Zoom`, et autres, restera inchangé et même, s'il a été codé dans un module séparé, n'aura même pas à être recompilé !

## 6.4 Familles polymorphiques

Les mécanismes de surcharges fonctionnelle et d'abstraction sont généralisables sur toute une arborescence. La conception de grosses applications n'est pas un exercice trivial et déborde du cadre de ce manuel d'introduction.

Certaines applications peuvent nécessiter la conception d'une classe ancêtre, abstraite. Cette classe est ensuite dérivée en sousclasses qui commencent à implanter telles ou telles méthodes virtuelles, tout en restant elles-mêmes abstraites. La règle de base, pour pouvoir instancier une classe, est que toutes les méthodes de cette classe et de tous ses ancêtres existent bel et bien.

Il faut toutefois signaler un petit problème potentiel lors de la destruction d'objets polymorphiques. Par exemple, en reprenant notre exemple ci-dessus, on veut en fin d'exécution détruire la table et tout ses objets :

```
int ii;

// Destruction des objets
for(ii = 0; ii < total; ii++) delete table[ii];

// Destruction de la table elle-meme
delete[] table;
```

Le problème est que le compilateur va détruire des objets connus via des pointeurs de `Figure`. Il n'a aucun moyen de savoir qu'en réalité il s'agit de `Cercle`, de `Rectangle`, ou même de classes qui ont été crées par la suite et qui n'existaient pas au moment de la compilation.

Si une classe dérivée a besoin de faire un traitement spécifique dans son destructeur, celui-ci ne sera jamais appelé.

Sauf... si l'on utilise le mécanisme de surcharge fonctionnelle pour le destructeur[6] !

▸ Une règle, très générale, est que toute classe destinée à servir de support polymorphique doit implémenter un destructeur virtuel, même vide ! Ceci garanti un fonctionnement correct lors de la destruction de classes dérivées :

6. Bon sang ! Mais c'est bien sûr !

```
class Figure
{
public:
    // Constructeurs
    Figure();
    Figure(float xpos, float ypos);

    // Destructeur vide, en ligne
    virtual ~Figure()  { }
```

**Nota :** certains compilateurs sympathiques (`gcc` en particulier) pousseront la courtoisie jusqu'à signaler par un *warning* que telle classe possède des méthodes virtuelles mais que son destructeur ne l'est pas.

# A Compléments

CETTE ANNEXE PRÉSENTE, un peu pêle-mêle, un certain nombre de particularités C++. Ces points n'ont pas été évoqués dans les chapitres précédents pour ne pas encombrer une première lecture, mais il faut les connaitre, on en a parfois besoin.

## A.1 Membres statiques

Contrairement à un membre de classe, attribut ou méthode, qui n'a de sens que relativement à une instance réalisée de cette classe, un membre statique existe toujours, en un seul exemplaire dans le cas d'un attribut, même en l'absence totale d'instances de cette classe.

Il s'agit, en quelque sorte, d'un *patrimoine commun* à toutes les instances de cette classe.

▶ Pour illustrer ceci, imaginons d'implanter dans une classe un système de comptage d'instances. Le code application souhaite pouvoir connaître, à tout moment, le nombre d'objets réalisés de cette classe. Ce compteur doit être unique, bien sûr, et non dupliqué dans chaque instance. D'autre part il doit exister et être accessible même si aucune instance de la classe n'est disponible (on voudra savoir s'il n'existe, à un moment donné, aucune instance).

Dans l'interface de classe, on implantera un compteur statique, et un accesseur public, également statique :

```
class Truc
{
    // Constructeur, destructeur
public:
    Truc();
    ~Truc();

    // Comptage
private:
    static unsigned  iTotal;
public:
    static unsigned  GetTotal()  { return iTotal; }
    ...
```

ATTENTION : le qualifieur `static`, dans un tel contexte C++, n'a aucun rapport avec le `static` du langage C ! Une bonne idée aurait été d'inventer un autre mot-clé, cela n'a pas été fait.

Implémentation :

```
/* Attribut(s) statique(s) */
unsigned Truc::iTotal = 0;

/* Constructeur */
Truc::Truc()
{
    ...
    iTotal++;
}

/* Destructeur */
Truc::~Truc()
{
    iTotal--;
}
```

On remarquera que, contrairement à un attribut « normal », un attribut statique s'implémente explicitement. C'est, en fait, une variable globale, dont la durée de vie est celle du programme, et qui DOIT être initialisée correctement à la déclaration.

Les méthodes statiques, elles, s'implantent de la manière classique, codage explicite dans l'implémentation ou codage en ligne dans la déclaration (c'est le cas ici).

Le mécanisme ci-dessus est parfaitement sûr. La variable est privée, et ne peut être modifiée que par création ou destruction d'une instance de cette classe (ou d'une classe dérivée). (Ne pas oublier, si l'on implante plusieurs variantes de constructeurs, d'incrémenter le compteur dans chaque constructeur !)

L'accès public peut se faire de deux manières. Si l'on dispose d'une instance « sous la main », on effectue une invocation classique :

```
Truc t1;
...
unsigned total = t1.GetTotal();
printf("Il existe %u Trucs !\n", total);
```

Sinon, on peut utiliser le mécanisme (déjà vu, voir page 41) de *résolution de portée* :

```
unsigned total = Truc::GetTotal();
```

**Nota :** cet exemple est assez souvent mis en œuvre, au moins lors de développements importants. Lorsqu'une application manipule des objets par centaines, en gestion dynamique à travers `new` et `delete`, une bonne idée est de vérifier, juste en fin de programme, qu'on n'a pas oublié des `delete` ici ou la !

Ce n'est pas gênant en soi puisque, lors de l'arrêt d'un processus, toute la mémoire est libérée, mais c'est le symptôme d'une programmation malpropre quelque part ou d'une erreur de conception.

## A.2 Résolution de portée

La résolution de portée a une dernière utilisation, pour lever des ambiguïtés.

Imaginons une classe comportant une méthode nommée `sqrt` :

```
class Toto
{
    ...
    double   sqrt(double x);
    double   calcul(double x);
```

Que se passe-t-il si, dans une méthode de cette classe, par exemple la méthode `calcul`, on veut utiliser la fonction racine carrée de la librairie C ? On peut d'abord considérer que c'est bien fait pour nous, on n'avait qu'a réfléchir et choisir un autre nom pour notre méthode !

On peut aussi utiliser la résolution de portée, sans nom de classe, puisque `sqrt` de la librairie mathématique C n'est pas une méthode de classe :

```
double Toto::calcul(double x)
{
    double y;

    y = sqrt(x);        // Appelle notre methode

    y = ::sqrt(x);      // Appelle LA sqrt de la libm C
}
```

## A.3 Qui suis-je ?

Nous avons signalé, page 11, que dans une méthode de classe, par exemple :

```
void Cercle::Zoom(float scale)
{
    cR *= scale;
}
```

la référence au nom d'un attribut ou d'une méthode se rapporte à « l'objet courant pour cet appel ».

Cet objet courant existe bel et bien, sous la forme d'un pointeur sur l'instance appelé : `this` (mot réservé en C++).

Ce pointeur sert peu, en pratique, puisque implicitement, écrire :

```
    cR *= scale;
```

revient à écrire :

```
    this->cR *= scale;
```

Il faut toutefois savoir qu'il existe, on l'utilise dans certains cas où depuis le code d'une méthode, on aurait besoin d'appeler « sur nous » une fonction externe à la classe, prenant un pointeur d'objet en argument. Un autre cas d'utilisation est signalé dans l'annexe C, page 60.

## A.4 Structures

Pour des raisons de compatibilité indispensable, C++ supporte les structures de données du langage C.

Comme les structures, au sens du C, n'ont aucun intérêt dans un langage objets, en C++ les structures sont en fait des classes !

Syntaxiquement, cela se traduit par le fait qu'on peut déclarer des données de type structure comme on instancie des classes, le déclarateur `struct` est facultatif :

```
struct Toto {          // Definition structure C
   int a;
   int b;
   };

   struct Toto S1;     // Declaration avec "look" C
   Toto S2;            // Declaration avec "look" classe C++
```

En fait, en C++, une structure est une classe où tous les champs, les membres, sont accessibles.

On aurait le même effet avec :

```
class Toto
{
public:
   int a;
   int b;
};

   Toto S1;
   S1.a = 0;
   S1.b = 1;
```

(On rappelle que, par défaut, dans une classe et sans spécification de visibilité tout est `private`.)

- Comme il s'agit en fait de classes, il devrait donc être possible d'implanter dans des structures des méthodes, un constructeur, etc. :

```
struct Toto {
   Toto() { a = b = 0; }  // Constructeur en ligne
   int a;
   int b;
   };
```

En effet, c'est possible. L'écriture ci-dessus est tout à fait valide. Et ça n'a AUCUN INTÉRÊT !

Ou bien on porte du code C existant dans une application C++, auquel cas on laisse ce qui existe déjà et le code C n'utilisait sûrement pas de méthodes de structures, ou alors on fait un nouveau développement C++ et on utilise des classes, uniquement. La structure, en C++, est un outil de compatibilité, pas un outil de développement.

# A.5 Gestion des déclarations

## Prédéclarations

Assez souvent, dans un développement important, on se trouve confronté à des problèmes d'imbrications de déclarations.

Par exemple, une classe `Toto` possède des attributs qui sont des pointeurs de classe `Truc`, et la classe `Truc` possède des pointeurs de `Toto`. Pour pouvoir compiler correctement, C++ autorise des *prédéclarations* de classes (*forward declarations* dans la terminologie anglaise).

Fichier interface `toto.h` :

```
class Truc;           // Forward

class Toto
{
private:
    Truc*  monTruc;   // Attribut pointeur
};
```

Fichier interface `truc.h` :

```
class Toto;           // Forward

class Truc
{
private:
    Toto*  monToto;   // Attribut pointeur
};
```

On peut remarquer que, lors de l'inclusion des *headers* dans un module source qui a besoin des deux interfaces de classes :

```
#include "toto.h"
#include "truc.h"
```

l'une des prédéclarations devient en fait une postdéclaration. C'est toléré, à savoir que le compilateur ignore une prédéclaration s'il a déjà vu l'interface de classe.

## Inclusions multiples

Dans un gros développement, plusieurs dizaines de classes, on se trouve vite à la tête de multiples fichiers *headers* (au minimum un par classe, parfois plus).

Souvent des modules sources ont besoin d'inclure de multiples fichiers, lequels peuvent comporter aussi des directives `#include`, et il n'est pas rare qu'un même fichier puisse se trouver inclus plusieurs fois (indirectement).

Ça, par contre, c'est strictement interdit, en C++ comme en C ! Il faut donc systématiquement protéger tous les fichiers *headers* contre l'inclusion multiple. La technique est classique, chaque fichier définit un symbole en rapport avec son nom, et n'implante son code que si c'est la première lecture. Exemple `toto.h` :

```
#ifndef TOTO_H     // TOTO_H n'existe pas, 1ere fois

class Toto
{
   ...
};

#define TOTO_H     // Il existera la prochaine fois !
#endif
```

# B Compatibilité C/C++

LE LANGAGE C++ est une extension, orientée objets, du C ANSI. Les deux dialectes sont suffisament proches pour que beaucoup de compilateurs soient capable de traiter les deux. On parle de compilateurs mixtes, C/C++.

On dira que C++ est un surensemble du C ANSI dans la mesure où toute la syntaxe du C ANSI est valide en C++. Et donc, tout programme écrit en C ANSI peut être compilé avec un compilateur C++. Le C ANSI est impératif, la syntaxe ancienne du C K&R, sans prototypes de fonctions, est rejettée en C++.

L'inverse est évidemment faux, un source C++ ne sera pas, en général, compilable par un compilateur C. Sera rejetté tout ce qui concerne les classes, mais également, dans des écritures de fonctions classiques, les passages par références inconnus en C et les surcharges de sélection : C n'autorise pas plusieurs fonctions de même nom, même si les arguments d'appel diffèrent.

## B.1 Points d'entrée

Un premier problème va apparaître lorsque l'on voudra mélanger, pour un même programme, des modules écrits et compilés en C avec des modules écrits et compilés en C++.

En C (comme en Fortran ou Pascal), le nom d'une fonction est un identifiant global pour tout un programme :

```
void toto(int n);
```

Au moment de l'édition des liens, une et une seule fonction `toto` doit exister dans l'ensemble des modules objets.

C++ supporte la surcharge de sélection :

```
void toto(int n);
void toto(int i, int j);
void toto(float x, float y);
```

et les méthodes de classes :

```
class Machin
{
public:
    void toto(int a);
    ...

class Truc
{
public:
    void toto(int a);
    void toto(float x, float y);
    ...
```

De fait, la portée symbolique d'un identificateur C++ est propre au contexte d'invocation. Pour implémenter cela, les compilateurs disposent d'un algorithme interne de génération de noms (appelé *name mangling* dans la terminologie anglaise). Les points d'entrée d'exécution, i.e. les noms qui seront utilisés à l'édition de liens pour résoudre les appels, sont entièrement reconstruits.

Un mécanisme classique consiste à prendre le nom de la fonction ou de la méthode, précédé du nom de la classe si c'est une méthode, et suivi de codes symboliques décrivant les arguments d'appel. Par exemple, tous les `toto` ci-dessus pourraient être renommés en interne :

```
_toto_i
_toto_ii
_toto_ff
Machin_toto_i
Truc_toto_i
Truc_toto_ff
```

Cet algorithme est ensuite utilisé lors de la compilation des fonctions ou méthodes, mais également lors des appels :

```
    Machin M;
    Truc T;
    float x, y;
    ...
    toto(x, y);          // Appel _toto_ff(x, y)
    M.toto(3);           // Appel Machin_toto_i(3)
    T.toto(1.5, 0.0);    // Appel Truc_toto_ff(1.5, 0.0)
```

En conclusion, rien de magique, tout cela n'est qu'une cuisine interne de compilation. Deux problèmes potentiels existent :

**1**· l'algorithme d'encodage de noms est propre aux compilateurs, rien n'est standardisé aujourd'hui ! Et donc, une même méthode de classe, écrite en syntaxe C++ :

```
void Truc::toto(int a);
```

pourra conduire à un encodage interne :

```
Truc_toto_i
```

avec tel compilateur, ou :

```
_DEC$CXX$Truc$toto$I
```

avec tel autre.

Il est donc fortement déconseillé de lier, dans une même application, des modules objets compilés avec des compilateurs C++ différents ! On en choisi un, sur une plateforme, et on s'y tient !

**2·** il est en général impossible d'appeler une fonction écrite en C depuis du code C++, ou inversement. Par exemple on veut, dans du code C++, utiliser une ou des fonctions de la librairie mathématique C :

```
double sqrt(double);     // Prototype

double x;
x = sqrt(4.0);           // Appel _sqrt_f(4.0) !!!
```

et à l'édition de liens, le linker ne va pas trouver le point d'entrée `_sqrt_f` et pour cause, dans la librairie mathématique, compilée en C, l'entrée s'appelle `sqrt` !

Ce second problème est réel et pour cela, C++ permet d'inhiber l'encodage des noms et de compiler avec des points d'entrée aux normes du C. On peut inhiber le *name mangling* fonction par fonction :

```
extern "C" void toto(int a);

extern "C" float truc(float x, float y);
```

ou pour tout un groupe de prototypes :

```
extern "C" {
   void toto(int a);
   float truc(float x, float y);
}
```

Il faut connaitre et utiliser ce mécanisme, très important. Depuis déjà une dizaine d'années, avec le déploiement de C++, toutes les librairies C sous Unix sont disponibles pour C++ et donc les *headers*, `stdio.h`, `stdlib.h`, `math.h`, etc., comportent ces déclarations `extern "C"`.

Quiconque développe en C, utilitaires, librairies, devrait fournir des fichiers de prototypes conformes à cette règle. Petit détail, la déclaration `extern "C"` est du C++ et sera rejettée par un compilateur C. C'est pourquoi TOUT fichier *header* prototypant du C doit respecter les écritures suivantes :

```
#ifdef __cplusplus
extern "C" {
#endif

/* Prototypes C */
void toto(int a);
...
```

```
#ifdef __cplusplus
}
#endif
```

Le symbole `__cplusplus`, standard, est prédéfini par tous les compilateurs C++. Moyennant cette précaution, le *header* est compatible C ANSI, C++.

(De plus, on s'interdira l'utilisation du commentaire ligne `//`, spécifique C++, dans des *headers* mixtes.)

# B.2 Interface objets

Il est donc possible d'écrire des fonctions C++, appelables depuis du code C, sous réserve d'utiliser la déclaration `extern "C"` présentée ci-dessus.

Il n'est pas possible d'interfacer directement des objets, C n'étant pas un langage objets. Par contre, C manipulant des pointeurs, il est tout à fait possible d'imaginer une interface *par pointeurs*.

A titre d'illustration, on se propose d'interfacer notre petite librairie de figures géométriques, dans sa version polymorphique (chapitre 6), avec du code C. On utilisera un type générique « pointeur de figure », défini différemment selon le langage cible (cf. ci-dessous) et, pour la création, des codes symboliques spécifiant la nature de l'objet à créer.

## Header interface

(Fichier `libfig.h`)

```
#ifdef __cplusplus
typedef Figure* figure;  /* Pointeur de figure pour C++ */
extern "C" {

#else
typedef void* figure;    /* Pointeur quelconque pour C  */
#endif

/* Types figures */
#define FIG_CERCLE      1
#define FIG_RECTANGLE   2
 ... etc

/* Interface */
figure figCreate(int type);
void   figDelete(figure F);

/* Methodes */
void  figMove(figure F, float dx, float dy);
void  figZoom(figure F, float scale);
float figArea(figure F);

#ifdef __cplusplus
```

```
}
#endif
```

## Source interface

(Fichier `libfig.cxx`)

```
#include "figure.h"         // Classes
#include "cercle.h"
#include "rectangle.h"

#include "libfig.h"         // Interface C

/* Constructeur */
figure figCreate(int type)
{
    switch( type ) {
    case FIG_CERCLE    : return new Cercle;
    case FIG_RECTANGLE : return new Rectangle;
    default : return (Figure*)0;
    }
}

/* Destructeur */
void figDelete(figure F)
{
    delete F;
}

/* Methodes */
void figMove(figure F, float dx, float dy)
{
    F->Move(dx, dy);
}

void figZoom(figure F, float scale)
{
    F->Zoom(scale);
}

float figArea(figure F)
{
    return F->Area();
}
```

Comme on peut le constater, c'est un petit travail simple, même s'il n'a rien de passionnant. On notera également toute la puissance du polymorphisme qui permet de ne s'intéresser à la nature exacte des objets que lors de la construction. Tout le reste du code est générique, pointeur de figure quelconque.

**Nota :** c'est parce que toutes les fonctions de ce module sont déclarées `extern "C"`, dans le fichier *header*, qu'elle pourront être liées aux appels venant du code C.

# C Surcharges d'opérateurs

QUICONQUE A DÉJÀ PROGRAMMÉ, en Fortran, en Pascal, en C, connait la notion d'*opérateur*. Il s'agit d'une notation symbolique spécifiant une opération à faire, un traitement, dont la nature peut varier en fonction des arguments ou opérandes.

Ainsi, dans tous les langages, l'écriture `10 / 3` spécifie une division entière dont le résultat sera `3`, alors que `10.0 / 3` spécifie une division réelle, résultat `3.3333`. C'est le compilateur qui va, à partir d'un même symbole opératoire et selon la nature des opérandes, déterminer le traitement à effectuer.

C++ permet de définir des opérations spécifiques, utilisant les notations symboliques du langage, sur des opérandes a priori inconnus des compilateurs, des objets par exemple. Le terme consacré, *surcharge d'opérateur*, est sans doute un peu abusif. On devrait plutôt parler de *définition d'opérateur* puisque cela s'applique à des opérations inconnues.

Les domaines d'application sont essentiellement mathématiques, calculs en complexes, algèbre linéaire, etc.

## C.1 Arithmétique complexe

On se propose de développer cette notion, en construisant une classe d'objets *nombres complexes*, lesquels n'existent pas en standard en C ou C++ (voir Note C.1 en fin d'annexe).

On construit des objets à deux attributs, parties réelle et imaginaire. On va choisir d'implanter trois constructeurs :

- un constructeur par défaut, classique
- un constructeur avec initialisateurs

- un constructeur dit *de copie*, permettant de créer un objet par clonage d'un autre, passé par référence.

Interface (fichier `complex.h`):

```
class complex
{
    // Constructeurs
public:
    complex();
    complex(float real, float imag);
    complex(complex& model);

    // Attributs
private:
    float pR, pI;
};
```

Implémentation (fichier `complex.cxx`):

```
#include "complex.h"

/* Constructeur par defaut */
complex::complex()
{
    pR = pI = 0.0;
}

/* Constructeur initialisateur */
complex::complex(float real, float imag)
{
    pR = real;
    pI = imag;
}

/* Constructeur cloneur */
complex::complex(complex& model)
{
    pR = model.pR;
    pI = model.pI;
}
```

On dispose donc d'une base qui va permettre d'instancier des objets de différentes manières :

```
    complex c1;
    complex c2(3.5, 0);   // Avec initialisation
    complex c3(c2);       // Clonage sur c2
    ...
```

Que peut-on faire maintenant, de ces objets ? Pas grand chose en fait ! On peut vouloir accéder aux attributs, privés, donc par un mécanisme d'accesseurs (cf. 5.3), `SetReal`, `SetImag`, `GetReal`, `GetImag`.

La technique manque un peu d'élégance et on va très vite se trouver face à du code comme :

```
// Affectation : c1 = c3
c1.SetReal(c3.GetReal());
c1.SetImag(c3.GetImag());

// Somme composite : c1 += c2
c1.SetReal(c1.GetReal() + c2.GetReal());
c1.SetImag(c1.GetImag() + c2.GetImag());
```

Tout ce qu'on peut dire de la monstruosité précédente est que cela fonctionne !

## C.2 Opérateurs sur la classe

Heureusement, C++ permet de définir des opérateurs `=`, ou `+=`, valables pour des objets de la classe `complex`. On ajoutera dans l'interface :

```
class complex
{
    ...
    // Operations
public:
    void operator=(complex& arg);
    void operator+=(complex& arg);
```

et, dans l'implémentation :

```
/* Affectation */
void complex::operator=(complex& arg)
{
    pR = arg.pR;
    pI = arg.pI;
}

/* Somme composite */
void complex::operator+=(complex& arg)
{
    pR += arg.pR;
    pI += arg.pI;
}
```

On dispose maintenant d'outils propres, permettant d'écrire du code qui ressemble à de l'arithmétique en C :

```
complex c1(1, 0);
complex c2(2.5, -1);

c2 += c1;
c1 = c2;
...
```

## C.3 Associativité

En C++ comme en C, certains opérateurs sont associatifs, en particulier l'affectation, ce qui permet d'écrire des choses telles que :

```
int i, j, k;

i = j = k = 3;
```

Cela fonctionne parce que l'affectation est une opération qui retourne son opérande de droite. Syntaxiquement, le résultat d'une affectation est donc un opérande et peut figurer dans une expression.

Modifions donc notre opérateur d'affection pour qu'il retourne son opérande de droite, à savoir une référence sur un objet. L'interface devient :

```
class complex
{
    ...
    complex& operator=(complex& arg);
```

et l'implémentation :

```
complex& complex::operator=(complex& arg)
{
    pR = arg.pR;
    pI = arg.pI;
    return arg;
}
```

Maintenant, des écritures telles que :

```
complex c1, c2;
complex c3(2.5, -1);

c1 = c2 = c3;
```

sont licites[1].

**Nota :** dans l'écriture de l'opérateur =, on retourne l'opérande de droite. Si l'on a besoin, dans un opérateur, de retourner l'opérande de gauche, c'est à dire l'objet courant, on se souviendra du pointeur `this` (cf. A.3).

Attention, `this` est un pointeur sur nous, mais ce n'est pas nous ! Nous, c'est l'objet pointé par `this`, et donc :

```
complex& complex::operator=(complex& arg)
{
    pR = arg.pR;
    pI = arg.pI;
    return *this;   // Mais oui !
}
```

1. Isn't it great ?

## C.4 Surcharge de sélection

Encore mieux[2], la surcharge permettant au compilateur de choisir telle ou telle méthode en fonction des arguments est applicable aux opérateurs.

Implantons donc la multiplication composite, `*=`, entre complexes ou entre complexe et scalaire.

On ajoute dans l'interface :

```
class complex
{
    ...
    void operator*=(float arg);
    void operator*=(complex& arg);
```

et on code, dans l'implémentation :

```
void complex::operator*=(float arg)
{
    pR *= arg;
    pI *= arg;
}

void complex::operator*=(complex& arg)
{
    float R = pR * arg.pR - pI * arg.pI;
    float I = pR * arg.pI + pI * arg.pR;
    pR = R;
    pI = I;
}
```

On peut maintenant écrire :

```
    complex c1(1, 1);
    complex c2(5, 0);

    c2 *= c1;
    c2 *= 0.5;
```

et le compilateur choisira l'opération ad-hoc en fonction des opérandes exactement comme dans le cas de l'arithmétique scalaire.

2. Mais où cela va-t-il s'arrêter ?

## C.5 Objets temporaires

Dans les exemples précédents, on a sournoisement contourné un problème en implantant des opérateurs composés, `+=` ou `*=`, mais pas des opérateurs simples, `+` ou `*`.

Que se passe-t-il dans une expression arithmétique :

```
int i, j, k;

i = 3 + j + k;
```

En pratique, le compilateur utilise une ou plusieurs variables intermédiaires, anonymes (le plus souvent des registres du CPU), pour conserver les résultats partiels des opérations et les utiliser comme opérandes pour les opérations suivantes.

L'expression ci-dessus est traitée, en interne, comme :

```
int i, j, k;

int __1, __2;    // Temporaires

__1 = 3 + j;
__2 = __1 + k;
i = __2;
```

On utilisera le même principe en créant, dans les opérateurs, un objet temporaire résultat. Par contre, on ne devra plus implanter des opérateurs internes à la classe, utilisant l'objet courant, mais des opérateurs externes à deux opérandes. Pour qu'ils puissent tout de même accéder aux attributs, ils seront déclarés `friend`.

Ajoutons une addition dans l'interface :

```
class complex
{
    ...
    friend complex operator+(complex& a, complex& b);
```

et dans l'implémentation :

```
complex operator+(complex& a, complex& b)
{
    float R = a.pR + b.pR;
    float I = a.pI + b.pI;

    complex result(R, I);   // Nouvel objet
    return result;
}
```

**Nota :** le type retour est bien un objet, `complex`, et non une référence, `complex&`. Le compilateur doit être prévenu que l'opération a créé une instance, temporaire, et qu'elle devra être détruite après utilisation, donc en fin de traitement de l'expression.

On dispose maintenant d'une véritable addition (voir Note C.2) :

```
complex c1(1, 0);
complex c2(1, 1);
complex c3(c2);

complex c4;
c4 = c1 + c2 + c3;
```

Ce mécanisme ne supportant pas, a priori, la commutativité que le compilateur ne peut pas deviner, dans le cas d'opérations hybrides on devra implanter les différentes variantes. Par exemple, pour la multiplication :

```
complex operator*(complex& a, float b);
```

et :

```
complex operator*(float a, complex& b);
```

La variante commutative peut d'ailleurs s'implanter en ligne en utilisant l'opérateur déjà disponible :

```
inline complex operator*(float a, complex& b)
{ return b * a; }
```

# C.6 Remarques

Les exemples précédents illustraient les principes de la surcharge, ou redéfinition d'opérateurs.

Tous les opérateurs de C peuvent être redéfinis, arithmétiques mais aussi booléens. On pourrait comparer des nombres complexes, par des écritures telles que :

```
complex c1, c2;
...
if( c1 == c2 ) ...
```

en implantant des opérateurs `==`, `!=`, etc.

Il est conseillé de conserver un minimum de bon sens. Même si C++ le permet, ce n'est PAS une bonne idée d'implanter une addition de complexes sur l'opérateur `*` et une multiplication sur l'opérateur `+`, d'autant que les priorités d'évaluation restent celles de l'arithmétique (`a + b * c` se traite par `a + (b * c)`).

▶ Il peut arriver qu'on ait besoin d'opérations qui n'existent pas sous forme scalaire. Par exemple, dans une application d'algèbre linéraire implémentant une classe `vecteur`, on aimerait disposer de deux opérations de multiplication, les produits scalaire et vectoriel.

Les notations symboliques, pour un mathématicien, sont $V1.V2$ pour le produit scalaire et $V1 \wedge V2$ pour le produit vectoriel. Le `.` en C, C++, est réservé à l'accès aux membres de structures ou d'objets et est donc inutilisable.

Un choix convenable consisterait à implanter le produit scalaire sur l'opérateur `*`, et le produit vectoriel sur l'opérateur `^`, le `xor` binaire de C, lequel n'a aucun intérêt dans un contexte vecteurs. On aura alors des écritures informatiques lisibles, à défaut d'être rigoureuses :

```
vecteur V1, V2, V3;
float sc;
...
V1 = V2 ^ V3;
sc = V1 * V2;
```

- Enfin, ne pas oublier que C++ ne sert pas qu'à construire des objets mais qu'on peut aussi écrire des fonctions. Le mécanisme de surcharge de sélection permettant de lever les ambiguïtés de nom à partir de la nature des arguments d'appel, on pourra, pour les besoins d'un module comme celui de ce chapitre, réimplanter des fonctions existantes en version scalaire :

```
complex sqrt(complex& arg);
```

et autres...

# Notes

**[C.1]** *En fait, c'est faux ! Les complexes en C++, s'ils ne sont pas supportés en standard par le langage, sont disponibles sous forme d'un petit* package.

*Cela existe parce que c'est utile et assez rapide à écrire, et c'est parce que c'est rapide à écrire que nous l'utilisons comme illustration du principe.*

*Ce n'est qu'une illustration, volontairement simplifiée. En particulier on a omis tout le support d'accès en lecture seule,* `const`*, qui doit normalement figurer dans tous les opérateurs !*

**[C.2]** *Derrière l'élégance apparente d'écritures comme :*

```
c4 = c1 + c2 + c3 ;
```

*il ne faut pas perdre de vue que se cache un traitement assez important : instanciations d'objets, appels des constructeurs, appels de méthodes opérateurs, calculs, destruction !*

*Dans le cas d'objets assez conséquents, par exemple une implantation d'algèbre linéaire avec toutes les opérations nécessaires implantées sur vecteurs, matrices, on peut aboutir à de véritables monstres arithmétiques.*

*C'est pourquoi, très souvent, on favorise les opérateurs d'affectation composée, plus simples à écrire et à exécuter.*

*Ainsi, on trouvera du calcul matriciel encodé comme suit :*

```
matrice M1, M2, M3, M4 ;
...
M4 = M1 ;
M4 += M2 ;
M4 *= M3 ;
```

*plutôt que :*

```
M4 = (M1 + M2) * M3 ;
```

*même si la seconde écriture est plus jolie !*

# D Les streams

DEUX OPÉRATEURS STANDARDS DE C, `>>` et `<<`, opérateurs de décalages de bits, jouent un rôle particulier en C++.

Comme ils sont assez peu utilisés, et en tout état de cause limités à des opérandes entiers, ils ont été redéfinis (voir l'annexe C) pour un tout autre usage : manipuler des entrées/sorties !

En plus de la librairie C d'entrées/sorties, `stdio.h`, C++ dispose d'une librairie, `iostream.h`, permettant des entrées/sorties plus sophistiquées. En particulier, la gestion des formatages d'arguments a été intégrée, et ces objets entrées/sorties sont interfacés par des opérateurs de transfert.

Exemple, la *sortie standard*, qui en C est une structure `stdout`, est un objet C++ `cout` :

```
#include <stdio.h>       // E/S du C
#include <iostream.h>    // E/S du C++

    float x;
    x = 3.14159;

    // Affichage a la mode C
    fprintf(stdout, "x = %g\n", x);

    // Affichage a la mode C++
    cout << "x = " << x << '\n';
```

On appréciera l'élégance de la syntaxe, on « vide[1] » vers le *stream* une suite d'entités, chaîne de caractères, réel, caractère de contrôle, etc. En particulier, la spécification de format a disparu.

Le but de cette annexe n'est pas de détailler la librairie `iostream` mais d'expliciter le mécanisme pour pouvoir l'adapter à nos propres besoins.

1. C'est la raison du choix de ces opérateurs. Aucun rapport avec des décalages de bits, mais l'effet visuel est évocateur !

Cette librairie repose sur des classes, `istream` pour les entrées, `ostream` pour les sorties. La classe `istream` surcharge l'opérateur `>>`, la classe `ostream` surcharge l'opérateur `<<`.

Par surcharge de sélection, ces opérateurs sont implantés pour tous les types d'arguments de base (les opérations de formatage en fonction de la nature, entier, réel, chaîne, ..., sont donc internes à l'opérateur) et, pour permettre l'écriture associative, ces opérateurs retournent leur opérande de gauche qui est une référence sur un `ostream` (dans le cas des sorties).

On peut compléter l'exemple de l'annexe C, construisant une classe de nombres complexes, en ajoutant par exemple un opérateur d'affichage de complexe. On peut imaginer d'imprimer les parties réelle et imaginaire, entre parenthèses, séparées par une virgule.

On ajoutera, dans l'interface :

```
class complex
{
    ...
    friend ostream& operator<<(ostream& out, complex& c);
```

et dans l'implémentation :

```
ostream& operator<<(ostream& out, complex& c)
{
    out << '(' << c.pR << ',' << ' ' << c.pI << ')';
    return out;
}
```

On utilise des types opérandes, caractères, réels, déjà existants dans la classe `ostream`.

Ensuite :

```
    complex c1(3.5, 0.5);

    cout << "c1 = " << c1 << '\n';
```

affichera :

```
c1 = (3.5, 0.5)
```

Joli non ?

**Nota :** ce n'est pas la meilleure programmation. Pour pouvoir accéder aux attributs du complexe, on doit déclarer l'opérateur `friend` ce qui oblige à adapter l'interface de classe. En pratique il serait préférable d'utiliser des accesseurs, `GetReal`, `GetImag`, disponibles dans la classe.

Ainsi, et sans devoir toucher au fichier interface de classe, n'importe quel code utilisateur pourrait implanter son opérateur de sortie personnel :

```
ostream& operator<<(ostream& out, complex& c)
{
   out << '(' << c.GetReal() << ", " << c.GetImag() << ')';
   return out;
}
```

# E Les patrons

LE DERNIER GROS APPORT DE C++, par rapport à C, concerne la possibilité de définir des *patrons* (*templates* en anglais) ou modèles. Il n'y a rien de conceptuel, c'est simplement un outil de programmation mais très puissant.

L'idée est d'éviter de devoir écrire et reécrire du code presque identique, et donc de donner au compilateur les informations nécessaires pour générer automatiquement du code. Un *patron* C++ est, en quelque sorte, du code source paramétré.

L'utilisation intensive de *patrons* est un vaste sujet, cette annexe se borne à présenter le principe.

Dans l'annexe C, on avait initié le développement d'une petite classe de nombres complexes. Ceux-ci comportaient des attributs parties entière et imaginaire, encodés en réels simple précision, `float`

Si l'on désire également travailler avec des complexes en double précision, il faudrait reconstruire une classe analogue, dupliquer les fichiers, remplacer les types `float` par des types `double`, etc. Bref, un abominable travail qui n'est plus de la programmation. Quant à la maintenance, ajout de nouvelles opérations, n'en parlons pas.

La solution c'est le *patron* de classe. On va décrire, non plus une interface de classe C++ mais un modèle pour construire des classes C++, le type des attributs étant un paramètre symbolique que l'on appellera : `TYPE` (ou `TOTO` si l'on préfére).

## Interface

Nouveau fichier interface, `complex.h`, avec le maximum de code en ligne (on précisera pourquoi ensuite) :

```
template<class TYPE>
class complex
{
    // Constructeurs
public:
    complex()
        { pR = pI = (TYPE)0; }
    complex(TYPE real, TYPE imag)
        { pR = real; pI = imag; }
    complex(complex<TYPE>& model)
        { pR = model.pR; pI = model.pI; }

    // Attributs
private:
    TYPE pR, pI;

    // Operateurs
public:
    complex<TYPE>& operator=(complex<TYPE>& arg)
                     { pR = arg.pR; pI = arg.pI;
                       return arg; }
    void           operator+=(complex<TYPE>& arg)
                     { pR += arg.pR; pI += arg.pI; }
    ... etc
};
```

L'écriture est similaire, à quelques détails près. Les attributs sont typés par le type symbolique `TYPE`, les références à une classe patron s'écrivent, ici, `complex<TYPE>` et non plus simplement `complex`. En fait, le nom de classe lui-même est paramétré.

L'utilisation d'une classe patron est immédiate, sous réserve de résoudre le paramètrage :

```
    complex<float> c1;   // Un complexe simple precision
    complex<double> c2;  // et un autre, double precision
```

et c'est tout ! Le compilateur a tout ce qu'il faut pour implanter une « vraie » classe de complexes à attributs `float` et une à attributs `double`.

**Nota :** si l'on trouve la notation un peu disgracieuse, on peut la contourner facilement :

```
typedef complex<float>  acomplex;
typedef complex<double> dcomplex;

    acomplex c1;     // Un complexe simple precision
    dcomplex c2;     // et un autre, double precision
```

## Implémentation

On peut, bien sûr, implémenter du code de patron de manière classique, dans un fichier source séparé, moyennant quelques petites précautions syntaxiques, l'écriture est un peu plus lourde que pour du codage de méthodes classique :

```
/* Constructeur initialisation */
template<class TYPE>
complex<TYPE>(TYPE real, TYPE imag)
{
    pR = real;
    pI = imag;
}

/* Somme composee */
template<class TYPE>
void complex<TYPE>::operator+=(complex<TYPE>& arg)
{
    pR += arg.pR;
    pI += arg.pI;
}
```

L'intérêt de la génération en ligne, quand elle est envisageable (quelques lignes de code), est que d'une part le compilateur génére le code sur place, et d'autre part n'utilise le patron que par fragments, selon les besoins.

Une classe comme celle de notre exemple, des complexes, pourrait comporter plusieurs dizaines d'opérateurs, avec différents types d'arguments. Si le code application n'en utilise que deux ou trois, les autres n'auront même jamais existé dans le programme !

Enfin, si tout le codage est possible en ligne, seul le fichier de déclaration, `complex.h`, existe. Plus de fichier source, plus d'édition de liens avec tel ou tel module librairie pour le code utilisateur.

## Remarques

- La puissance de cet outil est assez remarquable ! On se borne à décrire comment construire des classes, de tel ou tel type. C'est un peu un mécanisme d'apprentissage, ensuite le compilateur se débrouille (en général bien, et toujours mieux qu'un humain qui fait du couper/coller et du cherche/remplace à l'éditeur de texte) !
- De plus, il est possible de définir des patrons à plusieurs paramètres :

```
template<class P1, class P2>
class machin
{
    ...
```

et on pourra ensuite instancier par :

```
    machin<float, float> m1;
    machin<int, double>  m2;
```

```
...
```

(Voir le nombre de couper/coller évités, avec un patron à simplement 2 paramètres pouvant chacun être substitué par 3 ou 4 types différents ! Et que penser d'un patron à 10 paramètres...)

- Ce mécanisme est très utilisé pour implanter du code générique, par exemple des algorithmes de gestion de listes d'objets, ajout, insertion, recherche, etc.

  On construit un patron `list<TRUC>`, pour gérer une liste de n'importe quoi, et on instancie au fur et à mesure des besoins :

  ```
  list<int>      L1;   // Ma liste d'entiers
  list<dcomplex> L2;   // Ma liste de complexes double
  ```

  Là, on quitte le domaine du trivial, mais le travail est intéressant et surtout réutilisable à un haut degré.

- Il faut signaler que les patrons sont un outil C++ récent et que la mise en œuvre n'est pas totalement stabilisée. En particulier, lorsqu'on commence à s'écarter des utilisations simples, comme présenté ici, il peut y avoir des variantes de comportement d'un compilateur à l'autre.

  Le `gcc`, par exemple, comporte des directives spécialisées pour la gestion des patrons, `#pragma interface`, `#pragma implementation`.

  Certains compilateurs C++, c'est le cas du `CC` de SUN, créent même des bases de données de patrons dans les répertoires de développement (sous répertoire `Templates.DB` pour SUN).

  Dans tous les cas, la documentation du compilateur utilisé s'avère indispensable.

# F Exceptions

C++ A INTRODUIT RÉCEMMENT un mécanisme de gestion d'erreurs par exceptions logicielles. Ce mécanisme n'est pas encore stabilisé, tous les compilateurs C++ n'en disposent pas, ou sur options, ou avec des variantes.

Ce manuel ne traite pas le sujet ! Si nécessaire, se référer au manuel de B. STROUSTRUP, [5], et étudier la documentation du compilateur utilisé.

# Index

# Bibliographie


**[1]** Brian W. KERNIGHAN & Dennis M. RITCHIE, *The C Programming Language*, Prentice-Hall, 1978

**[2]** Jean-François RABASSE, *Une introduction au langage C*, IPN-S2I, 1996

**[3]** Jean-François RABASSE, *Une introduction à la programmation objets et au langage C++*, IPN-S2I, 1998

**[4]** Bjarne STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, 1986

**[5]** Bjarne STROUSTRUP, *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1992