Langage C++

Essentiel

www.supinfo.com

Campus Booster ID : 302
Version 0.9

Sommaire

1. Introduction

1.1. Historique

Le C++ a été créé au cours des années 1980 par Bjarne Stroustrup alors qu'il travaillait au laboratoire Bell.

Son but était de modifier le C afin de permettre la programmation orientée objet. Il avait d'ailleurs appelé son langage *C with classes* (C avec des classes). Il voulait toutefois toucher un large public, donc son langage devait être entièrement compatible avec le C ANSI. Grâce à cette contrainte, pratiquement tout programme C peut être compilé en C++.

En plus de rendre le C orienté objet, Stroustrup a ajouté d'autres fonctionnalités telles que les fonctions virtuelles, la surcharge d'opérateurs, l'héritage simple et multiple, les « templates » et la gestion d'exceptions, entre autres.

1.2. Le langage

Le C++ est un langage *multi-paradigmes*, c'est-à-dire que le développeur peut choisir de quelle manière il veut programmer : de manière impérative, comme en C, ou orienté objets, une nouveauté du C++. Dans ce document nous aborderons principalement l'aspect orienté objet du langage.

Bien que le C++ soit l'évolution du langage C, il n'est pas nécessaire de connaître ce dernier pour écrire des programmes en C++ (c'est cependant préférable). Le sous-ensemble commun entre le C et le C++ est, d'ailleurs, généralement plus simple à apprendre en C++.

Certes, le fait de connaître un langage de programmation peut accélérer l'apprentissage de tout nouveau langage. En effet, quelqu'un qui programme en C peut apprendre le C++ pour devenir fonctionnel en une journée. Néanmoins, pour être tout à fait à l'aise avec les fonctionnalités avancées du langage, telles que l'abstraction des données, la programmation Orientée Objet, la programmation générique, etc. vous devrez travailler le C++ pendant au moins une ou deux années.

Nous allons, dans ce document, vous présenter les fonctionnalités de base ainsi que certaines fonctionnalités avancées du langage C++. Mais il va de soi que vous devrez fournir beaucoup d'effort, si vous souhaitez devenir « expert » dans la matière.

2. La compilation

2.1. Etapes

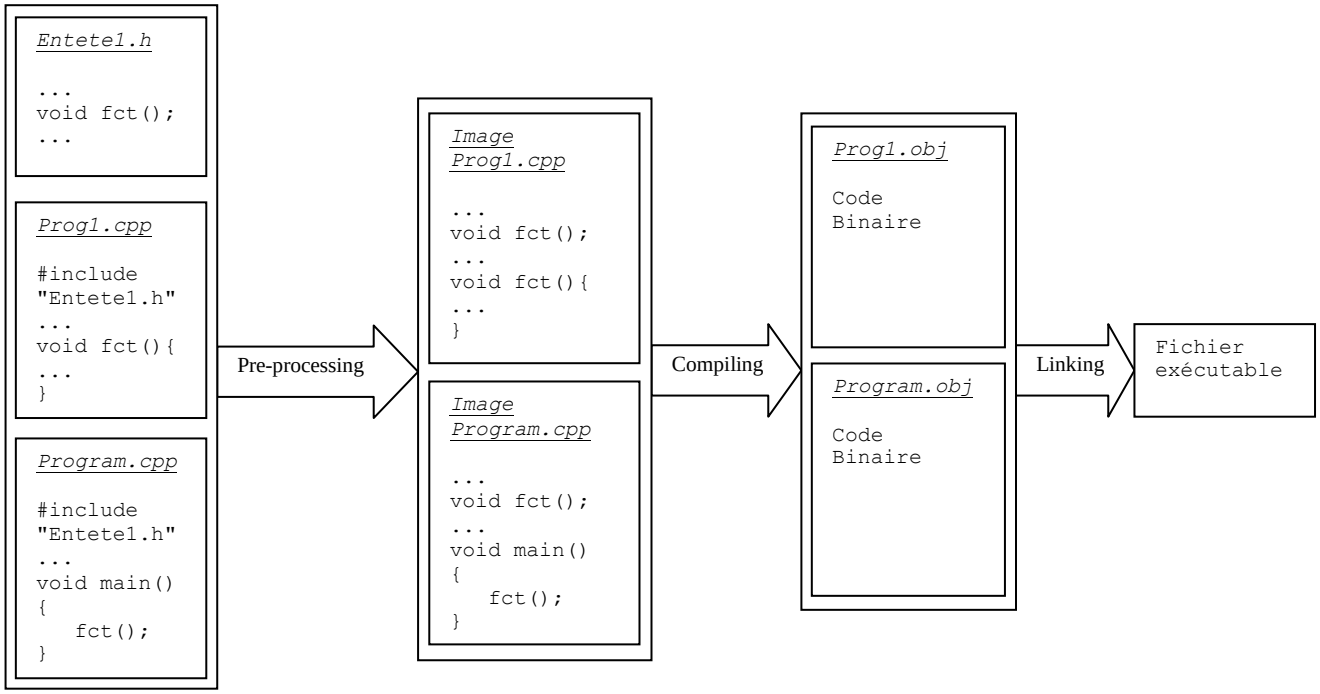
Le C++, comme le C, est un langage compilé. On ne dispose, au départ, que de fichiers sources.

Dans un premier temps, les fichiers doivent être traités avant la compilation. An appelle cette étape *pre-processing* en anglais et elle est effectuée, comme en C, par le préprocesseur. C'est dans cette étape que les directives du préprocesseur (#include, #define, …) sont interprétées.

Dans un deuxième temps, chaque fichier source généré par le préprocesseur est compilé indépendamment. Cette étape, réalisée par le compilateur, fabrique des fichiers objet qui contiennent la traduction en langage machine du fichier source correspondant.

Enfin, dans la dernière étape, appelée édition de liens (*linking* en anglais), on regroupe les différents fichiers objet en résolvant les références entre ceux-ci. Le fichier obtenu est exécutable par la machine.

Le schéma ci-dessous résume le procédé décrit :



2.2. Contraintes

Afin de compiler les fichiers intermédiaires créés par le préprocesseur, il est impératif que les entités suivantes soient être déclarées avant leur utilisation :

- les structures et classes
- les variables
- les fonctions

C'est l'éditeur de liens, dans la dernière étape de la compilation, qui se charge de faire correspondre les définitions de ces entités, aux endroits ou elles sont utilisées.

Pour déclarer les fonctions sans les définir, on utilise, comme en C, les prototypes. Les structures et les classes peuvent être déclarées de la manière suivante :

```
struct UnTypeDeStructure;
class CMaClasse;
```

Par défaut, une variable globale n'est accessible que dans le fichier source dans lequel elle est déclarée, car chaque fichier source va être compilé dans un fichier objet indépendant.

Cependant, il est possible de rendre une variable globale accessible dans tous les fichiers source d'un programme grâce au mot clé **extern**. Cette pratique, est toutefois à éviter.

```
extern UneVariableGlobale;
```

3. Généralités du langage

3.1. Organisation des fichiers sources

Il y a deux types de fichiers :

- les fichiers qui contiennent le code source, dont l'extension est **.cpp**
- les fichiers d'en-tête (*headers*), dont l'extension est **.h** ou **.hpp**

Seuls les fichiers contenant le code source (.cpp) sont compilés. Les fichiers d'en-tête servent à partager des déclarations (de variables, fonctions, classes, etc.) entre les fichiers source. Pour inclure les fichiers en-tête dans les fichiers source, on utilise la directive du préprocesseur **#include**.

Un fichier en-tête risque d'être inclus plusieurs fois dans le même fichier ce qui peut entraîner des collisions dans les déclarations. Pour éviter ceci très souvent les fichiers d'en-tête utilisent la compilation conditionnelle du préprocesseur. Par exemple :

```
Declaration.h (fichier d'en-tête)

#ifdef DECLARATION_H
#define DECLARATION_H
...
// déclarations
...
#endif DECLARATION_H
```

3.2. Les commentaires

Un bon développeur se doit de commenter son code afin de permettre aux personnes qui le liront plus tard (lui inclus) de le comprendre facilement.

On peut ajouter des commentaires au code source de deux manières :

- Commentaire sur une seule ligne :

```
int i;       // compteur
int j;
```

- Commentaire sur plusieurs lignes :

```
int i;       /* première ligne
int j;       la variable j ne sera pas prise en compte */
```

3.3. Les structures

On peut, comme en C, définir des types complexes grâce aux structures.

Toutefois, à la différence du C, en C++ les structures peuvent aussi contenir des fonctions.

```
struct employe
{
      char nom[10];              /*
      int heures;                   déclaration des données membre
      float salaire;             */
      void affiche();           // déclaration de la fonction membre affiche
};

void employe::affiche( )          // définition de la fonction affiche
```

```
{
	cout << "Nom : " << nom << "/n";
	cout << "Durée de service : " << heures << "/n";
	cout << "Salaire horaire : " << salaire << "/n";
}
```

:: est un opérateur qui montre le rattachement de la fonction « affiche » à la structure « employe ».

« affiche » peut afficher le nom, les heures et le salaire d'une variable de type « employe ».

Création de la variable « dupont » :

```
struct employe dupont ;
```

Appel des données membre de la structure par la variable « dupont »:

```
dupont.nom
dupont.heures
dupont.salaire
```

Nouveauté : en C++ la variable peut appeler aussi une fonction :

```
dupont.affiche( );
```

4. Spécificités du C++

4.1. Les références

4.1.1. Référence sur une variable

Une référence est un nom alternatif sur une variable. En conséquence, toutes les opérations réalisées sur une référence agissent aussi sur la variable référencée.

On utilise l'opérateur **&** pour créer une référence sur une variable :

```
int i = 1;
int &r = i; // r et i désignent le même emplacement de mémoire
i = 2;              // donc r = 2
r = 3;              // donc i = 3
i++;         // r = 4 et i = 4
r--;         // r = 3 et i = 3
```

Toutes les références doivent toujours être initialisées, sinon le compilateur générera une erreur, et une fois initialisée, une référence ne peut plus être changée.

```
int i = 1;
int x = 0;

int &r1 = i; // OK
int &r2;      // erreur de compilation : r2 non initialisé
r1 = x;      // r1 = 0 et i = 0
```

Les deux utilisations principales des références sont:

- le passage par référence des paramètres des fonctions
- les opérateurs surchargés

4.1.2. Passage par référence des paramètres des fonctions

En C il était possible de passer des paramètres aux fonctions de deux manières : par valeur ou par adresse. En C++ on introduit, en plus, la possibilité de passer des paramètres par référence.

Le passage de paramètres par référence apporte les avantages du passage par adresse : accès direct aux données et non pas à la copie, gain de performance ; sans les inconvénients : syntaxe simplifiée.

```
void swap(int &a, int *b)
{
	a ^= *b;
	*b ^= a;
	a ^= *b;
}

int main()
{
	int i = 4;
	int j = 5;

	swap(i, &j);
	// maintenant : i = 5, j = 4
}
```

Vous remarquez, dans l'exemple précédent, que le premier paramètre, passé par référence, est passé à la fonction et utilisé dans celle-ci sans avoir à préfixer les variables de caractères spéciaux. Tandis que le paramètre passé par adresse doit être préfixé par une esperluette (&) pour le transmettre à la fonction, et dans celle-ci il doit être préfixé d'un astérisque (*).

Le seul danger des paramètres passés par référence concerne les utilisateurs qui ne connaissent pas le prototype de la fonction et qui risquent de passer des paramètres à celle-ci sans savoir que ces paramètres seront peut-être modifiés après l'appel. Ceci n'arrive pas dans le passage par adresse, car l'utilisateur doit utiliser l'opérateur **&** pour passer les paramètres à la fonction.

Pour éviter ce problème, mais continuer à profiter du gain de performance obtenu en évitant la copie des paramètres, on peut utiliser le mot clé **const**. Le compilateur vérifiera alors que la variable référencée n'est jamais modifiée, ou générera une erreur dans le cas contraire.

```
void swap(const int &a, int *b)
{
      a ^= *b;     // ERREUR: a ne peut pas être modifié
      *b ^= a;
      a ^= *b;     // ERREUR: a ne peut pas être modifié
}
```

4.2. Arguments par défaut d'une fonction

Il est possible de donner une valeur par défaut aux arguments passés en paramètres d'une fonction.

Les valeurs par défaut des arguments sont assignées soit :

- dans le prototype de la fonction
- dans la déclaration de la fonction si un prototype n'est pas utilisé.

```
int fact(int = 0);

int puissance(int n, int m = 2)
{
    int res = n;
    for (int i = 0 ; i < m ; i++)
          res *= n;
    return res;
}

int main()
{
    cout << puissance(2);        // Au carré
    cout << puissance(2,3);      // Au Cube
    cout << fact();              // 0! = 1
    cout << fact(3);             // 3! = 6
    return 0;
}

int fact(int x)
{
    if(x == 0)
       return 1;
    else
       return x * fact(x - 1);
}
```

Il ne faut pas perdre de vue que ce sont des valeurs par **défaut**, et que vous pouvez toujours modifier cette valeur en l'indiquant lors de l'appel de la fonction comme d'habitude.

Les arguments ayant une valeur par défaut doivent être en fin de la liste des arguments dans le prototype de la fonction.

4.3. Fonctions surdéfinies

On parle de surdéfinition ou de surcharge lorsqu'un même symbole possède plusieurs significations, la sélection se faisant en fonction du contexte. Par exemple, l'opérateur « + » est surdéfini. On emploie le même opérateur, qu'il s'agisse d'une addition entre des entiers ou d'une addition entre des réels.

Dans le cas des fonctions, on peut en définir plusieurs avec le même nom tant qu'elles auront toutes soit :

- des paramètres de types différents
- un nombre différent de paramètres
- ou les deux

```
char fct(int a);
char fct(double a);             // type du paramètre différent du précédent
char fct(int a, int b);  // nombre de paramètres différent du premier
int  fct(int a, int b);  // ERREUR : seul le paramètre de retour change
```

C'est au compilateur de choisir quelle fonction appeler d'après le contexte d'appel.

Les types suivants sont considérés comme équivalents :

- un type et une référence sur un type
- une structure et ses alias
- un pointeur sur un type et un tableau d'éléments de ce même type

Lorsque le compilateur doit faire une conversion de type implicite pour appeler une fonction, il peut se retrouver avec cas ambigus :

```
void toto(int);
void toto(double);

char c; float y; long int z;

toto(c);     // Appelle toto(int) : promotion char => int
toto(y);     // Appelle toto(double) : promotion float => double
toto(z);     // ERREUR ! Ambigüité
```

Une autre situation qui peut être source d'ambigüité, est lorsque des valeurs par défaut sont utilisées dans des fonctions surdéfinies.

```
void string-copy(char *dest, const char *src);

void string-copy(char *dest, const char *src, int len = 10);

string-copy (destination, "cela");      // Ambigüité
```

Dans l'exemple précédent, le compilateur ne sait pas s'il doit appeler la première fonction, ou la deuxième avec la valeur par défaut.

4.4. Le type *bool*

En C, on n'avait pas de type spécifique pour les résultats de l'évaluation d'expressions booléennes. En effet, tout entier différent de 0 était évalué comme étant « vrai ». En C++ on a introduit le type **bool** pour définir les booléens.

Le type **bool**, en interne, est simplement une variable de type **char**. Le booléen est considéré faux si le caractère est égal à 0, et vrai pour les 255 autres valeurs.

Dans votre code, vous utiliserez les deux nouveaux mots clé : **true** et **false** pour affecter une valeur à une variable de type **bool**.

4.5. Déclaration des variables

En C la déclaration de toutes les variables doit être faite au début du bloc d'instructions. Ce n'est plus le cas en C++, il est possible d'écrire des instructions et de déclarer des variables « au milieu » du code.

```
int main() {
    cout << "Blabla";

    int val = 1 ; //déclaration d'une variable après une instruction

    cout << val;

    return 0;
}
```

4.6. Fonctions en ligne (*inline*)

Une fonction en ligne est identique à une fonction standard. Seules deux choses diffèrent :

- On ajoute le mot clé **inline** au début de la signature
- Les fonctions ne sont plus un bloc d'instructions exécuté chaque fois que la fonction est appelée. Leur code est « recopié » au moment de la compilation, partout ou la fonction est appelée. (Les appels de fonctions sont remplacés).

```
inline void affiche();

int main()
{
    affiche();
    return 0;
}

inline void affiche()
{
    cout << "Hello";
}
```

Attention : les fonctions **inline** sont en général de petites fonctions, sinon le gain de performance risquerait de ne plus en être un.

4.7. Les valeurs constantes nommées

Les constantes sont déclarées avec le mot clé **const** ou bien avec la directive **#define**.

Une variable doit être initialisée lors de sa déclaration.

```
#define PI = 3.14
const string HELLO = "Salut";
const int i = 7;
```

Leurs valeurs ne peuvent changer au cours de l'exécution du programme.

```
i++;   // erreur à la compilation
```

U

```
int main()
{
    cout << HELLO;
}
```

Une constante peut être déclarée dans tout bloc de code, mais sa portée est limitée à ce bloc et ses sous blocs. Si la constante est déclarée en dehors d'un bloc, sa portée est limitée au fichier source dans lequel elle est définie.

En utilisant le mot clé **const** on peut spécifier le type de la constante. Ce qui n'est pas le cas avec **#define**.

4.8. Type énumération

Le type **enum** désigne une liste de constantes entières. Il est souvent utilisé pour définir une liste d'éléments n'ayant pas forcément de valeur significative (des couleurs, des jours dans la semaine, etc.)

On peut l'utiliser de deux façons différentes :

- enum **nom** {liste_elements};
- typedef enum {liste_elements} **nom**;

```
enum couleur {rouge, vert, bleu, gris, tomate};

int main()
{
    couleur maCouleur;

    maCouleur = tomate;    // maCouleur = 4

    return 0 ;
}
```

Dans cet exemple, les valeurs des constantes ont été automatiquement attribuées par le compilateur en commençant à 0 et en incrémentant de 1 pour chaque constante :

rouge = 0, vert = 1, bleu = 2, etc.

On peut aussi spécifier les valeurs souhaitées :

```
enum couleur { rouge = 10, vert, bleu = 14, gris, tomate};
```

Dans cet exemple les valeurs sont :

rouge = 10, vert = 11, bleu = 14, gris = 15, tomate = 16.

Vous remarquerez que le compilateur à automatiquement assigné des valeurs successives aux constantes qui n'en avaient pas.

On peut convertir une valeur d'énumération en un entier, mais pas l'inverse (On ne peut pas récupérer la valeur « rouge » à partir de l'entier 0).

Les énumérations offrent, entre autres, les avantages suivants :

- Permettent d'attribuer un nom à un ensemble de noms (compact et clair)
- Permettent la création de variables du type de l'énumération
- Les constantes numériques de l'énumération sont affectées aux variables

4.9. Le mot clé *mutable*

Le mot clé **mutable** permet de rendre un champ d'une structure constante accessible en écriture. Voici un exemple :

```
struct T
{
	long a;
	mutable long b;
};
...
const T MaStructure = { 5,6 };
MaStructure.a = 7;  // erreur de compilation
MaStructure.b = 8;  // OK
```

5. Les entrées/sorties

En C, les entrées sorties sont gérées principalement via les fonctions ***printf*** et ***scanf***.

Le C++ facilite ces opérations grâce à des flux d'octets, appelés également **flots**.

Il existe un flot standard d'entrée (saisie d'informations par l'utilisateur) et un flot standard de sortie (écriture d'informations à l'écran).

Les objets nécessaires pour accéder à ces flots sont définis dans la librairie **iostream** (**I**nput **O**utput **S**tream = flot d'entrée sortie) qui fait partie de la STL (voir chapitre 11). Il est donc nécessaire de l'inclure en utilisant la directive **#include**.

5.1. Envoyer des informations vers le flot standard de sortie

Pour écrire des informations à l'écran, on redirige (insère) des données dans le flot via l'objet **cout** et l'opérateur <<.

Il faut préalablement importer cet objet qui se trouve dans l'espace de noms **std**. Pour plus d'informations sur les espaces de noms, se référer au chapitre 8.

```
#include <iostream>
using std::cout;

int main()
{
    cout << "Bonjour";
    return 0;
}
```

L'exécution de ce programme affiche « Bonjour » à l'écran.

Vous remarquerez que la syntaxe est plus simple que celle de la fonction ***printf***. Cette syntaxe est d'autant plus simple qu'elle va permettre d'insérer des valeurs de variable dans le flot comme ci-dessous :

```
#include <iostream>
using std::cout;

int main()
{
    char* nom = "bibifoc";
    cout << "Bonjour " << nom << " !";
    return 0;
}
```

Ce programme affiche « Bonjour bibifoc ! ».

L'opérateur << est appelé « opérateur d'indirection ». Il est dit comme étant associatif à gauche, ce qui signifie que les éléments sont envoyés dans le flot en commençant par celui le plus à gauche.

Dans l'exemple précédant, la chaîne « Bonjour » est envoyé dans le flot **cout** puis la valeur de la variable nom, puis la chaîne « ! ».

L'opérateur << accepte tous les types prédéfinis en C++, c'est-à-dire **short**, **int**, **long**, **float**, **double**, **char**, et **char***. Il effectue lui-même un formatage implicite en fonction du type de donnée (Avec **printf** il fallait préciser le formatage via **%d**, **%f**, etc.)

5.2. Récupérer des informations à partir du flot standard d'entrée

Pour récupérer des informations saisies par l'utilisateur, on les récupère à partir du flot via l'objet **cin** et l'opérateur d'indirection >>.

Il faut préalablement importer cet objet qui se trouve dans l'espace de noms **std**. Pour plus d'informations sur les espaces de noms, se référer au chapitre 8.

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    char c;
    cin >> c;
    cout << c;
    return 0;
}
```

L'exécution de ce programme attend la saisie d'un caractère au clavier, l'enregistre dans la variable **c** et l'affiche à l'écran.

Le type de la variable définit le type de donnée attendu.

cin est beaucoup plus simple d'utilisation que ***scanf*** grâce au formatage implicite de l'opérateur >> (identique au formatage implicite de <<).

6. Programmation Orientée Objet

6.1. Remarques sur la programmation objet

La programmation classique est basée sur la dichotomie « données – procédures ».

La programmation objet est construite autour des notions d'objet et de classe.

Une classe est un type de données (tel que **int** ou une structure en C). Un objet est une instance d'une classe. Tout programme peut être vu comme un ensemble d'objets qui interagissent entre eux.

La programmation objet implique donc une « structure modulaire ».

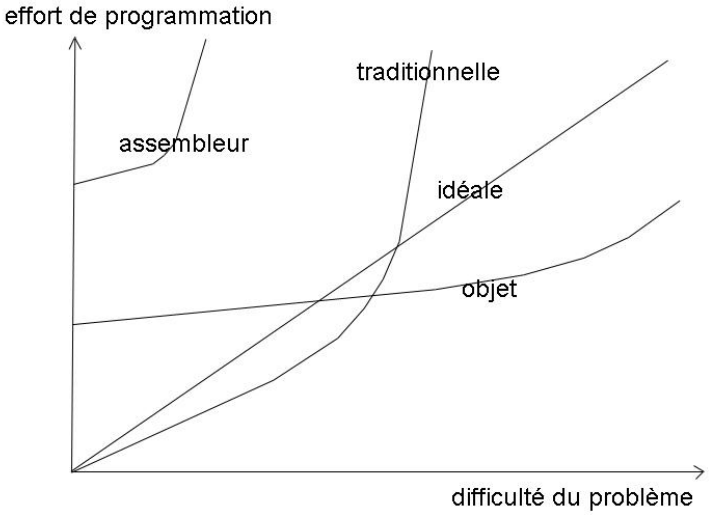
Chaque module est une entité indépendante, cohérente et manipulable.

Un programme est une collection d'objets actifs et autonomes.

Qualités de la programmation objet :

- Productivité accrue car :
  - équipes de programmation indépendantes
  - héritage du code
- Maintenance et mise à jour faciles
- Robustesse (procédures et données encapsulées donc protégées)
- Extensibilité (ajout de code) facile

6.1.1. Comparaison entre la programmation objet et la programmation traditionnelle



Conclusions :

- La programmation traditionnelle est plus simple à apprendre et à utiliser pour les problèmes à difficulté réduite.
- La programmation en assembleur est difficile à apprendre et à utiliser même dans les cas simples.
- La programmation objet est préférable dans le cas des problèmes complexes.

6.2. Les classes

6.2.1. Notions et vocabulaire

Une classe est un type de données, au même titre que **int** ou une structure. On dit même parfois, par abus de langage, qu'une instance d'une classe est une variable du type de la classe.

Une classe est essentiellement constituée de :

- attributs
- méthodes

On les appelle les membres de la classe.

Les attributs permettent à chaque objet de renfermer une quantité d'information sous forme de variables. Ces variables peuvent être d'un type natif (**int**, **float**, etc.) ou, à leur tour, des objets.

La notion de méthode se rapproche de celle de fonction, à ceci près qu'une méthode est appelée sur un objet de la classe. Une méthode agit sur l'objet sur lequel elle est appelée. Elle peut aussi bien lire ou écrire les attributs, qu'effectuer une action sur l'objet. Les méthodes, tout comme les fonctions, peuvent être surdéfinies.

Tout comme les variables, les objets sont construits et détruits. Une méthode est automatiquement appelée lors de la construction. Elle est appelée constructeur. Elle permet par exemple d'initialiser l'objet, ou d'allouer des ressources. Comme toute autre méthode, le constructeur peut être surdéfini afin de construire un objet de différentes façons.

De même, une autre méthode est automatiquement appelée lors de la destruction d'un objet. C'est le destructeur. Il sert principalement à libérer les ressources utilisées par l'objet, mais contrairement au constructeur, le destructeur ne peut pas être surdéfini.

6.2.2. Déclaration de classes et l'opérateur de résolution de portée

La déclaration d'une classe ressemble beaucoup à la déclaration d'une structure :

```
class nom_de_la_classe
{
      type attribut1;
      ...
      type attributN;

      methode1(...);
      ...
      methodeM(...);
}; // Ne pas oublier le ;
```

Elle contient, à l'intérieur, la déclaration de tous les attributs et de toutes les méthodes de la classe.

Les attributs doivent être déclarés, mais en aucun cas peuvent-ils être initialisés dans la définition de la classe.

Contrairement aux attributs, les définitions des méthodes de la classe peuvent être écrites à l'intérieur ou à l'extérieur de la définition de la classe :

```
class Point
{
      int x;
      int y;

      Point() { x = y = 0; }  // Méthode déclarée et définie à l'intérieur
      void afficher();        // Méthode déclarée à l'intérieur, mais pas définie
};

void Point::afficher()
```

```
{
	cout << "(" << x << "; " << y << ")";
}
```

L'opérateur **::** est appelé « opérateur de résolution de portée ». Il permet d'accéder à une variable ou à une fonction dans un contexte déterminé. Si rien n'est indiqué devant l'opérateur de résolution de portée, on considère que l'on veut accéder à des variables ou des fonctions de portée globale.

```
#include <iostream.h>

int i = 10;		// Variable, portée globale (fichier dans où elle est définie)
void fct();		// Fonction, portée globale (idem)

class MaClasse
{
public:
	static int i;		// Variable de la classe MaClasse
	void fct();		// Méthode de la classe MaClasse
};

int MaClasse::i = 4;

void fct()		// Définition de la fonction globale
{
	...
}

void MaClasse::fct()		// Définition de la méthode de la classe en spécifiant le
				// nom de la classe devant l'opérateur ::
{
	...
}


int main()
{
	int i = 22;		// Variable locale, portée: méthode main

	cout << "Variable locale : " << i << endl;	// Appel à la variable locale
	cout << "Variable globale : " << ::i << endl; // Appel à la variable globale
							// grâce à l'opérateur ::
	// Appel à la variable de classe grâce en spécifiant le nom de la classe
	// devant l'opérateur ::
	cout << "Variable de classe : " << MaClasse::i << endl;
}
```

L'exemple ci-dessus montre comment accéder à trois variables de même nom, mais avec des portées différentes. Il fait de même pour la définition de deux fonctions ayant le même nom. Certaines fonctionnalités plus avancées utilisées dans cet exemple seront abordées avec plus de détails dans la suite du chapitre.

6.2.3. L'encapsulation

L'encapsulation est l'un des principes de base de la programmation objet. C'est un puissant concept qui permet de maîtriser la visibilité qu'on a des membres d'une classe vus de l'extérieur de la classe.

L'utilisateur d'une classe, c'est-à-dire un développeur qui utilise une classe préalablement définie, ne pourra donc voir et modifier, ou utiliser, qu'un nombre restreint d'attributs et de méthodes. Cela réduit d'autant la complexité d'utilisation des objets, et empêche l'accès accidentel aux données de ceux-ci.

Il existe trois niveaux de visibilité :

- **public** : un attribut qui a une visibilité publique peut être lu et écrit à partir d'une instance d'une classe. De même une méthode publique peut être appelée partout dans le code, à partir d'une instance de la classe.
- **private** : un attribut privé ne peut être lu et écrit qu'à partir d'une méthode de la classe. De même, une méthode privée ne peut être appelée qu'à partir d'une autre méthode de la classe.
- **protected** : Le niveau de visibilité protégé est utilisé pour restreindre l'accès aux attributs et aux méthodes d'une classe par les méthodes des classes dérivées. Cette notion est abordée dans la section sur l'héritage.

Voici un exemple pour exposer la syntaxe avec les mots clés **public**, **protected** et **private** :

```
class UneClasse
{
	int UnAttributPriveParDefaut;
	void UneMethodePriveeParDefaut();
public:
	long UnAttributPublique;
	long UneMethodePublique(long a);
protected:
	char UnAttributProtege;
	short UneMethodeProtegee(char * p);
private:
	char* UnAttributPrive;
	short UneMethodePrivee(double d);
};
```

Notez que par défaut les membres d'une classe déclarée avec le mot clé **class** sont privés. En effet, on peut déclarer une classe avec le mot clé **struct**. Dans ce cas les membres sont par défaut publiques. Ceci montre qu'en C++ la notion de structure et de classe est identique.

6.2.4. Accès aux membres d'une classe

Nous venons de voir qu'en C++ les concepts de classe et de structure sont pratiquement identiques. Etant donné que les structures existaient déjà en C et que lors de la création du C++ la compatibilité avec son prédécesseur était vitale, on peut supposer que les opérateurs utilisés pour accéder aux membres d'une structure ou d'une classe n'ont pas été modifiés.

En effet, comme en C, on utilise les opérateurs '**.**'ou '**->**' pour accéder aux membres d'une classe, selon que l'on a une instance ou un pointeur sur une instance.

Ces opérateurs ne sont, cependant, pas nécessaires si, dans une méthode de la classe, on souhaite accéder aux attributs de celle-ci.

```
class Point
{
	int x;
	int y;

public:
	Point() { x = y = 0; }  // On accède aux attributs sans utiliser les
				// opérateurs . ou -> car il se trouvent dans la
				// même classe
	void afficher();
};

void Point::afficher()
{
	cout << "(" << x << "; " << y << ")";
}

int main()
{
	Point p;		// On crée un point p
	Point *pp = &p;		// On crée un pointeur sur le point p

	p.afficher();			// Utilisation de . pour appeler une méthode
à				// partir d'une instance d'une classe
	pp->afficher();		// Utilisation de -> pour appeler une méthode à
				// partir d'une pointeur sur une instance
	(*pp).afficher();	// Utilisation de . car (*pp) est equivalent à
				// l'instance de la classe et non pas le pointeur
	int i = p.x;		// ERREUR, car x est un membre privé de p
}
```

6.2.5. Les constructeurs

Il a été dit plus haut qu'il existe une méthode qui est appelée automatiquement lorsque l'on veut créer une instance d'une classe. On l'appelle le constructeur.

Le constructeur est une méthode comme toute autre à deux contraintes près :

- le constructeur doit avoir exactement le même nom que la classe
- le constructeur ne doit pas retourner de valeur, même pas le type **void**.

Une classe peut avoir zéro ou plusieurs constructeurs :

- Pas de constructeur : dans ce cas le compilateur fournit automatiquement un constructeur par défaut, qui n'accepte pas de paramètres.

- Un seul constructeur sans paramètres : dans ce cas ce sera toujours lui qui sera appelé. Un constructeur qui propose des valeurs par défaut pour tous ces paramètres peut être considéré, par le compilateur, comme un constructeur sans paramètres.
- Plusieurs constructeurs : le constructeur, comme toute autre méthode, peut être surchargé. Dans ce cas, le compilateur déterminera le constructeur à appeler en fonction des paramètres qui lui sont passés.

Attention : Lorsque l'on déclare un ou plusieurs constructeurs avec des paramètres, le constructeur par défaut n'est plus fourni par le compilateur.

Pour créer une instance d'une classe, c'est-à-dire un objet, il suffit de déclarer une variable dont le type sera le nom de la classe.

```
MaClasse uneVariable;
```

L'exemple ci-dessus crée un objet en appelant soit le constructeur par défaut fourni par le compilateur, soit un constructeur sans paramètres créé par le développeur.

Pour créer un objet en faisant appel à un constructeur qui requiert des paramètres, il suffit de passer la liste de paramètres entre parenthèses après le nom de la variable.

```
MaClasse uneVariable(param1, param2);
```

Voici un exemple d'une classe avec 2 constructeurs :

```
class Point
{
	int x, y;

public:
	Point(int a) { x = y = a; }
	Point(int a, int b)
	{
		x = a;
		y = b;
	}

	void afficher();   // Définie dans un exemple précédent
};

int main()
{
	Point p1;          // ERREUR : seulement des constructeurs avec un ou
	                   // plusieurs paramètres ont été définis
	Point p2(1);
	p2.afficher();     // (1; 1)

	Point p3(1, 4);
	p3.afficher();     // (1; 4)
	return 0;
}
```

Les rôles principaux du constructeur sont :

- l'initialisation des attributs
- l'allocation des ressources (zone mémoire, connexions, attributs dynamiquement alloués…).

6.2.6. Les listes d'initialisation

Bien qu'ils ne soient pas limités à cela, les constructeurs sont utilisés la plupart du temps pour initialiser les attributs d'un objet. Dans l'exemple précédent, nous avons utilisé la méthode « traditionnelle » pour initialiser les attributs de la classe Point, mais il existe une autre méthode, qui vous aidera à réduire le nombre de lignes de code et à rendre votre code plus lisible : les listes d'initialisation.

Voici trois nouveaux constructeurs pour la classe Point :

```
Point() : x(0), y(0) {}
Point(int a) : x(a), y(a) {}
Point(int a, int b) : x(a), y(b) {}
```

Il est aussi possible de combiner ces trois constructeurs en un seul :

```
Point(int a = 0, int b = 0) : x(a), y(b) {}
```

Etant donné que les constructeurs peuvent aussi être utilisés pour effectuer des opérations lors de la création d'un objet, il est évidemment possible d'ajouter des instructions dans les accolades qui suivent la liste d'initialisation.

6.2.7. Le destructeur

Le destructeur, comme son nom l'indique, sert à détruire un objet, c'est-à-dire, à libérer l'espace mémoire qu'il occupe. Contrairement aux constructeurs, il ne peut y avoir qu'un seul destructeur par classe.

On peut créer son propre destructeur, dans ce cas il doit respecter les règles suivantes :

- le destructeur doit avoir exactement le même nom que la classe précédé du caractère tilde (~)
- le destructeur ne doit pas retourner de valeur, même pas le type **void**.

Si aucun destructeur n'a été défini par le développeur, le compilateur en crée un automatiquement. Ceci est nécessaire car le destructeur est toujours appelé automatiquement lorsqu'un objet n'est plus nécessaire.

Souvent, il n'est pas nécessaire de définir ses propres destructeurs. Cependant, il est impératif de définir des destructeurs personnalisés, si l'objet a fait des allocations dynamiques de mémoire.

Voici un exemple avec un attribut alloué dynamiquement:

```
class CFoo
{
private:
	long * m_pAttribut;
public:
	 CFoo(long a);
	 ~CFoo();
};

CFoo::CFoo(long a) {
	m_pAttribut  = (long*) malloc(sizeof(long));
	*m_pAttribut = a;
}

CFoo::~CFoo() {
	free( m_pAttribut );
}
```

6.2.8. Constructeur par copie

Il existe un constructeur spécial appelé constructeur par copie.

Le constructeur par copie recopie bit par bit les champs d'un objet donné dans un autre de la même classe. Si la définition de la classe ne contient pas explicitement un tel constructeur, il est créé par défaut.

Comme son nom l'indique il est appelé lorsque le compilateur a besoin de créer une copie d'un objet. Ceci peut arriver dans l'un des trois cas suivants :

- un objet est utilisé pour initialiser un autre objet de la même classe
- un objet est passé par valeur à une fonction
- un objet est retourné par valeur par une fonction

Pour créer un constructeur par copie, il suffit de créer un constructeur qui prend, comme seul paramètre, une référence constante à un objet du type de la classe dans laquelle le constructeur est défini.

```
MaClasse(const MaClasse& mc) { ... }
```

Si on spécifie un paramètre constant, c'est simplement pour éviter que la valeur originale soit accidentellement modifiée lors de la copie. Par contre, il est impératif de passer une référence de l'objet à copier et non pas une copie, car dans le cas contraire on se trouverait dans une récursivité infinie.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}

	Point(const Point&);	// Déclaration du constructeur par copie

	void afficher();
};

Point::Point(const Point& p) : x(p.x), y(p.y) {} /* Définition du constructeur
par                                                copie */
int main()
{
	Point p1;		// Appel au constructeur par défaut
	p1.afficher();		// (0; 0)

	Point p2(p1);			// Appel au constructeur par copie
	p2.afficher();		// (0; 0)

	return 0;
}
```

Notez qu'il est possible d'utiliser les listes d'initialisation dans le constructeur de copie, en faisant appel aux attributs du paramètre.

Dans l'exemple ci-dessus, le compilateur aurait pu créer le constructeur par copie automatiquement. Toutefois, il est nécessaire de déclarer un constructeur de copie si la classe contient des attributs alloués dynamiquement.

Finalement, notez aussi que ce n'est pas le constructeur de copie qui est appelé lors de l'affectation d'un objet (a = b;). Il s'agit la de l'utilisation de l'opérateur d'affectation, qui peut à son tour être surdéfini. Nous aborderons la surdéfinition d'opérateurs, dans un chapitre ultérieur.

6.2.9. Constructeur de transtypage (*cast*)

Les constructeurs sont aussi utilisés dans les conversions de type dans lesquelles le type cible est la classe du constructeur.

Par défaut, dans une classe utilisateur, aucun constructeur de transtypage n'est défini. C'est à l'utilisateur de le définir, avec pour argument du constructeur un objet du type source.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int*); // Conversion d'un tableau de int en Point

	void afficher();
};

Point::Point(int* i) : x(i[0]), y(i[1]) {} // Définition du constructeur de cast

int main()
{
	Point p1;		// Appel au constructeur par défaut
	p1.afficher();		// (0; 0)

	int i[2] = {2, 3};
	p1 = i;			// Appel au constructeur de cast
	p1.afficher();		// (2; 3)

	return 0;
}
```

Dans l'exemple ci-dessus, le transtypage se fait de manière transparente pour l'utilisateur. Toutefois, il y a des cas dans lesquels il est important que l'utilisateur soit au courant de l'opération de transtypage et donc on doit l'obliger à l'indiquer explicitement. Typiquement, il est important qu'un développeur valide les opérations de transtypage où il risque d'y avoir une perte de précision.

On obtient ce comportement en précédant le prototype du constructeur du mot clé **explicit**.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);	// Conversion d'un tableau de float en Point

	void afficher();
};
```

```
Point::Point(float* f)     // Définition du constructeur de cast
{
       x = (int)f[0];
       y = (int)f[1];
}

int main()
{
       Point p1;           // Appel au constructeur par défaut
       p1.afficher();      // (0; 0)

       float f[2] = {4.2, 6.8};
       p1 = f;             // ERREUR: le cast doit être spécifié explicitement
       p1 = (Point)f;      // Appel au constructeur de cast
       p1.afficher();      // (4; 6)

       return 0;
}
```

6.2.10. Membres statiques

En règle générale, un attribut ou une méthode ne peuvent être utilisés qu'à partir d'instances d'une classe. Ces attributs et ces méthodes sont alors complètement indépendants et peuvent être différents d'un objet à l'autre.

Il existe toutefois, des situations dans lesquelles on souhaite que tous les objets d'une classe partagent les mêmes informations, ou on souhaite faire appel à des méthodes sans avoir à instancier une classe. On peut grâce au mot clé **static**, créer des attributs et des méthodes statiques qui répondent à ces besoins.

Aucun attribut ne peut être initialisé lors de sa déclaration, mais toute variable doit être initialisée avant d'être utilisée. Normalement, un attribut est initialisé dans le constructeur de la classe, mais un attribut statique peut être utilisé avant qu'une instance de la classe soit créée. Il est même possible que la classe ne soit jamais instanciée.

Pour initialiser un attribut statique, on utilise l'opérateur de résolution de portée. On assigne la valeur à l'attribut après la définition de la classe. On utilise le même opérateur pour accéder à l'attribut.

```
class Compteur
{
public:
       static int c;
};
int Compteur::c = 0;

int main()
{
       cout << Compteur::c;       // 0
}
```

Les méthodes statiques ne sont pas applicables sur un objet. Dans leur contexte, le pointeur **this** n'existe pas (voir 6.2.12). Elles ne peuvent accéder qu'aux attributs statiques de la classe. Il y a deux manières d'appeler une méthode statique : directement à partir du nom de la classe, ou à partir d'une instance de la classe.

Voici un exemple qui utilise des attributs et des méthodes classiques et qui met en évidence les erreurs qu'il est possible de commettre.

```
class Roue
{
public:
	int taille;
	static bool roule;

	Roue(int t = 24) : taille(t) {}

	static void rouler(bool b) { roule = b; }
	static void changerTaille(int i) { taille = i; }     // ERREUR: car taille
                                                         // n'est pas statique
	void changerTaille(int i) { taille = i; }
};

bool Roue::roule = false;

int main()
{
	Roue r1, r2(18);

	cout << "r1 mesure " << r1.taille << " pouces" << endl
	     << "r2 mesure " << r2.taille << " pouces" << endl;

	cout << "r1 roule? " << r1.roule << " | r2 roule? " << r2.roule << endl;

	r2.rouler(true);

	cout << "r1 roule? " << r1.roule << " | r2 roule? " << r2.roule << endl;

	Roue::roule = false;

	cout << "r1 roule? " << r1.roule << " | r2 roule? " << r2.roule << endl;

	Roue::rouler(true);

	cout << "r1 roule? " << r1.roule << " | r2 roule? " << r2.roule << endl;

	Roue::taille = 24;  // ERREUR: car taille n'est pas statique, donc il ne peut
	                    // être modifié que par une instance de la classe
	Roue::changerTaille(24);   // ERREUR: car changerTaille n'est pas statique,
	                           // donc il ne peut être modifié que par une
	                           // instance de la classe

	r2.changerTaille(24);

	cout << "r1 mesure " << r1.taille << " pouces" << endl
	     << "r2 mesure " << r2.taille << " pouces" << endl;

}
```

Ce programme affiche :

```
r1 mesure 24 pouces
r2 mesure 18 pouces
r1 roule? 0 | r2 roule? 0
r1 roule? 1 | r2 roule? 1
r1 roule? 0 | r2 roule? 0
r1 roule? 1 | r2 roule? 1
r1 mesure 24 pouces
r2 mesure 24 pouces
```

6.2.11. Les méthodes constantes

C++ offre la possibilité de définir des méthodes constantes, c'est à dire que le compilateur garantit que leurs appels ne modifient aucun attribut de l'objet.

Il suffit de rajouter le mot clé **const** à la fin de la méthode.

Le compilateur n'autorise que l'appel de méthodes constantes sur un objet constant.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);

	void afficher() const
	{
		cout << "(" << x << "; " << y << ")";
	}

	void incrementer() const
	{
		x++; y++;	// ERREUR: car la methode ne doit pas modifier l'objet
	}
};

int main()
{
	Point p;
	p.afficher();
}
```

6.2.12. Le pointeur *this*

Dans toutes les méthodes (non statiques, voir 6.2.10) de toutes les classes, C++ définit automatiquement un pointeur nommé **this**. C'est un pointeur, du type de la classe, qui pointe vers l'objet courant.

On s'en sert surtout pour communiquer un accès à l'objet courant, à une entité extérieure à la classe.

```
class Point; // Déclaration de la classe Point nécessaire car la fonction fct est
             // déclarée avant la définition de Point

void fct(Point*);   // Fonction globale définie ailleurs. Reçoit un pointeur car
                    // this est un pointeur

class Point         // Définition de la classe Point
{
      int x, y;

public:
      Point(int a = 0, int b = 0) : x(a), y(b) {}
      Point(const Point& p) : x(p.x), y(p.y) {}
      Point(int* i) : x(i[0]), y(i[1]) {}
      explicit Point(float*);

      void afficher() const
      {
            cout << "(" << x << "; " << y << ")";
      }

      void incrementerXDe(int x)
      {
            this->x += x;       // Pointeur this utilisé pour distinguer entre
                                // l'attribut et le paramètre.
                                // PRATIQUE NON RECOMMANDÉE!
      }

      void fonctionMysterieuse()
      {
            fct(this);          // La fonction globale fct agira sur l'objet
                                // courant
      }
};
```

6.3. Opérateurs *new* et *delete*

Le langage C++ introduit deux nouveaux opérateurs de gestion dynamique de la mémoire : **new** et **delete**. Le premier effectue l'allocation dynamique d'un objet dans la mémoire, le second réalise la libération.

Ces opérateurs ont été créés pour la création et la destruction des objets, mais on peut les utiliser avec tout autre type. Voici un exemple :

```
class Dynamique
{
	long* pl;

public:
	Dynamique(long l)
	{
		pl = new long(l);
	}

	~Dynamique()
	{
		delete pl;
		pl = 0; // TOUJOURS mettre à 0 les pointeurs qui ne sont plus valides
	}
};

int main()
{
	Dynamique* d = new Dynamique(23);

	delete d;
	d = 0;		// TOUJOURS mettre à 0 les pointeurs qui ne sont plus valides
}
```

Le constructeur est appelé automatiquement après l'allocation de l'objet.

Le destructeur est appelé automatiquement avant la libération de la mémoire.

Tout objet alloué avec l'opérateur **new** doit être supprimé avec l'opérateur **delete** sinon le programme présentera des fuites de mémoire.

6.4. Les tableaux d'objets

6.4.1. Introduction

Comme pour tout autre type de données, il est possible en C++ de créer des tableaux d'objets de même type.

Il faut savoir qu'un constructeur est appelé pour chaque élément du tableau lors de sa création, en commençant par l'élément 0 et allant jusqu'à l'élément $n - 1$, où n est le nombre d'élément dans le tableau. De même, lorsqu'un tableau est détruit, le destructeur est appelé pour chacun des éléments, mais dans l'ordre inverse.

Pour créer un tableau non-initialisé, la classe doit absolument avoir un constructeur sans paramètres, ou le constructeur par défaut. Toutefois il est possible d'initialiser un tableau lors de sa création. Il faut alors prévoir des constructeurs qui utilisent le type de données transmis.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);

	void afficher() const
	{
		cout << "(" << x << "; " << y << ")";
	}
};

int main()
{
	Point tabPoints[2];		// Le constructeur par défaut est appelé pour
					// chacun des éléments du tableau

	tabPoints[0].afficher();	// (0; 0)

	Point tabPoints2[3] = { 3, 4 };	// Le constructeur qui prend un int est
					// appelé pour les 2 premiers éléments et le
					// constructer par défaut est appelé pour le
					// dernier

	tabPoints2[1].afficher();	// (4; 0)
	tabPoints2[2].afficher();	// (0; 0)

	Point tabPoints3[2] = { 3, Point(5, 6)};	// Le constructeur qui prend 1
					// int est appelé pour le premier
					// élément et le constructeur par copie est
					// appelé pour le 2ème

	tabPoints3[0].afficher();	// (3; 0)
	tabPoints3[1].afficher();	// (5; 6)
}
```

6.4.2. Création dynamique des tableaux d'objets

Contrairement au C, on peut créer des tableaux dont la taille est spécifiée dynamiquement.

Ceci est possible grâce à l'opérateur **new[]**. De même la mémoire occupée par les tableaux dynamiques est libérée avec l'opérateur **delete[]**. Ces opérateurs appellent les constructeurs et destructeurs de la même façon que lors de la création des tableaux d'objets non dynamiques.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);

	void afficher() const
	{
		cout << "(" << x << "; " << y << ")";
	}
};

int main()
{
	int taille = 2;
	Point* tabPoints = new Point[taille];

	tabPoints[0].afficher();  // (0; 0)

	delete[] tabPoints;
	tabPoints = 0;
}
```

6.5. Héritage

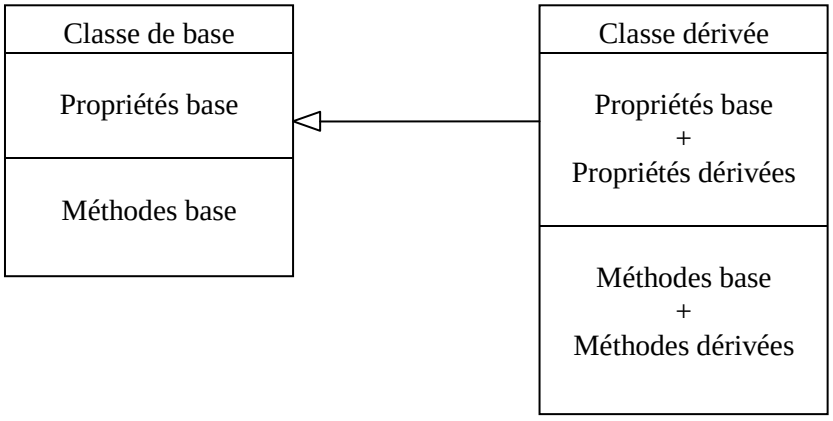
6.5.1. Introduction

Le mécanisme de dérivation ou d'héritage, est un des points clés de la programmation objet. Il est la réponse au problème de la réutilisation de code.

L'héritage consiste à transmettre les attributs et les méthodes d'une classe à une sous-classe. On appelle la première la classe mère ou classe de base, et la seconde classe fille ou classe dérivée.

La classe fille possédera tous les attributs et les méthodes de sa classe mère et peut l'étendre avec ses propres attributs et méthodes. Ayant une classe de base, on peut dire qu'on ne programme que la différence pour obtenir la classe dérivée.

On peut représenter l'héritage par un graphe :



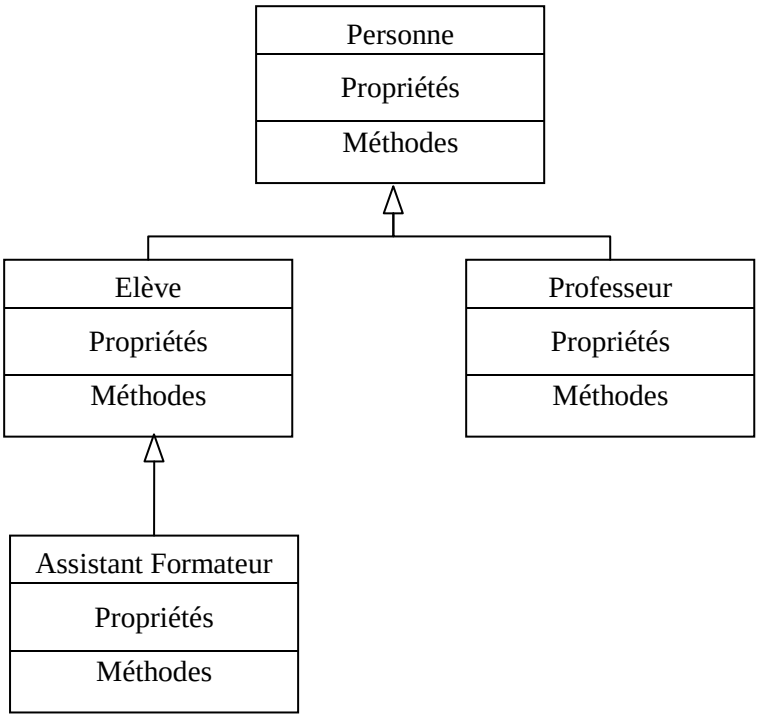
Lorsque l'on appelle une méthode sur la classe dérivée le programme vérifie si elle est définie dans celle-ci. Si ce n'est pas le cas, le message remonte dans la hiérarchie jusqu'à ce qu'une définition de la méthode est trouvée.

On verra qu'une classe fille peut non seulement étendre la classe mère, mais elle peut aussi redéfinir certaines méthodes de celle-ci.

6.5.2. Héritage simple

Dans le cas de l'héritage simple, une classe ne peut dériver que d'une seule autre classe. Cependant, plusieurs classes peuvent dériver d'une seule classe de base.

Ceci est représenté dans un schéma de dérivation par un arbre dans lequel chaque nœud n'a qu'un seul père direct, et il n'existe donc qu'un seul chemin unique de chaque nœud vers la racine.




```
class Personne
{
protected:
	char* nom;
};

class Eleve : public Personne		// Eleve hérite de Personne
{
protected:
	int moyenne;
};

class Professeur : Personne		// Professeur hérite de Personne
{
	char* matiere;
};

class AssistantFormateur : Eleve	// AssitantFormateur hérite d'Eleve
{					// et donc aussi de Personne
	char* labo;
	AssistantFormateur(char* n, char* l, int m)
	{
		strcpy(nom, n); strcpy(labo, l);
		moyenne = m;
	}
};
```

6.5.3. Contrôle d'accès aux classes de base

Vous avez-vu dans l'exemple précédent qu'une classe dérivée peut aussi servir de base à une autre classe. Toutefois, des modificateurs d'accès doivent être utilisés pour définir les propriétés et les méthodes de la classe de base auxquelles la classe dérivée a accès.

En effet lors de l'héritage d'une classe de base on peut préciser un des 3 mots clés pour préciser le contrôle d'accès : **public**, **protected**, **private**, par exemple :

```
class Eleve : public Personne
```

Par défaut, si aucun modificateur d'accès n'est précisé, C++ utilise le modificateur **public** pour les classes déclarées avec le mot clé **struct** et **private** pour les classes déclarées avec le mot clé **class**.

Le modificateur d'accès agit différemment sur les membres de la classe de base selon qu'ils sont **public**, **protected**, **private**.

On peut résumer les effets des modificateurs d'accès dans un tableau :

Niveau de visibilité dans la classe dérivée		Modificateur d'accès de la classe de base		
		public	**protected**	**private**
Niveau de visibilité dans la classe de base	**public**	public	protected	private
	protected	protected	protected	private
	private	Accès interdit	Accès interdit	Accès interdit

Grâce au tableau, on comprend la nécessité de déclarer au moins **protected** les attributs des classes « Personne » et « Eleve » dans l'exemple ci-dessus, de même que l'utilisation du modificateur d'accès **public** lorsque la classe « Eleve » hérite de « Personne ».

6.5.4. Constructeurs d'une classe dérivée

Le constructeur d'une classe dérivée est défini comme le constructeur de n'importe quelle autre classe. Cependant, on peut utiliser les listes d'initialisation afin d'utiliser les constructeurs des classes de base.

```
class Personne
{
protected:
	char* nom;
public:
	Personne(char* pNom) : nom(pNom) {}
};

class Eleve : public Personne
{
protected:
	int moyenne;
public:
	Eleve(char* pNom, int pMoyenne) : Personne(pNom), moyenne(pMoyenne) {}
};
```

Dans cet exemple, le constructeur de la classe « Eleve », fait appel au constructeur de la classe « Personne » afin d'initialiser l'attribut « nom » de l'objet créé.

6.5.5. Accès aux méthodes de la classe de base

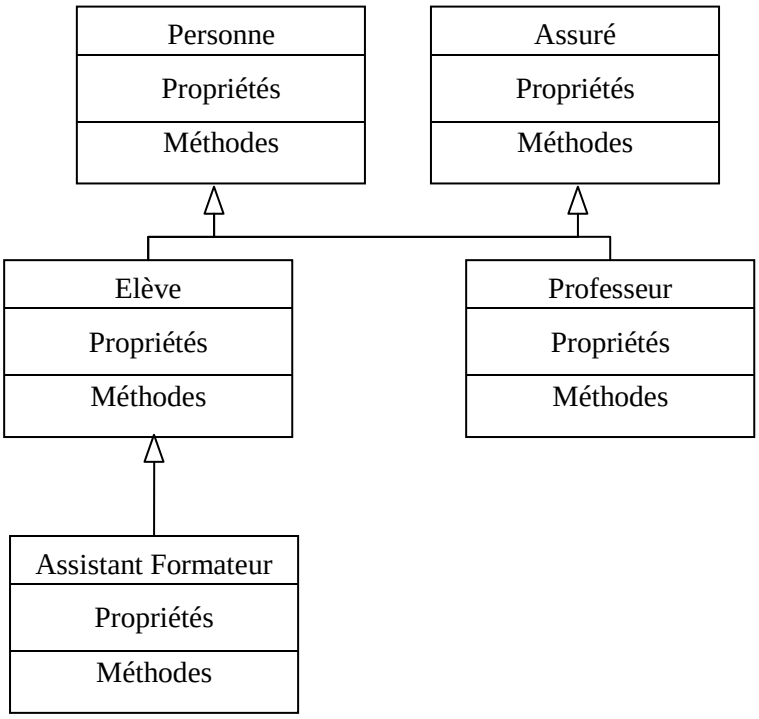
Afin d'éviter le code redondant, il est possible d'appeler une méthode d'une classe de base que l'on souhaite redéfinir. On utilise pour cela l'opérateur de modification de portée (**::**) en spécifiant le nom de la classe de base.

```
class Personne
{
protected:
	char* nom;
public:
	Personne(char* pNom) : nom(pNom) {}
	void infos() { cout << nom; }
};

class Eleve : public Personne
{
protected:
	int moyenne;
public:
	Eleve(char* n) : Personne(n) {}
	void infos() {
		Personne::infos();  // On appelle la méthode de la classe de base avant
		                    // de rajouter des fonctionnalités
		cout << ", Eleve";
	}
};
```

6.5.6. Héritage multiple

En C++ on peut créer des classes dérivées qui héritent de plusieurs classes de base simultanément.



Dans ce cas, le schéma de dérivation est un graphe dans lequel chaque nœud peut avoir plusieurs parents, et donc il peut exister plusieurs chemins d'un nœud vers la racine. Dans l'exemple ci-dessus, le graphe n'est en effet, même pas un arbre.

Bien que puissante, la possibilité d'hériter de plusieurs classe de base soulève beaucoup de problèmes, qu'il vaut mieux maîtriser avant de l'utiliser. D'ailleurs certains langages orientés objet comme Java ou C# ne permettent pas l'héritage multiple.

Nous n'aborderons qu'un seul cas dans lequel l'héritage multiple est intéressant, puissant, mais toutefois facile à utiliser. Il s'agit de l'implémentation d'interfaces (voir 6.5.8).

A part cette utilisation particulière nous déconseillons l'héritage multiple, qui produit du code illisible et difficile à maintenir.

6.5.7. Fonctions virtuelles – Polymorphisme

Le polymorphisme est l'une des caractéristiques les plus importantes de la programmation objet et qui lui confère une grande partie de sa puissance.

Il s'agit, en résumé, d'être capable d'avoir des méthodes avec des signatures identiques (même nom, nombre et type de paramètres et valeur de retour) mais avec des définitions différentes.

Dans notre exemple, nous souhaiterions afficher les informations concernant chaque Eleve, Professeur et AssistantFormateur d'une école. Pour chaque personne, nous aimerions afficher son nom, ainsi que ses rôles.

Dans un premier temps nous allons écrire un méthode appelée « infos() » pour afficher ces informations.

```
class Personne
{
protected:
	char* nom;
public:
	Personne(char* pNom) : nom(pNom) {}
	void infos() { cout << nom; }
};

class Eleve : public Personne
{
protected:
	int moyenne;
public:
	Eleve(char* n) : Personne(n) {}
	void infos() {
		Personne::infos();
		cout << ", Eleve";
	}
};

class Professeur : public Personne
{
	char* matiere;
public:
	Professeur(char* n) : Personne(n) {}
	void infos() {
		Personne::infos();
		cout << ", Professeur";
	}
};



class AssistantFormateur : public Eleve
{
	char* labo;
public:
	AssistantFormateur(char* n) : Eleve(n) {}
	void infos() {
```

```
            Eleve::infos();
            cout << ", Assistant Formateur";
      }
};

int main()
{
      Personne p("Jean");
      Eleve e("Pascal");
      Professeur prof("Pierre");
      AssistantFormateur af ("Mathieu");

      p.infos();
      cout << endl;
      e.infos();
      cout << endl;
      prof.infos();
      cout << endl;
      af.infos();
}
```

Ce programme affiche :

```
Jean
Pascal, Eleve
Pierre, Professeur
Mathieu, Eleve, Assistant Formateur
```

On voit que chaque objet a appelé la méthode qui lui correspondait pour afficher le nom de la personne et ses rôles.

Toutefois, si on crée un pointeur de type « Personne » pour essayer d'avoir la même fonctionnalité, on n'obtient pas l'affichage souhaité :

```
int main()
{
      Personne* p = new Personne("Jean"); // Pointe vers un objet de type Personne
      p->infos();
      cout << endl;
      delete p;

      p = new Eleve("Pascal");   // Pointe vers un objet de type Eleve
      p->infos();
      cout << endl;
      delete p;

      p = new Professeur("Pierre");    // Pointe vers un objet de type Professeur
      p->infos();
      cout << endl;
      delete p;

      p = new AssistantFormateur("Mathieu"); // Pointe vers un objet de type
                                             // AssistantFormateur
      p->infos();
      cout << endl;
      delete p;
      p = 0;
}
```

Affichage :

```
Jean
Pascal
Pierre
Mathieu
```

Ici, le programme a traité tous les objets comme étant de type « Personne ».

Pour résoudre ce problème, il suffit de déclarer la méthode « infos() » comme étant virtuelle, à l'aide du mot clé **virtual** dans chacune des classes.

```
class Personne
{
protected:
	char* nom;
public:
	Personne(char* pNom) : nom(pNom) {}
	virtual void infos() { cout << nom; }
};
...
```

Le programme affiche maintenant :

```
Jean
Pascal, Eleve
Pierre, Professeur
Mathieu, Eleve, Assistant Formateur
```

Que s'est-il passé ?

Sans **virtual** le compilateur a associé le pointeur « p » à la classe « Personne » (liaison ou ligature statique) ***avant*** l'exécution (*early binding*).

Avec **virtual** le pointeur est associé à différentes classes ***pendant*** l'exécution. Ceci est appelé liaison dynamique (*late binding*).

6.5.8. Classes abstraites et interfaces

Dans l'exemple précédent, afficher les informations d'une personne a un sens. Il est donc utile de mettre du code dans la méthode virtuelle « infos() » de la classe de base « Personne ». Mais, il arrive que la sémantique de la classe de base soit si abstraite, qu'on n'ait pas de code à mettre dans la méthode virtuelle.

Par exemple, si on avait une classe « FigureGeometrique » il n'y aurait rien à mettre dans la méthode virtuelle « dessiner() » puisqu'on ne sait pas quelle type de figure géométrique on instancie.

On appelle les méthodes virtuelles sans code à l'intérieur, méthodes virtuelles pures.

Si une classe admet au moins une méthode virtuelle pure on dit que c'est une **classe abstraite**. Ce qui veut dire qu'elle ne peut jamais être instanciée, puisque certaines méthodes ne sont pas codées.

En revanche il peut y avoir des classes dérivées. Ces classes doivent alors implémenter toutes les méthodes virtuelles pures. Elles ne sont donc plus abstraites et peuvent être instanciées.

Une classe qui n'a que des méthodes virtuelles pures et aucun attribut est appelée une **interface**. Ce concept est très puissant et très utilisé. Il est à la base de l'architecture Microsoft COM, sur laquelle reposent tous les systèmes type NT/XP.

6.5.9. Destructeurs virtuels

En C++, le destructeur d'une classe de base doit être virtuel du moment que la classe de base est polymorphe, c'est-à-dire, qu'elle a au moins une méthode virtuelle.

Dans le cas contraire, le destructeur d'une classe dérivée n'est pas appelé par polymorphisme, ce qui comporte des problèmes lors de la libération de la mémoire.

6.6. Fonctions, méthodes et classes amies d'une classe

Bien qu'il ne soit pas très recommandé, il est possible en C++ de violer le mécanisme d'encapsulation définie par les niveaux de visibilité des membres d'une classe.

Concrètement, on peut décider de donner un accès publique à tous les membres d'une classe à :

- une fonction
- une méthode particulière d'une autre classe
- toutes les méthodes d'une autre classe

Voici un exemple pour illustrer la syntaxe :

```
class Amie;
class Hote;
class SemiAmie
{
   char * AfficheHote( Hote & hote );
   ...
};

class Hote
{
      friend char * SemiAmie::AfficheHote( Hote &);
      friend char * FctAmie(Hote &);
      friend class Amie;
   ...
};

char * FctAmie( Hote & hote );
```

Dans cet exemple, nous avons tout d'abord déclaré la méthode « AfficheHote » de la classe « SemiAmie » comme étant amie de la classe « Hote » en rajoutant le prototype de la méthode, précédé du mot clé **friend**, dans la définition de « Hote ». Ceci permettra à la méthode « AfficheHote » d'accéder à tous les membres de la classe « Hote » comme s'ils étaient publics. Cependant, d'autres méthodes de la classe « SemiAmie » n'auront pas ce privilège.

De la même manière, on a donné le statut s'amitié à la fonction globale « FctAmie ». Elle aussi pourra accéder à tous les membres de la classe « Hote » comme s'ils étaient publics.

Finalement, toujours dans la définition de la classe « Hote », on a précédé la déclaration de la classe « Amie » du mot clé **friend**, ce qui est équivalent à rendre amies toutes les méthodes de cette classe.

Remarquez que toues les classes doivent au moins être déclarées avant d'être utilisées. Etant donné que la classe « Hote » est utilisée dans le prototype de la méthode « AfficheHote », il a fallu déclarer cette classe avant la définition de la classe « SemiAmie ». De même pour la classe « Amie ».

Notez que l'amitié entre classes n'est pas réflexive : A peut être amie de B sans que B soit amie de A. Elle n'est pas non plus transitive : A amie de B et B amie de C n'implique pas A amie de C.

Notez aussi que l'amitié à une classe de base n'a aucune influence implicite sur les classes dérivées.

Bien que le fait de violer le principe d'encapsulation reste dangereux, et à utiliser avec précaution, l'utilisation de fonctions, méthodes et classes amies peut s'avérer très utile lors de la surcharge d'opérateurs qui sera abordée dans le chapitre suivant.

7. Surcharge des opérateurs (operator overloading)

7.1.1. Introduction

En C++ il est possible de surcharger les opérateurs de base, afin de les utiliser avec des instances des classes définies par le développeur.

Ceci veut dire que si on a créé une classe « Point », il est tout à fait concevable de surcharger l'opérateur − pour calculer la distance entre deux points.

La majorité des opérateurs (45) peuvent être surchargées. Entre autres :

```
+       -       *       /       &       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

Cependant, ne peuvent pas être surchargés :

- **.** opérateur de sélection de membre de classe
- **::** opérateur de résolution de portée
- **?:** opérateur d'expression conditionnelle
- **sizeof** opérateur « taille de »

Certaines restrictions s'appliquent aux opérateurs surchargés :

- on ne peut pas changer l'***arité*** d'un opérateur (nombre d'opérandes)
- on ne peut pas changer la ***priorité*** des opérateurs
- on ne peut pas changer l'***associativité*** des opérateurs
- on ne doit pas changer la ***signification*** d'un opérateur (on pourrait rapidement rendre le code incompréhensible)
- on ne peut pas inventer de nouveaux opérateurs
- l'opérateur doit porter sur au moins un objet d'une classe.

7.1.2. Surcharge interne et externe des opérateurs

Un opérateur peut être surchargé de deux manières :

- interne : l'opérateur est une **méthode** de la classe.
- externe : l'opérateur est une **fonction amie** de la classe.

On doit donc, tout d'abord, être capable de représenter un opérateur comme une fonction.

Soit **@** un opérateur surchargé à deux opérandes (ex: +, *, &&, <, etc.)

On peut l'utiliser de trois manières différentes :

1) objet1 = objet2 @ objet3 ;

   (écriture générale de l'opération @ appliquée aux deux opérandes)

2) objet1 = objet2.**operator**@(objet3);

   (équivaut à 1. lorsque la fonction « operator@ » est membre de la classe)

3) objet1 = **operator**@(objet2, objet3);

   (équivaut à 1. lorsque la fonction « opérateur@ » est déclarée en dehors de la classe ; dans ce cas elle doit être déclarée **friend**).

Seules les syntaxes 2 et 3 peuvent être utilisées pour déclarer et définir un opérateur surchargé.

7.1.2.1. Surcharge interne

Nous allons tout d'abord surcharger l'opérateur somme (+) pour la classe « Point » qui a été utilisée tout au long de ce document en utilisant une méthode de la classe.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);

	Point operator+(const Point&);   // Protoype de l'opérateur surchargé

	void afficher() const
	{
		cout << "(" << x << "; " << y << ")";
	}
};

Point Point::operator+(const Point& operandeDroite)
{
	Point p;
	p.x = x + operandeDroite.x;
	p.y = y + operandeDroite.y;
	return p;
}
```

Lorsque l'on utilise la surcharge interne, l'objet appelant est toujours l'opérande de gauche. Dans le cas d'un opérateur binaire (comme la somme dans l'exemple ci-dessus), l'opérande de droite est passé à la fonction en tant que paramètre.

Notez que les opérandes sont passés sous forme de références constantes. Ceci par souci d'efficacité et de sécurité.

On peut tester l'opérateur défini ci-dessus avec le code suivant :

```
int main()
{
	Point p1(5, 6), p2(3, 2);
	Point p3 = p1 + p2;
	p3.afficher();			// (8; 8)
}
```

7.1.2.2. Surcharge externe

Nous allons maintenant surcharger le même opérateur que dans la section précédente, mais en une fonction externe et non plus une méthode de la classe.

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	Point(const Point& p) : x(p.x), y(p.y) {}
	Point(int* i) : x(i[0]), y(i[1]) {}
	explicit Point(float*);

	friend Point operator+(const Point&, const Point&);  // Protoype fct amie

	void afficher() const
	{
		cout << "(" << x << "; " << y << ")";
	}
};

Point operator+(const Point& operandeGauche, const Point& operandeDroite)
{
	Point p;
	p.x = operandeGauche.x + operandeDroite.x;
	p.y = operandeGauche.y + operandeDroite.y;
	return p;
}
```

Lorsque l'on utilise la surcharge externe, tous les opérandes sont passés en paramètre à la fonction. Dans le cas d'un opérateur binaire, le premier paramètre correspond à l'opérande de gauche et le deuxième à l'opérande de droite.

Etant donné que la fonction devra probablement faire appel aux attributs des objets passés en paramètre, il est nécessaire de déclarer la fonction comme étant amie de la classe.

On obtient avec cette nouvelle syntaxe les mêmes résultats que précédemment.

Savoir utiliser la surcharge interne ou externe, dépend du contexte et de l'opérateur. Par exemple pour additionner un **int** et un « Point » il vaut mieux définir l'opérateur en externe puisqu'on, ne peut toucher au type **int**.

On présente par la suite quelques opérateurs à utiliser plus subtilement et des exemples pour faciliter la compréhension. Néanmoins cette présentation est assez brève pour un sujet assez vaste.

7.1.3. L'opérateur d'affectation (*operator*=)

Bien que le compilateur soit capable de créer des constructeurs par copie, des destructeurs et des opérateurs d'affectation par défaut, on peut dire qu'il existe une règle en C++ qui dit que si on a besoin de redéfinir l'une de ces trois méthodes, il faut les redéfinir toutes.

En effet, si l'on a besoin d'une de ces méthodes, cela veut dire qu'un des attributs de la classe est dynamique, auquel cas il faut coder les trois méthodes et ne pas compter sur les méthodes par défaut de C++.

Il y a trois règles à respecter lorsqu'on code un opérateur d'affectation :

- Prévoir le cas ou on s'auto-affecte.
- Retourner une copie de l'objet courant (**return *this**) afin de pouvoir enchaîner les opérateurs d'affectation.
- Il ne faut pas utiliser le constructeur de copie dans l'opérateur d'affectation et vice-versa. Par contre il est utile de coder une fonction « copie() » appelée par le constructeur de copie et l'opérateur d'affectation.

```
class Rational {
public:
	void operator=(const Rational&);  //déclaration de la fonction operator=
private:
	int num;
	int den;
};

void Rational::operator=(const Rational& r)
{
	num = r.num;
	den = r.den;
}

int main( )
{
	Rational x,  y,  z(22, 7) ;
	x = y = z;		// ERREUR : car equivalent à : x = ( y = z )
				// mais (y = z) retourne void , donc le résultat ne peut
				// pas être assigné à x (il faudrait définir une fonction
				// operator=(void)
}
```

L'exemple ci-dessus ne tient pas compte des règles mentionnées ci-dessus. Quelques modifications mineures suffisent pour les respecter :

```
class Rational {
public:
	Rational& operator=(const Rational&);  // L'operator= retourne un Rational
private:
	int num, den;
};

Rational& Rational::operator=(const Rational& r)
{
	if(&r == this)		// Si l'objet assigné (r) est le même que l'objet courant
		return *this;// (this) sortir sans rien faire

	num = r.num;
	den = r.den;
	return *this;		// *this retourne la valeur de l'objet affecté
}
```

```
int main( )
{
	Rational x,  y,  z(22, 7) ;
	x = y = z;		// equivalent à :x x = ( y = z )
				// *this retourne la valeur de l'opération (y = z) et qui
				// peut être à son tout affectée à x
}
```

Cette fois, la fonction qui redéfinit l'opérateur d'assignation retourne une copie de l'objet courant en utilisant le pointeur **this**, ce qui permet l'enchaînement des opérations d'assignation.

La fonction vérifie aussi le cas de l'auto-affectation en comparant l'adresse de l'objet courant (valeur du pointeur **this**) à celle de l'objet passé en paramètre. Si les adresses coïncident, il s'agit d'une auto-assignation (x = x), aucune opération n'est donc pas nécessaire. Il suffit de retourner, comme avant, une copie de l'objet courant.

Il est important de noter que **operator=** est uniquement l'opérateur d'***assignation*** et non pas d'initialisation, bien qu'on utilise le symbole = pour celle-ci aussi.

En effet, dans le cas d'une initialisation, l'opérande de gauche n'existe pas. C'est donc le ***constructeur par copie*** qui est utilisé. D'où la règle mentionnée précédemment qui dit que si on a besoin de définir un opérateur d'assignation, il est fort probable qu'on ait besoin de définir un constructeur par copie, et vice-versa.

7.1.4. L'opérateur d'indexation ([ ])

En plus des tableaux, d'autres structures de données se prêtent à l'indexation. Même la classe « Rational » définie précédemment peut servir d'exemple.

```
class Rational {
public:
	int operator[](const int &indice);  // indice
private:
	int num, den;
};

int Rational::operator[](const int &indice)
{
	if(indice == 0)		// On condière l'indice 0 équivalent au numérateur
		return num;
	if(indice == 1)		// et l'indice 1 équivalent au dénominateur
		return den;

	// Dans le cas contraire, afficher une erreur, et sortir
	cerr << "Index undefined" << endl;
	exit(0) ;
}

int main( )
{
	Rational x(22, 7);
	cout << x[0];		// 22
	cout << x[1];		// 7
	cout << x[2];		// ERREUR
}
```

7.1.5. Opérateurs avec plusieurs arités

Certains opérateurs, tel que −, peuvent avoir une ou deux opérandes selon le contexte.

L'exemple ci-dessus reprend la classe « Rational » et définit les deux cas de l'opérateur − :

- opérateur unaire de négation (inverse le signe du numérateur)
- opérateur binaire de soustraction

```
class Rational {
public:
	Rational operator-(void);			// opérateur de négation
	Rational operator-(const Rational& r);	// opérateur de soustraction
	Rational& operator=(const Rational&);	// opérateur d'assignation
```

```
        Rational& operator-=(const Rational&); // opérateur combiné
private:
        int num, den;
};

Rational& Rational::operator-=(const Rational& r)
{
        *this = *this - r;
        return *this;
}

int main( )
{
        Rational x(22, 7), y(3, 14), z;
        z = x - y;    // z.num = 287, z.den = 98
}
```

Ayant défini l'opérateur d'assignation dans la section précédente, et l'opérateur de soustraction ci-dessus, il est possible de définir l'opérateur « combiné » −= :

```
class Rational {
public:
        Rational operator-(void);               // opérateur de négation
        Rational operator-(const Rational& r); // opérateur de soustraction
private:
        int num, den;
};

Rational Rational::operator-(void)                    // opérateur de négation
{
        num = -num;
        return *this;
}

Rational Rational::operator-(const Rational& r)      // opérateur de soustraction
{
        return Rational(num * r.den - r.num * den, den * r.den);
}

int main( )
{
        Rational x(22, 7), y(3, 14);
        x -= y;       // x.num = 287, x.den = 98
}
```

7.1.6. Les opérateurs d'incrémentation et de décrémentation

C++ fait la différence entre l'utilisation de ces opérateurs en suffixe ou en préfixe. De plus, ces opérateurs sont unaires, donc travaillent sur l'objet lui-même.

Il a donc fallut établir une convention pour pouvoir différencier l'utilisation de ces opérateurs en préfixe ou en suffixe, dont voici un exemple :

```
class Rational {
public:
        Rational operator++(int);               // opérateur suffixé (a++)
        Rational& operator++(void);             // opérateur préfixé (++a)
private:
        int num, den;
};

// Dans la postincrémentation, on retourne la valeur et ensuite on l'incrémente
Rational Rational::operator++(int)
{
        Rational temp(num, den);   // On crée une copie de l'objet
        num += den;                // On incérmente les valeurs de l'objet original
```

```
	return temp;			// Mais, on retourne la copie, dont la valeur n'a
					// pas été pas incrémentée
	// Notez que le paramètre int n'a pas été utilisé
}

// Dans la préincrémentation, on incrémente la valeur et ensuite on la retourne
Rational& Rational::operator++(void)
{
	num += den;			// On incérmente les valeurs
	return *this;			// On retourne l'objet
}

int main( )
{
	Rational x(22, 7), y;
	y = x++;	// y.num = 22, y.den = 7; x.num = 29, x.den = 7
	y = ++x;	// y.num = 36, y.den = 7; x.num = 36, x.den = 7
}
```

7.1.7. Les opérateurs relationnels

Les opérateurs relationnels peuvent être surchargés comme n'importe quel autre opérateur de la liste mentionné précédemment. Toutefois, bien que le compilateur ne l'impose pas, il est fortement recommandé de les faire retourner une variable de type **bool**.

En effet, on pourrait surcharger les opérateurs pour retourner n'importe quel autre type de valeur, mais le code deviendrait très vite incompréhensible.

```
class Rational {
public:
	// fonction amie pour surcharger la comparaison
	friend bool operator==(const Rational&, const Rational&);
private:
	int num, den;
};

bool operator==(const Rational& r1, const Rational& r2)
{
	return (r1.num * r2.den == r2.num * r1.den);
}
```

7.1.8. L'opérateur de redirection (<<)

La surcharge de cet opérateur permet l'utilisation d'objets directement dans la redirection d'un flot, par exemple lors de l'impression à l'écran.

Par exemple, on souhaiterait que le code suivant :

```
Rational r(1, 2);
cout << r << endl;
```

Affiche :

```
1/2
```

La signature de l'opérateur de redirection est toujours la même :

- Pour une surcharge interne :

```
ostream& operator<<(ostream&);
```

- Pour une surcharge externe :

```
friend ostream& operator<<(ostream&, const T&);
```

où T est le type pour lequel on souhaite surcharger l'opérateur

Voici deux exemples pour illustrer les deux méthodes :

```
class Rational {
public:
	ostream& operator<<(ostream&);
private:
	int num, den;
};

ostream& Rational::operator<<(ostream& os) {
	return os << num << '/' << den;
}

int main( ) {
	Rational x(22, 7);
	x << cout;	// Affiche : 22/7
			// Notez l'inversion des objets x et cout
}
```

```
class Point
{
	int x, y;

public:
	Point(int a = 0, int b = 0) : x(a), y(b) {}
	friend ostream& operator<<(ostream&, const Point&);
};

ostream& operator<<(ostream& os, const Point& p)
{
	return os << '(' << p.x << " ; " << p.y << ')';
}
int main( )
{
	Point p(22, 7);
	cout << p;	// Affiche : (22 ; 7)
			// L'inversion des objets x et cout n'est plus nécessaire
}
```

7.1.9. L'opérateur de transtypage (*cast*)

On a vu comment utiliser des constructeurs pour faire des opérations de transtypage d'une classe source vers une classe cible. Cependant, avec cette méthode, il est nécessaire que ce constructeur soit dans la classe cible.

Il se peut que le code de la classe cible ne soit pas modifiable. Dans ce cas, on peut définir l'opérateur de transtypage dans la classe source.

L'opérateur de transtypage ne doit être utilisé que dans ce cas, car il requiert une copie de l'objet retourné vers l'objet cible en plus (donc moins performant).

De plus si le constructeur de copie et l'opérateur de transtypage sont disponibles, le compilateur choisira le constructeur de copie, a moins que le constructeur de copie soit explicite et que le transtypage ne le soit pas. Ceci relève néanmoins d'une mauvaise conception.

Voici un exemple où on veut convertir un « Rational » en un **float**. On ne peut pas faire autrement qu'utiliser un opérateur de transtypage puisqu'on ne peut ajouter un constructeur de transtypage au type **float** :

```
class Rational {
public:
```

```
        Rational(int a = 0, int b = 1) : num(a), den(b) {}
        operator float() const;          // Pas de type de retour !
private:
        int num, den;
};

Rational::operator float() const {
        return (float)num / den;
}

int main( ) {
        Rational x(22, 7);
        float f = x;                     // f = 3.14286
}
```

8. Espaces de noms (*namespaces*)

8.1. Introduction

Un problème survient souvent dans les grosses architectures utilisant de nombreuses librairies et fonctions développées séparément : il y a souvent collision entre les identifiants.

Par exemple un logiciel de cartographie anglophone, définira la classe « map » comme un point d'accès à une carte alors que la librairie standard définit la classe « map » comme une classe d'association entre objets.

Heureusement les espaces de noms résolvent totalement ce problème.

Les espaces de noms, ou *namespaces*, permettent de regrouper de manière logique des éléments (variables, fonctions, classes, etc.) pour éviter des conflits entre différentes parties d'un même projet. Ces espaces sont utilisés par le compilateur pour limiter sa recherche quand il rencontre un identificateur.

8.2. Espaces de noms nommés

Bien que l'on puisse utiliser des espaces de noms sans leur attribuer un identifiant (espaces de noms anonymes), cette pratique est peu recommandée et on ne l'abordera pas dans ce document.

La syntaxe pour déclarer un espace de noms nommé est la suivante :

```
namespace nom_de_l_espace
{
	// Déclarations | Définitions
}
```

Voici un exemple qui résout la collision de la classe de la librairie standard « map » et la classe personnelle « map » :

```
#include <map>		// ce fichier de la librairie standard définit une classe 'map',
			// à l'intérieur d'un espace de nommage nommé std
namespace MesMaps
{
	class map
	{
		void Methode();
	};

	void fct(map m, double);
}

void MesMaps::fct(MesMap::map m, double d){...}
void MesMaps::map::Methode(){...}

MesMaps::map m1,m2;			// instancie ma classe
std::map<int, double> m; // instancie la classe map de la librairie standard
```

Cette exemple montre bien le fonctionnement des espaces de noms.

A l'intérieur de notre espace « MesMaps » on donne les noms que l'on veut à nos classes et nos fonctions.

A l'extérieur de l'espace de noms on accède à nos classes et nos fonctions en les préfixant par « MesMaps:: ».

On peut imbriquer les espaces de noms :

```
namespace A
{
	namespace B
	{
		namespace C
		{
			void fct();
		}
	}
}
```

On accède à « fct » avec :

```
A::B::C::fct();
```

On peut scinder un espace de nommage pourvu qu'on garde le même nom :

```
namespace A { int i; }
namespace B { int j; }
namespace A { int j; }  // extension de l'espace de nommage A
```

8.3. Alias d'espace de nommage

Lorsqu'un espace de nommage porte un nom très compliqué, il peut être avantageux d'utiliser un alias de ce nom :

```
namespace nom_alias = nom_complique_a_renomer;
```

Encore une fois il faut faire attention à ne pas générer des collisions d'identificateurs.

8.4. La déclaration *using*

La déclaration **using** permet l'utilisation d'un identificateur déclaré dans un espace de nommage sans avoir à utiliser l'opérateur de résolution de portée :

```
namespace A
{
	int i;
	int fct(int j);
}

using A::i;
using A::fct;

i = fct(67);   // utilise i et fct de l'espace de nommage A
```

C'est grâce à ce mot clé que l'on peut utiliser les objets **cin** et **cout** sans les préfixer du nom de leur espace de nom **std::**.

```
#include <iostream>
using std::cout;

cout << "Hello World";
```

8.5. La directive *using namespace*

Avec **using** on peut importer soit des éléments individuels d'un *namespace*, soit tous les éléments de celui-ci.

```
namespace A
{
	int i;
	int fct(int j);
}

using namespace A;

i = fct(67);		// utilise i et fct de l'espace de nommage A
```

Il semblerait que cette pratique enlève le bénéfice de l'utilisation des espaces de nommage, mais dans le cas où il n'y a pas de collisions d'identificateurs, c'est beaucoup moins fastidieux que d'utiliser la déclaration **using** pour tous les identificateurs.

9. Programmation générique (*templates*)

9.1. Introduction

La généricité est un concept qui peut s'appliquer sur des fonctions ou sur des classes. Ce concept s'appelle *template* en anglais. Il permet de définir des fonctions (ou des classes) dont les arguments (respectivement, les données) n'ont pas un type fixé d'avance.

Par exemple :

- on souhaiterait avoir une fonction qui retourne le minimum des 2 paramètres qui lui sont passés, quel que soit leur type, du moment qu'il existe une surcharge de l'opérateur de comparaison pour ce type.
- on veut utiliser une classe qui définit le comportement d'une pile, quel que soit le type T des éléments de la pile. Les opérations « push(T element) » et « T pop() » doivent être implémentées.

Le développeur doit alors fournir la liste des types utilisés, de façon à ce que, au moment où l'on utilise un *template* de fonction, ou de classe, le compilateur puisse générer le code correspondant aux types indiqués. Cette étape est appelée génération des *templates*. Dans le fichier exécutable on a donc bien le code de la fonction ou de la classe, dupliqué pour chaque liste de types utilisée.

Une liste de types se présente sous la forme :

```
template <class type1, ..., typename typeN>.
```

Comme on le voit on peut indifféremment utiliser les mots clés **class** ou **typename**.

On a la possibilité de définir des types par défaut. Comme pour la surcharge des fonctions ces types se trouvent à la fin de la liste :

```
template <typename T, typename U, typename V = int, typename W = double>
```

On appelle souvent les types, comme dans l'exemple ci-dessus, par une seule lettre, en commençant par un T majuscule.

9.1.1. Les fonctions génériques

La déclaration et la définition d'une fonction générique se fait exactement comme si elle était une fonction normale, à ceci près quelle est précédée de la liste des types génériques. Voici un exemple de code d'une fonction générique qui retourne le maximum des ses 3 arguments, quel que soit leur type du moment qu'il surcharge l'opérateur '>' :

```
template <typename T> T max( T param1, T param2, T param3 )
{
      if( param1 > param2 && param1 > param3 ) return param1 ;
      if( param2 > param1 && param2 > param3 ) return param2 ;
      return param3;
}

int main() {
      int i = max(1 , 3 , 5);        // i = 5
      double d = max(1.1 , 3.1 , 5.1);     // d = 5.1
}
```

On commence par définir un type générique « T » dans la liste de types. On écrit ensuite le protype de la fonction « max » en indiquant qu'elle retourne une valeur de type « T » et qu'elle attend trois paramètres de type « T » aussi.

On peut ensuit faire appel à cette fonction dans le programme, en lui passant des paramètres de n'importe quel type, pour autant qu'ils puissent être comparés avec l'opérateur « > ».

Bien que la fonction n'ait été définie qu'une fois dans le code source, le compilateur crée en fait 2 fonctions « max() ». Une où le type « T » est **int** et une ou le type « T » est **double**.

9.1.2. Les classes génériques

La déclaration d'une classe générique se fait exactement comme si elle était une classe normale, à ceci près quelle est précédée de la liste des types génériques. Exactement comme une fonction générique.

Voici un exemple de code d'une classe dont les objets sont des vecteurs de dimension 3. Le type des éléments est générique. On définit les opérations produit vectoriel '^' et produit scalaire '*':

```
template <typename T> class Vecteur3D {
private:
	T m_x, m_y, m_z;

public:
	Vecteur3D(T x = 0, T y = 0, T z = 0): m_x(x), m_y(y), m_z(z) {}

	Vecteur3D operator^( Vecteur3D& arg ) const {	// produit vectoriel
		Vecteur3D tmp;
		tmp.m_x = m_y * arg.m_z - m_z * arg.m_y;
		tmp.m_y = m_z * arg.m_x - m_x * arg.m_z;
		tmp.m_z = m_x * arg.m_y - m_y * arg.m_x;
		return tmp ;
	}

	T operator*( Vecteur3D& arg ) const {		// produit scalaire
		return m_x * arg.m_x + m_y * arg.m_y + m_z * arg.m_z;
	}
};

int main()
{
	Vecteur3D<double> v1(1, 2, 3);
	Vecteur3D<double> v2(3, 5, 6);
	Vecteur3D<double> v3;
	v3 = v1 ^ v2;			// v3 = (-3, 6, -3)
	double d = v3 * v2;		// d = 0
}
```

Dans l'exemple ci-dessus, parce que toutes les méthodes de la classe étaient définies dans la déclaration de la classe, la liste de types n'a été spécifiée qu'une fois.

Toutefois, si des méthodes de la classe sont déclarées en hors de celle-ci, elles doivent elles aussi être déclarées génériques :

```
template <typename T> class Vecteur3D {
	...
	T operator*( Vecteur3D& arg ) const;
};

template <typename T> T Vecteur3D<T>::operator*( Vecteur3D<T> &arg ) const {
	return m_x * arg.m_x + m_y * arg.m_y + m_z * arg.m_z;
}
```

9.1.3. Les méthodes génériques

Mis à part les destructeurs, les méthodes d'une classe peuvent être génériques, que la classe soit elle-même générique ou pas.

Si la classe n'est pas générique, on utilise la même syntaxe que pour une fonction générique.

Si la classe est aussi générique, il faut spécifier 2 fois la syntaxe **template**, une fois pour la classe et une fois pour la fonction. Si la méthode générique est définie à l'intérieure de la classe, il n'est pas nécessaire de donner les paramètres génériques de la classe. Dans ce cas on utilise encore la même syntaxe que pour une fonction générique, comme cela a été fait dans l'exemple.

Notez qu'une méthode ne peut être virtuelle et générique à la fois.

Bien qu'il soit courant d'utiliser des fonctions et des classes génériques, il est plus rare de devoir utiliser des méthodes génériques, surtout si la classe est déjà générique.

9.1.4. Instanciation des *templates*

Lorsque tous les types d'une liste générique de types sont spécifiés, le compilateur génère l'instanciation des *templates*, c'est à dire qu'il fabrique le code de la fonction, classe ou méthode, en utilisant les types fournis.

Une contrainte assez lourde survient lorsqu'on instancie une classe générique (respectivement, qu'on appelle une fonction générique) : les corps de toutes les méthodes (respectivement, le corps de la fonction) doit être défini.

En plus de cette contrainte imposée au programmeur, qui doit modifier ses habitudes, cela implique que :

- Les instances des *templates* sont compilées pour chaque liste de type et pour chaque fichier objet, d'où une performance moindre du compilateur.
- Le code des instances des *templates* sont en multiples exemplaires dans les fichiers objets, d'où une augmentation de la taille de l'exécutable.

Toutefois, ces problèmes sont plus ou moins bien résolus dans les compilateurs modernes : fichiers d'entête précompilés, partage des instances de *templates* lors de l'édition de liens, etc.

Notez que l'instanciation des *templates* peut se produire de deux façons :

- Implicitement : l'instanciation implicite se produit, par exemple, lorsque le compilateur doit instancier une fonction générique à partir des types des paramètres d'appel et de retour, lors de l'appel d'une fonction générique dans le code.
- Explicitement : c'est une technique permettant au programmeur de forcer le compilateur à instancier un *template*. Par exemple, dans les exemples précédents, le programmeur peut rajouter une des lignes suivantes pour instancier explicitement la classe ou la fonction générique:

```
template int max(int, int, int);
template Vecteur3D<double>;
```

Il est vivement conseillé d'utiliser l'instanciation de *templates* explicite. D'ailleurs certains compilateurs permettent d'interdire l'instanciation implicite.

10. Les exceptions

10.1. Introduction

Des situations peuvent se présenter, dans l'exécution d'un programme, qui sont indépendantes du programmeur. On peut, par exemple, essayer d'accéder à un fichier qui n'existe pas, ou demander une allocation de mémoire alors qu'il n'y en a plus. Ces situations que, par abus de langage, on appelle erreurs, engendrent un arrêt du programme si elles ne sont pas traitées.

Pour les traiter on peut tester les codes d'erreurs retournés par les fonctions critiques, mais ceci présente deux inconvénients :

- le code devient lourd, puisque chaque appel à une fonction critique doit être suivi de nombreux tests
- le programmeur doit prévoir toutes les situations possibles dès la conception du programme, ainsi que définir les réactions du programme et les traitements à effectuer

En fait ces inconvénients sont majeurs, et il a fallu trouver d'autres solutions à ce problème : c'est la gestion des exceptions.

10.2. Principe de la gestion des exceptions

La gestion d'une exception se déroule en plusieurs étapes :

1) Le déroulement du programme est interrompu
2) Un objet contenant, éventuellement, des paramètres descriptifs de l'interruption est créé
3) Une exception est levée, paramétrée par l'objet créé à l'étape précédente
4) Deux choses peuvent alors arriver :
    a. un gestionnaire d'exceptions rattrape l'exception, l'analyse, et a la possibilité d'exécuter du code, par exemple, pour se récupérer de l'interruption.
    b. il n'y a pas de gestionnaire d'exception prévu pour ce type d'exception, donc le programme se termine.

10.3. Mots clés

La syntaxe de traitement des exceptions fait appel à trois mots clés :

- **try**
- **throw**
- **catch**

Voici un exemple :

```
struct ErrorAccesServeur
{
	unsigned long m_AddressIP;
	unsigned short m_Port;

	ErrorAccesServeur(unsigned long AddressIP, unsigned short Port) :
		m_AddressIP(AddressIP), m_Port(Port) {}
};

int main(void)
{
	try
	{
		...
		if( !PingServeur( AddressIP, Port ) )
			throw ErrorAccesServeur(AddressIP, Port);
		...
	}
	catch(ErrorAccesServeur &e) // Gestionnaire d'erreurs type
ErrorAccessServeur
	{
		// traitement de l'exception
	}
	catch(...)			// Gestionnaire d'erreurs par défaut
	{
		// ce gestionnaire rattrape toutes les erreurs, non déjà
		// rattrapées par un gestionnaire d'erreur précédent
		// Il est très pratique, par exemple pour assurer qu'un
		// serveur, ne plante jamais, mais on ne peut récupérer
		// aucune information sur l'erreur rattrapée
	}
}
```

Cette syntaxe est la même que celle d'autres langages comme Java ou C#.

Notez que l'on peut imbriquer les blocs try/catch.

Notez qu'un gestionnaire d'exceptions peut lever à son tour une exception avec le mot clé **throw**.

10.4. Algorithme du choix du gestionnaire d'exceptions

Lorsqu'une exception levée rencontre un gestionnaire d'exceptions elle ne sera prise en compte que si une des conditions suivantes est vérifiée :

- Le type exact de l'objet mentionné dans **throw** est rencontré.
- Un type correspondant à une classe de base de l'objet mentionné dans **throw** est rencontré.
- Un type correspondant à un pointeur sur une classe dérivée du type mentionné dans **throw** est rencontré.
- Le gestionnaire d'exception est de type **catch(...)**.

Enfin notez que le mot clé **throw** peut lever une exception sans objet associé. L'exception ne sera donc rattrapée que si elle rencontre un gestionnaire de type **catch(...)**.

10.5. Libération des ressources dynamiques lors d'une exception

Il faut bien comprendre que lorsqu'une exception est lancée, elle remonte pas à pas la pile des fonctions appelées jusqu'à ce qu'elle soit rattrapée par un gestionnaire d'exception approprié.

Ceci implique que les tous les objets déclarés dans les fonctions seront détruits (à part les objets statiques).

Malheureusement, les objets alloués dynamiquement dans ces fonctions ne seront pas détruits :

```
void fct()
{
	long i = 8 ;			// allocation statique d'un long
	long * p = new long(9);		// allocation dynamique d'un long

	throw;		// une exception est levée

	delete p;	// ce code n'est pas exécuté, par conséquent l'objet alloué
			// dynamiquement n'est pas libéré
	p = 0;
}

int main()
{
	try
	{
		fct();
	}
	catch(...)
	{
	}
}
```

Heureusement il existe une astuce pour résoudre ce problème.

Elle consiste à définir une classe qui contient un pointeur vers l'objet allouée dynamiquement. Dans le destructeur de la classe, on libère alors l'objet pointé.

Voici le même exemple corrigé:

```
struct long_ptr
{
	long * m_p;
	long_ptr( long * p ): m_p( p ) {}
	~long_ptr()
	{
		if( m_p != 0 )
			delete m_p;
	}
};

void fct()
{
	long i = 8 ;			// allocation statique d'un long
	long * p = new long(9);		// allocation dynamique d'un long
	long_ptr pptr(p);		// association du long alloué dynamiquement à un
					// objet de type long_ptr
	throw;				// une exception est levée
}					// lorsque l'on quitte la fonction le destructeur
					// de pptr est appelé et la mémoire libérée
```

11. *Standard Template Library* (STL)

La STL, Standard Template Library ou la librairie générique standard, est un ensemble de librairies définies par la norme ISO qui devraient être disponibles dans chaque implémentation du langage C++.

Le mot générique est utilisé pour montrer que la plupart des fonctionnalités disponibles le sont au travers de fonctions et de classes génériques. Par exemple on peut définir des listes d'objets, le type des objets étant un paramètre générique.

Concrètement la STL contient :

- Des types d'aides à la manipulation des chaînes de caractères.
- Des types numériques, comme les complexes.
- Des utilitaires généraux, comme les objets fonctions et les auto pointeurs.
- Des types d'aide à la gestion des flux d'entrée sortie.
- Des outils d'aide à la gestion des particularités locales (calendrier…).
- Des conteneurs qui permettent de stocker et de manipuler des collections d'objets.
- Des itérateurs qui aident à travailler sur des collections d'objets.
- Des algorithmes qui aident à consulter et à manipuler des collections d'objets.

Les déclarations requises pour utiliser la bibliothèque standard C++ se trouvent dans 32 fichiers en-tête :

```
<algorithm>  <bitset>     <complex>    <deque>      <exception>  <fstream>
<functional> <iomanip>    <ios>        <iosfwd>     <iostream>   <istream>
<iterator>   <limits>     <list>       <locale>     <map>        <memory>
<new>        <numeric>    <ostream>    <queue>      <set>        <sstream>
<stack>      <stdexcept>  <streambuf>  <string>     <typeinfo>   <utility>
<valarray>   <vector>
```

Les noms ci-dessus peuvent ne pas être des noms de fichier corrects, ou ne pas nommer des fichiers existants sur votre système ; la manière de les transformer en de vrais noms de fichier dépend du compilateur.

Les noms de tous les éléments de la bibliothèque standard, sauf les macros et les opérateurs **new** et **delete**, sont membres de l'espace de noms **std** ou d'espaces de noms imbriqués dans **std**.

Lorsque le risque de collision de nom est minime on peut donc utiliser la directive :

```
using namespace std;
```

Les éléments de la bibliothèque standard de C appartiennent aussi à la bibliothèque standard de C++ à travers les fichiers en-tête suivants, qui renvoient de manière évidente aux fichiers correspondants de la bibliothèque C :

```
<cassert>   <cctype>    <cerrno>    <cfloat>    <ciso646>   <climits>
<clocale>   <cmath>     <setjmp>    <csignal>   <cstdarg>   <cstddef>
<cstdio>    <cstdlib>   <cstring>   <ctime>     <cwchar>    <cwtype>
```

12. Bibliographie

Delannoy, Claude. *Programmer en C++*. Eyrolles, 2004.

Hubbard, J.R. *Programmation en C++*. McGraw Hill/Schaum, 1999.

Smacchia, Patrick. *Le langage de programmation C++*. 2002.