Les bases de la programmation C++

Mr Benameur / Mr Tondeur

1. La classe

Avant de pouvoir créer et manipuler un objet, un classe doit préalablement avoir été définie. La classe représente le type de donnée qui permet de définir des objets dans un programme.

L'encapsulation est le principe qui permet de regrouper les données et les fonctions au sein d'une *classe*.

1.a Définition des classes

Pour définir une classe, on utilise le mot clé *class* en respectant la syntaxe suivante :

```
class NomClasse
{
	// définition de la classe
} ;
```

1.b Déclaration des données membres

Les données membres sont les variables qui sont définies à l'intérieur d'une classe. On précise la portée de ces variable en les positionnant de manière avec les étiquettes *public*, *private* et *protected*.

```
class Point
{
	public :
		int x,y ;
} ;
```

1.c Déclaration des fonctions membres (ou méthodes)

Les fonctions membres correspondent aux fonctions définies au sein de la classe. Ces fonctions représentent l'ensemble des traitements que vous pouvez mettre en œuvre avec les objets de cette classe.

La surcharge de fonctions peut être utilisé avec les fonctions membres.

Il existe deux techniques pour définir des fonctions membres : *la définition déportée* et *la définition inline* implicite que l'on présentera plus loin.

La définition déportée

Ce type de définition est la plus courante. Elle consiste à définir le prototype de la fonction à l'intérieur de la classe, puis à définir le corps de la fonction en dehors de cette classe en utilisant le nom complet de la méthode (on utilise pour cela *l'opérateur de résolution de portée, ORP* '**::**').

```
#include <iostream.h>

class Point
{
	public :
		int x,y ;
```

```
            void Affiche() ;
} ;

void Point::Affiche()
{
      cout << x <<y ;
}
```

La définition inline

```
#include <iostream.h>

class Point
{
      public :
            int x,y ;

      void Affiche(){cout << x <<y ;}
}
```

1.d Les fonctions membres *const*

Le mot clé *const* qui sert à définir des constantes peut également être utilisé pour des fonctions membres d'une classe. Dans ce cas, le mot clé *const* doit être indiqué à la fois à la fin du prototype de la méthode et à la fin de la première ligne du corps de cette méthode.

Une fonction membre constante ne peut pas modifier les données membres de la classe.

On donne un exemple :

```
class Point
{
      private :
            int x,y ;

      public :
            void Affiche() const ;
}


void Point::Affiche() const
{
      cout << "x,y : " << x << y ;
}
```

1.e Les membres *static*

Le mot clé *static* peut également s'appliquer aux données ou au fonctions membres des classes dans un tout autre contexte ! On parle alors de membres de classe, en ce sens qu'un membre de classe appartient à la classe plutôt qu'à l'objet et qu'il est partagé par tous les objets plutôt que dupliqué.

Les membres statiques sont le seuls qui peuvent être utilisés sans créer d'objet.

Données membres statiques

Pour définir une donnée membre statique, il faut respecter les deux étapes suivantes :

- Faire précéder la déclaration de la donnée statique par le mot clé *static*.
- Créer cette variable en dehors de la classe.

```
class Point
{
	public :
		static int x ;
} ;

int Point::x = 100 ;
```

La création de la variable associée à la donnée membre statique doit obligatoirement accompagner la définition de la classe. En effet, contrairement aux autres données membres, cette variable ne sera pas allouée au moment de la création des objets de cette classe. Il est donc nécessaire de créer cette variable en dehors de la classe en utilisant une syntaxe similaire à celle employée pour les variables globales. Toutefois, il s'agit bien d'une donnée membre de la classe, qui respecte dans ces contions l'étiquette de protection (*private*, *public*, *protected*) qui lui a été attribuée.

Après avoir déclaré une donnée membre statique, on peut accéder à cette variable soit en utilisant un objet (ou un pointeur d'objet), soit en utilisant le nom de la classe directement :

```
Point::x = 70 ;
```

Une donnée membre définie comme une constante (mot clé *const*) doit être déclarée en statique. En effet, il n'existe aucune raison pour que chaque objet de la classe possède une version de cette constante.

Par exemple, la constante PI de la classe *Cercle* est déclarée en statique, et de cette manière, tous les objets de type Cercle partageront cette même constante.

N.B. : Une donnée membre statique peut être utiliser pour échanger des informations entre les différents objets d'une classe dans la mesure où ces objets ont tous accès à la même variable.

Fonctions membres statiques

Une fonction membre statique à l'instar d'une donnée membre statique ne sont pas utilisées pour un objet spécifique de la classe ; c'est d'ailleurs la seule fonction membre qui ne récupère pas le pointeur système *this*. Elle ne peut donc pas accéder aux données membres associées à un objet ou appeler une fonction membre non statique.

Comme les données membres statiques, une méthode peut être appelée avec un objet de la classe ou avec le nom de la classe. Dans ce dernier cas, c'est donc le nom complet de la fonction qui est utilisé :

```
Point::setX(70) ;
```

On les utilise le plus souvent pour manipuler les données statique de la classe. De manière générale, on peut définir ce type de méthode chaque fois que l'on souhaite mettre en œuvre un traitement qui n'est pas lié aux objets de la classe.

2.Les fonctions et les classes amies

Le mot clé *friend* est utilisé pour permettre à des fonctions ou à des classes d'accéder aux membres privés et protégés d'une autre classe. Cette technique a donc pour objectif de lever l'interdiction d'accès liée aux étiquettes *private* et *protected*.

Les fonctions et les classes amies sont spécifiées dans la classe pour laquelle vous souhaitez accéder aux membres, comme par exemple :

```
class Point
{
	private :
		int Age ;
```

```
            int Longevite ;

      public :
            Point(Age) : Age(a) {Longevite = 100 ;} ;
            friend void afficheLongevite() ;
            friend class AffichePoint ;
}

void afficheLongevite()
{
      Point Point ;
      cout << Point.Longevite ;
}

class AffichePoint
{
      public :
            Point * Point ;
            AffichePoint(Point * a) : Point(a) {} ;
            void Afficher() ;
}

void AffichePoint::Afficher()
{
      cout << " Age : " << Point->Age ;
      cout << " Longévité : " << Point->Longevite ;
}
```

Le mot clé *friend* permet d'obtenir les conséquences suivantes :

- Une fonction déclarée en tant qu'amie peut accéder à tous les membres d'une classe exactement de la même manière que si elle appartenait à cette classe. Cette relation privilégiée se limite à l'accès aux données et aux fonctions membres, et ne fait pas de cette fonction amie une fonction membre de la classe. Par conséquent, les fonctions amies doivent créer un objet de la classe pour accéder aux membres protégés. Le mot clé *friend* s'applique à la fois aux fonctions classiques et aux fonctions membres des classes.
- Pour une classe amie, toutes les fonctions membres de cette classe seront considérées comme des fonctions amies.

3.La création et la destruction des objets

Une classe est un type de données qui permet de créer des objets. On distingue deux solutions pour créer des objets.

3.a Dans le cas statique

La création statique des objets se fait selon la syntaxe suivante :

```
Point P1 ;
```

La variable *P1* correspond à un objet de type *Point* que l'on peut utiliser pour accéder aux données et fonction membres.

```
P1.x = 10 ;
P1.Affiche() ;
```

Les objets créés de façon statique sont automatiquement détruits dès que le programme quitte la portée où ils ont été créés, en général à la sortie du bloc. Le programmeur n'a donc pas besoin de les détruire.

3.b Dans le cas dynamique

Pour la création dynamique d'un objet, il convient de respecter les étapes suivantes :

- définir un pointeur vers la classe ;
- utiliser l'opérateur *new* ;
- détruire l'objet après son utilisation.

```
Point * pP1 ;
PP1 = new Point ;
```

L'opérateur *new* crée un objet de type *Point* et renvoie l'adresse de l'objet créé, ce qui permet d'initialiser le pointeur *pP1*. Ce pointeur peut ensuite être utilisé pour accéder aux données et fonctions membres de la classe.

Dans le cas de la création dynamique, il faut obligatoirement détruire les objets après les avoir utilisés. Pour cela, C++ fournit l'opérateur *delete* qui permet de détruire l'objet référencé par un pointeur.

```
delete pP1 ;
```

Attention ! *delete* génère une erreur si le pointeur est *NULL*. Il convient donc de vérifier ce point. Le code suivant surmonte le problème.

```
if (pP1)
{
      delete pP1 ;
      pP1 = NULL ;
}
```

Remarque : La valeur *NULL* est compatible avec le booléen faux.

La variable *pP1* correspond à un pointeur sur objet de type *Point* que l'on peut utiliser pour accéder aux données et fonction membres de l'objet.

```
pP1->x = 10 ;
pP1->Affiche() ;
```

3.c La notion d'objet courant ou « this »

Le mot clé *this* correspond à une variable système définie par le C++ désignant l'objet courant, c'est-à-dire l'objet pour lequel vous avez appelé une fonction membre. Au niveau technique, la variable *this* est passée en tant qu'argument caché à toute fonction membre. C'est grâce à cette variable que l'on peut désigner directement les membres d' une classe au sein d'une fonction membre. Cependant, pour éviter tout risque de collision avec une variable locale, on peut utiliser *this* explicitement ; voir l'exemple suivant.

```
void InitAge(int Age)
{
      this->Age = Age ;
}
```

4. Les constructeurs

Lors de la création d'un objet en statique ou en dynamique, le C++ alloue la zone mémoire nécessaire à l'objet, puis appelle une fonction membre spéciale de la classe nommée *constructeur*. L'appel au constructeur d'une classe fait partie intégrante du cycle de vie d'un objet dans la mesure où le constructeur appelé automatiquement à la création d'un objet. Il s'agit donc de la première méthode exécutée pour un objet et elle constitue de ce fait l'endroit idéal pour initialiser les objets d'une classe.

4.a Constructeur par défaut (ou sans argument)

C++ intègre un constructeur par défaut à toute classe qui n'en possède pas. Ce constructeur est une fonction membre sans argument qui est utilisée par défaut à chaque fois que l'on crée un objet en statique ou en dynamique et que la classe ne possède pas d'autre.

Le constructeur par défaut existe uniquement si la classe n'en possède pas déjà un.

4.b Définition d'un constructeur

Un constructeur est une fonction membre qui doit respecter les règles suivantes :

- il possède le même nom que la classe ;
- il ne possède pas de valeur de retour (même pas *void*).

```
#include <iostream.h>
#include <string.h>

class Point
{
      public :
            int x ;
            int y ;

            Point(int x, int y) ;
} ;

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

4.c Appel du constructeur

Lorsque l'on ne veut pas utiliser le constructeur par défaut, il convient de faire appel à un constructeur. Les constructeurs ne sont jamais appelés directement comme les autres fonctions membres des classes. Le constructeur est appelé au moment de la création d'un objet en utilisant une syntaxe très particulière.

L'appel du constructeur diffère selon que la création de l'objet est statique ou dynamique.

Cas statique

```
Point P1(10,20) ;
```

Dans le cas où le constructeur ne possède qu'un seul argument, on peut aussi utiliser la syntaxe raccourcie :

```
Point P1 = 10 ;
```

Dans le cas où le constructeur ne possède pas d'argument, on écrit directement :

```
Point P1 ;
```

Cas dynamique

```
Point * pP1 = new Point(10,20) ;
```

4.d La surcharge des constructeurs

La surcharge des fonctions est très souvent appliquée aux constructeurs des classes. On peut définir plusieurs constructeurs pour répondre à plusieurs contextes d'initialisation des objets.

```
#include <iostream.h>
#include <string.h>

class Point
{
      public :
            int x ;
            int y ;

            Point(int a, int b) ;
            Point() ;

} ;

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

Point::Point()
{
      x = 10 ;
      y = 20 ;
}
```

4.e Les valeurs par défaut avec les constructeurs

Les constructeurs utilisent fréquemment des arguments avec des valeurs par défaut. Il n'est donc pas nécessaire de passer tous les arguments au constructeur de l'objet que vous créez. Voir l'exemple suivant.

```
#include <iostream.h>
#include <string.h>

class Point
{
      public :
            int x ;
            int y ;

            Point(int a = 10, int b = 20) ;

} ;

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

Ce qui donne les appels corrects suivants :

```
Point P1 ;
```

```
Point P1(12) ;
Point P1(12,24) ;
```

4.f Les listes d'initialisation

Jusqu'à présent, on initialisait les données membres d'une classe dans le corps du constructeur ou en utilisant des valeurs par défaut. C++ propose une autre syntaxe pour initialiser les données membres au niveau du constructeur, *les listes d'initialisation.*

L'opérateur : introduit la liste d'initialisation, que l'on aligne avec l'ORP :: de manière conventionnelle. Pour chaque affectation le nom de la donnée membre est suivi de sa valeur d'initialisation entre parenthèses. Toutes ces expressions sont séparées par des virgules.

Reprenons l'exemple précédent avec l'utilisation d'une liste d'initialisation.

```
#include <iostream.h>
#include <string.h>

class Point
{
      public :
            int x ;
            int y ;

            Point(int a, int b) ;

} ;

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

L'argument d'un constructeur utilisé pour l'affectation et la donnée membre doivent être du même type. Par ailleurs, on ne peut pas utiliser cette technique pour initialiser des données membres de type tableau. Cela signifie en particulier que l'on ne peut pas utiliser cette technique pour initialiser une chaîne de caractères représentée par un tableau de caractères.

Cette technique est la seule solution permettant d'initialiser les données membres de type référence ! Une référence doit obligatoirement être initialisée à sa création.

4.g Constructeur de copie (passage par valeur et retour de fonction)

Le constructeur de copie a deux utilisations principales standards en C++ :

- le passage de paramètre par référence ;
- le retour de fonction.

Considérons l'exemple suivant qui surcharge *le constructeur de copie* :

```
class Point
{
      public :
            Point() ;
            Point(Point &) ; // constructeur de copie
            ~Point() ;
}

Point::Point()
{
      cout << "Constructeur de Point"
```

```
}

Point::Point(Point &)
{
      cout << "Constructeur de copie de Point"
}

Point::~Point()
{
      cout << "Destructeur de Point"
}
```

Le programme suivant va faire appel automatiquement au constructeur de copie d'une part pour le passage d'un argument par valeur et d'autre part pour le retour de fonction :

```
Point & PassageParValeur(Point Point)
{
      return Point ;
}

int main()
{
      Point P1 ;
      PassageParValeur(P1) ;
}
```

En conclusion, ce programme affiche :

```
Constructeur de Point (P1)
Constructeur de copie de Point (passage par valeur)
Constructeur de copie de Point (retour de fonction)
Destructeur de Point (destruction de la copie)
Destructeur de Point (destruction du retour)
Destructeur de Point (destruction de chien)
```

5. Le destructeur

Le destructeur est une fonction membre appelée automatiquement juste avant la destruction d'un objet, avant la libération de la mémoire attribuée à l'objet. Le destructeur est donc la dernière méthode exécutée pour un objet donné.

Le destructeur est une méthode :

- qui utilise le nom de la classe précédé du symbole tilde (~) ;
- qui ne renvoie pas de valeur (même pas *void*) ;
- qui ne possède pas d'argument.

Un destructeur ne possède pas d'argument, il ne peut donc pas être surchargé. Cela signifie que contrairement au constructeur, il ne peut y avoir qu'un seul destructeur par classe.

De la même manière que pour le constructeur, le C++ intègre *un destructeur par défaut* à toute classe qui n'en possède pas.

Reprenons l'exemple précédent avec la définition d'un destructeur.

```
#include <iostream.h>
#include <string.h>

class Point
{
      public :
```

```
            Point() ;
            ~Point() ;

} ;

Point::Point()
{
      cout << "Construction de Point : " << this
}

~Point::Point()
{
      cout << "Destruction de Point : " << this
}
```

Le destructeur n'est jamais appelé directement ; cette fonction est appelé automatiquement selon le mode utilisé, statique ou dynamique, pour créer l'objet.

- Pour un objet créé en statique, le destructeur est appelé dès que le programme quitte la portée où cet objet à été créé.
- Pour un objet créé en dynamique, le destructeur est appelé en réponse de l'utilisation de l'opérateur *delete* sur cet objet. Si, dans ce cas, on oublie d'utiliser l'opérateur *delete*, la mémoire utilisée par cet objet n'est pas restituée et le destructeur n'est jamais appelé.

6. La protection des données et des fonctions membres

La notion d'encapsulation est également associée à un système de protection qui permet de contrôler la visibilité d'une donnée ou fonction membre.

6.a public

Toutes les données ou fonctions membres d'une classe définies avec le mot clé *public* sont utilisables par toutes les fonctions. Il s'agit du niveau le plus bas de protection. Ce type de protection est employé pour indiquer que l'on peut employer sans contrainte les variables et fonctions d'une classe.

6.b private

Tous les membres d'une classe définis avec le mot clé *private* sont utilisables uniquement par les fonctions membres de cette classe. Cette étiquette constitue le niveau le plus fort de protection.

6.c protected

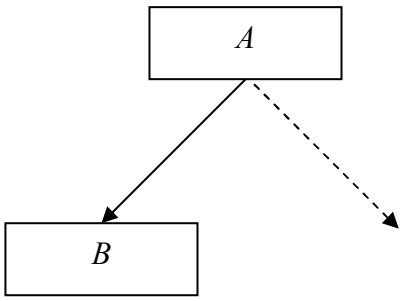
Tous les membres d'une classe définis avec le mot clé *protected* sont utilisables uniquement par les fonctionnements membres de la classe et par les fonctions membres des classes dérivées. Cette technique de protection est étroitement liée à la notion d'héritage qui sera développé au chapitre suivant.

7.L'héritage simple

7.a Le concept d'héritage

L'héritage est une technique rationnelle permettant de réutiliser et de spécialiser les classes existantes. L'héritage offre la possibilité de créer une classe à partir d'une autre. Cela signifie que la nouvelle classe bénéficie des attributs (données membres) et des comportements (fonctions membres) de la classe dont elle dérive.

Le concept d'héritage est aussi connu sous le nom *de dérivation des classes*. La classe dont dérive une classe se nomme la classe de base, la superclasse ou la classe mère… La classe dérivée se nomme la classe fille ou la sous-classe.



Pour bien comprendre l'intérêt de l'héritage, il est nécessaire de bien assimiler ce qu'implique cette relation par rapport à la création des objets. En effet, chaque fois que l'on crée un objet de la classe dérivée, le C++ crée un objet de la classe de base. Ainsi si la classe B dérive de la classe A, la création d'un objet de type B entraînera implicitement la création d'un objet de type A. Un objet de type B est en réalité un objet de type A avec des éléments supplémentaires.

Les données membres des classes de bases sont allouées au moment de la création de la même manière que pour les données membres de la classe dérivée. Cette opération est réalisée indépendamment des directives de protection. En d'autres termes, ce n'est pas parce qu'une donnée membre de A est définie *private* qu'elle n'est pas allouée.

Cette création automatique d'un objet de la classe de base, chaque fois que l'on déclare un objet de la dérivée, se propage tout au long de la hiérarchie. C'est-à-dire que si la classe de base dérive également d'une autre classe, le processus de création automatique d'objets continue.

C++ fournit une syntaxe particulière pour spécifier qu'une classe hérite d'une autre classe :

```
// classe de base
class A
{
} ;

// classe dérivée de A
class B : public A
{
} ;
```

L'opérateur : situé juste après le nom de la classe indique que celle-ci dérive d'une autre classe. Le modificateur qui suit peut correspondre à une des trois valeurs suivantes : *public, private, protected*. Ces étiquettes permettent de spécifier l'héritage. On abordera ce point dans une partie suivante. Notons simplement que l'accès *public* est de loin le plus employé.

7.b Héritage et protection des membres

Du point de vue de l'héritage, une classe dérivée peut accéder aux membres publiques et protégées de la classe de base. Les membres privés ne sont manipulables que par les fonctions membres de la classe qui les a définies.

Quelles que soient les étiquettes de protection, une fonction membre d'une classe peut accéder à toutes les données membres de cette classe.

Un constructeur comme toute fonction membre, doit être intégré à une étiquette de protection. Dans la plupart des cas, les constructeurs sont déclarés dans la section *public*, ce qui permet à quiconque de créer les objets de cette classe en utilisant les constructeurs.

7.c Spécification de l'héritage

Lorsqu'on dérive une classe, il faut indiquer entre l'opérateur : et le nom de la classe de base, un des mots clés *public, private* ou *protected*. Dans la pratique, on utilise presque exclusivement la spécification *public*.

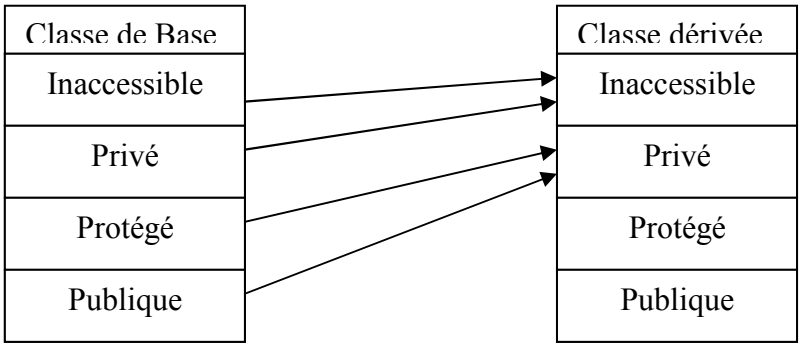
C++ impose deux règles d'utilisation des étiquettes dans le cas de l'héritage :

- Si l'on omet d'employer un de ces trois mots clés, le compilateur utilisera comme valeur par défaut l'étiquette *private*.
- Le choix de l'étiquette ne change rien pour la classe elle-même ; la spécification de l'héritage n'intervient que pour les classes dérivées.

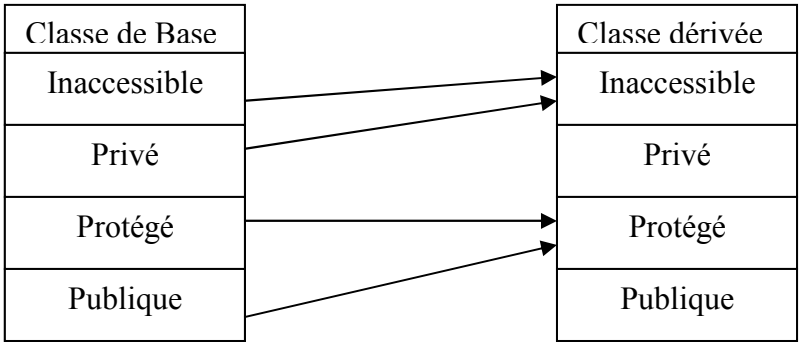
Le choix de spécification d'héritage permet de spécifier comment se propagera l'héritage (en terme d'accès) dans une hiérarchie de classes. Cela signifie que l'on peut préciser pour une classe , la politique d'accès de ses propres classes dérivées.

Les 3 types d'héritage :

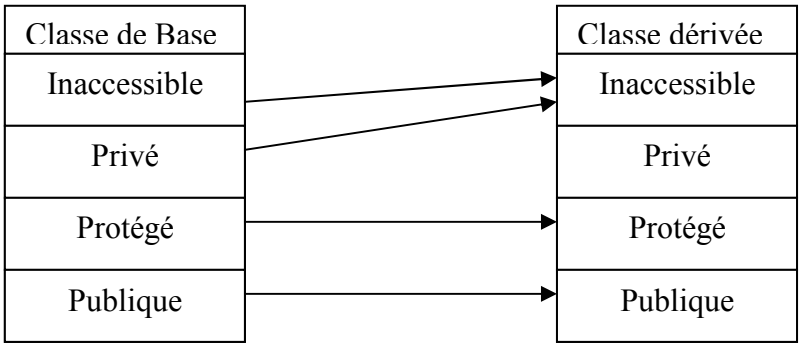
a) Héritage privé : (Par Défaut)



b) Héritage protégé



c) Héritage Public



De façon générale, on ne peut faire qu'augmenter le niveau de sécurité, jamais le diminuer ! Dans la pratique, les deux spécifications d'héritage *protected* et *private* sont rarement utilisées puisqu'elles ont tendance à couper le relation d'héritage.

7.d Héritage simple, constructeurs et destructeurs

Le constructeur d'une classe dérivée doit obligatoirement appeler un constructeur de sa classe de base. Lorsque l'on utilise des constructeurs par défaut comme dans l'exemple précédent, le compilateur gère automatiquement l'enchaînement de l'appel des constructeurs, ce qui simplifie la tâche. En revanche, et dans le cas général où les constructeurs possèdent des arguments, la mise en place d'une relation d'héritage nécessite de fournir des arguments aux constructeurs de la classe de base par un appel explicite à ce constructeur dans la classe dérivée.

Par ailleurs, la destruction des objets liés par une relation d'héritage passe par un enchaînement de l'appel aux classes impliquées. Ce processus est simple à mettre en œuvre dans la mesure où une classe ne peut disposer que d'un seul destructeur, ce que le compilateur gère lui-même. Il n'est donc pas besoin d'intervenir dans ce processus.

Chaque fois que l'on crée un objet lié par une relation d'héritage, il y a appel au constructeur de la classe de base, et ainsi de suite… En respectant ce processus, les objets de bases sont créés avant les objets dérivés. Pour les destructeurs, c'est l'ordre inverse qui est employé, ce qui signifie que le destructeur d'une classe dérivée est appelé avant celui de sa classe de base. L'exemple suivant illustre ce dernier propos et utilise des constructeurs sans arguments, dont on rappelle qu'ils sont les constructeurs par défaut !

```
class A
{
      public :
            A() ;
            ~A() ;
} ;

class B : public A
      public :
            B() ;
            ~B() ;
} ;

A::A()
{
      cout << "Constructeur A \n"
}

A::~A()
{
      cout << "Destructeur A \n"
}

B::B()
{
      cout << "Constructeur B \n"
}

B::~B()
{
      cout << "Destructeur B \n"
}
```

Le programme suivant :

```
int main()
{
```

```
        B * pB ;
        pB = new B ;
        delete pB ;
}
```

affiche le résultat suivant :

```
Constructeur A
Constructeur B
Destructeur B
Destructeur A
```

Dans l'exemple précédent, l'enchaînement des constructeurs se faisait automatiquement, car on utiliser les constructeurs sans arguments. Si l'on souhaite dériver une classe qui possède un constructeur avec arguments, il convient de respecter un certain nombre de règles.

On considère la classe *Point* suivante :

```
class Point
{
        protected :
                int x,y ;
        public :
                Point(int a,int b) ;
} ;

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

Cette classe ne comporte pas de constructeur sans argument, et si l'on ne définit pas de constructeur dans la classe dérivée, le compilateur C++ générera l'erreur suivante : *Cannot find default constructor to initialize base class 'Point'...* En vérité, l'appel automatique des constructeurs ne peut fonctionner que si la classe de base et la classe dérivée possèdent un constructeur sans argument, sinon il convient de réaliser cet appel explicitement. On appelle le constructeur de la classe de base et on lui passe les arguments adéquats, comme dans l'exemple suivant où la classe *Point3D* dérive de la classe *Point*.

```
class Point3D
{       protected :
                int z ;
        public :
                Point3D(int a,int b, int c) ;
} ;

Point3D::Point3D(int a,int b, int c):Point(a,b)
{z= c ;}
```

Il faut remarquer que l'appel au constructeur de la classe de base se situe avant le corps du constructeur lui-même, en respectant la syntaxe des listes d'initialisation. A ce niveau, on comprend mieux pourquoi l'exécution du constructeur de la superclasse est la première opération réalisée par le constructeur de la classe dérivée.

8.L'héritage multiple

8.a Le principe de l'héritage multiple

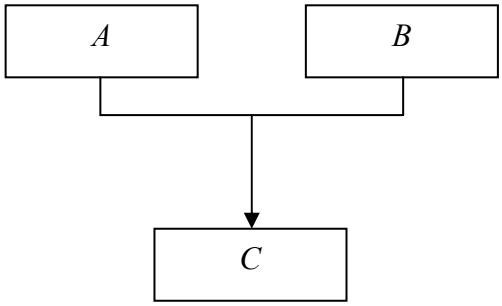
C++ à la différence du *Java,* par exemple, accepte l'utilisation de l'héritage multiple. Cela signifie que l'on peut définir qu'une classe dérive de une ou plusieurs classes. Cette technique offre ainsi la possibilité de factoriser des attributs et des comportements provenant de classes qui ont peu ou pas de points communs.

Pour définir une classe qui dérive de plusieurs classes, il convient de séparer chaque classe de base par une virgule, comme dans l'exemple suivant :

```
class C : public A, public B
{

} ;
```

La classe C dérive à la fois des classes A et B. Cela signifie que chaque fois que l'on crée un objet de type C, le C++ créera un objet de type A et un objet de type B associé à cet objet. Dans ce cas, le constructeur de la classe C doit appeler les deux constructeurs des classes de base A et B.



8.b Héritage multiple, constructeurs et destructeurs

On rappellera seulement que si l'on ne crée aucun constructeur, le C++ crée le constructeur sans argument par défaut et qu'il l'appelle automatiquement si aucun appel n'est explicité dans le constructeur d'une classe dérivée. Attention toutefois au cas où l'on définit un constructeur avec argument dans une classe de base, l'appel implicite au constructeur par défaut dans une classe dérivée ne pourra s'effectuer correctement que si le constructeur sans argument est effectivement définie dans la classe de base, sinon une erreur de compilation surviendra !

Exemple

```
class A
{
      protected :
            int a ;
      public :
            A(int aa) ;
} ;

class B
      protected :
            int b ;
      public :
            B(int bb) ;
} ;

class C : public A, public B
{
      protected :
            int c ;

      public :
            C(int aa, int bb, int cc) ;
} ;

A::A(int aa)
{
```

```
        a = aa ;
        cout << "Constructeur A \n"
}

B::B(int bb)
{
        b = bb ;
        cout << "Constructeur B \n"
}

C::C(int aa, int bb, int cc)
        :A(aa), B(bb)
{
        c = cc ;
        cout << "Constructeur C \n"
}
```

Le programme suivant :

```
int main()
{
        C objetC(1,2,3) ;
        return 0 ;
}
```

affiche le résultat suivant :

```
Constructeur A
Constructeur B
Constructeur C
```

9.Polymorphisme

Avec l'encapsulation et l'héritage, le polymorphisme constitue la troisième et la plus importante caractéristique des langages orientés objet.

9.a Redéfinition des fonctions

Le C++ offre la possibilité de redéfinir une fonction de la classe de base dans la classe dérivée, comme la méthode *Crier* dans l'exemple suivant.

```
class Animal
{
        public :
                void Crier() ;
} ;

class Chien : public Animal
{
        public :
                void Crier() ;
} ;

void Animal::Crier()
{
        cout << " ? " ;
}

void Chien::Crier()
{
```

```
        cout << " Ouah-ouah ! " ;
}
```

Considérons maintenant le programme suivant :

```
int main()
{
        Animal * pAnimal = new Chien ;
        pAnimal->Crier() ;
        delete pAnimal ;

        Chien * pChien = new Chien ;
        pChien->Crier() ;
        delete pChien ;
}
```

Le programme affiche :

```
?
Ouah-ouah !
```

Attention, un pointeur de base peut contenir des objets dérivés mais jamais l'inverse ! Ici, *pAnimal* est un pointeur de base sur *Animal* qui contient un objet de type *Chien*.

Ce petit programme met en évidence que la méthode *Crier* appelée est toujours celle correspondant au type du pointeur. Lorsqu'un objet de type *Chien* est désigné par un pointeur de type *Chien*, *pChien*, la méthode *Crier* appelée est celle de la classe *Chien* ; tandis que si le même objet de type *Chien* est désigné par un pointeur de type *Animal*, *pAnimal*, la méthode *Crier* appelée est celle de la classe *Animal*.

9.b Fonctions polymorphes ou virtuelles

On aurait espéré dans l'exemple précédent que la méthode *Crier* appelée soit, non celle désignée par le type du pointeur, mais celle désigné par le type de l'objet ; c'est-à-dire qu'un objet de type Chien réponde par « *Ouah-ouah !* » même si on utilise pour le désigner un pointeur de base !

Cela est possible à condition d'utiliser des fonctions dites *polymorphes* ou *virtuelles*. Une fonction virtuelle est une fonction qui est appelée en fonction du type d'objet et non pas en fonction du type de pointeur utilisé. Pour déclarer qu'une fonction membre est polymorphe, il suffit de faire précéder son prototype par le mot clé *virtual*. Il n'est pas nécessaire de faire apparaître ce mot clé devant le corps de la fonction.

Nous reprenons l'exemple précédent en y adjoignant la classe *Chat*.

```
class Animal
{
        public :
                virtual void Crier() ;
} ;
class Chien : public Animal
{
        public :
                virtual void Crier() ;
} ;

class Chat : public Animal
{
        public :
                virtual void Crier() ;
} ;


void Animal::Crier()
```

```
{
	cout << " ? " ;
}

void Chien::Crier()
{
	cout << " Ouah-ouah ! " ;
}

void Chat::Crier()
{
	cout << " Miaou-miaou ! " ;
}
```

Le programme suivant utilise un pointeur de base sur *Animal* pour désigner des objets de type *Chien* ou *Chat* et

```
int main()
{
	Animal * pAnimal ;

	pAnimal = new Chien ;
	pAnimal->Crier ;
	delete pAnimal ;

	pAnimal = new Chat ;
	pAnimal->Crier() ;
	delete pAnimal ;
}
```

affiche le cri attendu :

```
Ouah-ouah !
Miaou-miaou !
```

Techniquement, seule la fonction *Crier* de la classe *Animal* doit être déclarée virtuelle. Cependant, il est conseillé d'indiquer le mot clé *virtual* pour toute les fonctions concernées par le polymorphisme, et cela quelque soit leur niveau dans la hiérarchie des classes. De la sorte vous n'aurez pas à étudier toute l'arborescence des classes pour identifier les fonctions virtuelles.

Lorsque l'on appelle une fonction virtuelle pour un objet, le C++ cherche cette méthode dans la classe correspondante. Si cette fonction n'existe pas dans la classe concernée, le C++ remonte la hiérarchie des classes jusqu'à ce qu'il trouve la fonction appelée. La résolutions des appels des fonctions virtuelles à lieu à l'exécution (*ligature dynamique*) et non à la compilation comme pour les autres fonctions.

9.c Fonctions virtuelles pures et classes abstraites

La fonction virtuelle pure est une fonction virtuelle pour lequel le prototype est suivi de l'expression *=0*. L'instruction suivante montre la définition d'une fonction virtuelle pure :

```
virtual void Crier()=0 ;
```

Définir une fonction virtuelle pure (équivalent des méthodes abstraites en Java) dans une classe a deux incidences :

- Une classe qui contient la définition d'une fonction virtuelle pure devient *une classe abstraite*.
- La redéfinition des fonctions virtuelles pures est obligatoire dans toutes les classes dérivées. Dans le cas contraire, les classes deviennent également abstraites.

L'avantage de ces fonctions membres consiste donc à obliger les classes dérivées à les redéfinir sous peine de devenir elles-mêmes abstraites.

Reprenons l'exemple de la classe *Point*.

```
class Point
{
	public :
		void Chanter() ;
		virtual void Crier()=0 ;
} ;

class Chien : public Point
{
	public :
		virtual void Crier() ;
} ;

class Chat : public Point
{
	public :
		virtual void Crier() ;
} ;


void Chien::Crier()
{
	cout << " Ouah-ouah ! " ;
}

void Chat::Crier()
{
	cout << " Miaou-miaou ! " ;
}
```

Les fonctions virtuelles pures sont les seules fonctions qui peuvent ne pas avoir de corps de fonction. Cela n'est possible qu'à la condition expresse que la méthode ne soit pas appelée directement !

Une classe devient abstraite si elle possède au moins une fonction virtuelle pure. Dans notre exemple la classe *Point* est abstraite ; et de ce fait il devient impossible de créer un objet directement à partir de cette classe !

10.La surcharge des opérateurs

Les opérateurs du langage C++ (+, -, =, ==, …) servant à manipuler les types primaires ne peuvent pas être utiliser nativement avec les classes que l'on définit.

Pour définir un opérateur « x » on utilise le mot clef **operator x {…}**, il peut y avoir un retour et des arguments.

Exemple d'utilisation : A x B <-> A.operator x(B)

Pour les opérateurs binaire, il y a un seul argument (voir exemple ci dessous).

Il est possible de donner une signification à ces différents opérateurs pour les classes en utilisant *la surcharge d'opérateur*, comme dans l'exemple suivant où l'on surcharge l'addition +.

```
int operator + (Nombre a, Nombre b)
{
	// Corps
}
```

Les paramètres (*a* et *b*) correspondent au différents opérandes associés à l'opérateur + et devront correspondre au type de la classe pour laquelle on souhaite redéfinir cet opérateur, c'est-à-dire *Nombre* dans le cas présent.

```
class Nombre
{
	private :
		int Valeur ;
```

```
        public :
                Nombre(int v) {Valeur = v} ;
                int operator + (Nombre b) ;
                Nombre & operator ++ () ; // opérateur préfixe

} ;

int Nombre::operator + (Nombre b)
{
        return (this->valeur + b.valeur) ;
}

Nombre & Nombre::operator ++ ()
{
        ++Valeur ;
        return *this ;
}
```

Par ailleurs, il faut noter que contrairement à ce que l'on a indiqué plus haut, l'opérateur surchargé + ne semble posséder qu'un seul paramètre ! En fait, il en est rien ! Cette surcharge d'opérateur est définie comme membre de la classe *Nombre*, le premier argument sera donc implicitement l'objet courant pointé par *this*.

On donne ci-dessous un exemple de l'utilisation de la classe précédente qui devra afficher l'entier 301 :

```
Nombre A(100) ;
Nombre B(200) ;
++A ;
cout << (A+B) ;
```

Il est également possible de surcharger les opérateurs suffixes et les opérateurs de conversions (par exemple, *unsigned short*) qui ne retourne aucune valeur !

L'opérateur d'affectation = est redéfini automatiquement pour toutes les classes existante et créé plus tard. Le compilateur le fait automatiquement, mais il est possible de le redéfinir soit même quant il y a des pointeurs dans notre classe.

L'opérateur [ ] doit renvoyer une valeur par référence donc on utilise le &, l'opérateur crochet posséde un seul argument !

Exemple :

Class chaine

{char *T ;

public :

chaine (const char *)

char &operator [](int i)

{ return T[i] ;}

}

main ()

{chaine ch(« Bonjour ») ;

ch[1]= »a » ;

```
}
```

L'opérateur << ou >> on les définies comme fonctions amies.
Exemple de redéfinition :

```
Class Point
{ int x,y ;
 public :

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

friend ostream & operator <<(ostream & sortie, Point & p) ;
}

ostream & operator << (ostream & sortie, Point & p)
{ sortie << x ; sortie <<y ;
 return sortie ;}
```

<< doit etre défini en tant qu'amie car sinon ostream serait le second operateur.

Pour l'opérateur >> utiliser istream à la place de ostream
Exemple :

```
Class Point
{ int x,y ;
 public :

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

friend istream & operator >>(istream entree, Point  p) ;
}

istream & operator >> (istream  sortie, Point  p)
{ entree>> x ; entree>>y ;
 return entree ;}

main()
{ Point P ;
cin>>P ;
cout<<P ;}
```

La surcharge des opérateurs a tendance à réduire la lisibilité des programmes dans la mesure ou derrière une banale addition peut se cacher l'exécution d'un grand nombre de lignes de code.

Par ailleurs, la surcharge des opérateurs ne permet pas de modifier la signification des opérateurs dans le cas des types primaires du C++ (*short*, *int*, *long*, …).

La surcharge d'un opérateur s'applique à un type particulier de donnée. Par conséquent, il faut définir cette surcharge pour toutes les classes concernées (notamment lors de l'héritage), mais aussi tenir compte du type d'argument (pointeur, référence ou valeur). Pour ces raisons, il est souvent nécessaire de définir deux surcharges d'opérateurs (par pointeur et par référence).

11.Les templates

De la même manière que l'héritage ou le polymorphisme, les *templates* appelées aussi *modèles* représente une technique favorisant la réutilisation et la spécialisation des outils C++ que l'on définit. Les modèles permettent de définir des traitements identiques et applicables à plusieurs type de données. Les templates s'appliquent aux fonctions, aux classes et aux fonctions membres des classes modèles.

Ils utilisent un type paramétré (ou fictif) qui représente le type de donnée qui sera choisi (ou instancié) à l'utilisation du modèle.

11.a Le rôle des templates

Considérons l'extrait de code suivant :

```
void Affiche(short v)
{
      cout << v ;
}

void Affiche(float v)
{
      cout << v ;
}

void Affiche(double v)
{
      cout << v ;
}
```

Les trois fonctions surchargées *Affiche* réalisent le même traitement pour des types de données différents. Une fonction modèle permet dans ce cas de définir une seule et unique fonction s'appliquant à tous les types de données primaires, voire à tous les types sous certaines contraintes !

Un template peut utiliser un ou plusieurs types paramétrés qui seront instancié à l'utilisation du modèle. Dans ce contexte, cela signifie que vous définissez tout le code du template pour un type de donnée fictif qui sera instancié par un type réel au moment de l'utilisation du modèle.

En C++, les modèles concernent les éléments suivants :

- les fonctions classiques ;
- les classes avec leurs données et fonctions membres.

11.b Les fonctions templates

Une fonction template permet de définir une famille de fonctions qui partage le même traitement pour des types de données différents.

Les fonctions surchargées de l'exemple précédent pourrait avantageusement être remplacé par la fonction modèle suivante :

```
template <class T>
void affiche(T v)
```

```
{
	cout << v ;
}
```

La lettre T, utilisée pour désigner le type paramétré d'un modèle, constitue une convention d'écriture. Dans la pratique, on peut choisir n'importe quel nom pour désigner ce type fictif.

Il n'existe pas de syntaxe spécifique pour appeler une fonction template. Comme pour toute fonction, il faut simplement passer le nombre d'arguments adéquate. De plus, le type de ces arguments doit supporter l'ensemble des traitements mis en œuvre par la fonction.

Considérons plusieurs appels possibles :

```
short s = 1 ;
Affiche(s) ;
float f = 2.4 ;
Affiche(f) ;
Affiche('a') ;
```

Une fonction template peut utiliser plusieurs types paramétrés comme le montre la fonction suivante :

```
template <class T, class U>
void affiche(T v, U w)
{
	cout << v ;
	cout << w ;
}
```

11.c Les classes templates

Une classe modèle fournit la même facilité que les fonctions templates en donnant la possibilité de paramétrer un type de donnée aussi bien pour ses données que pour ses fonctions membres.

Lors de la définition d'un classe template, il faut tout d'abord définir la classe template, puis définir les fonctions membres comme des fonctions templates avec une syntaxe identique. Même si une classe modèle contient des fonctions membres qui n'utilisent pas les types fictifs, ces dernières doivent respecter la contrainte liée au nom complet (`NomClasse<NomDesArguments>::Fonction`) de la fonction membre.

Voyons un exemple pour éclaircir ce discours :

```
template <class T>
class Tableau
{
	private :
		T Valeurs[10] ;
		int indice ;

	public :
		Tableau() ;
		void Ajouter(T v) ;
} ;

template <class T>
T Tableau<T>::Tableau()
{
	indice = 0 ;
}

template <class T>
T Tableau<T>::Ajouter(T v)
{
	Valeurs[indice] = v ;
	indice++ ;
}
```

Une fois définie, une classe modèle peut être employée pour créer des objets statiques ou dynamiques. Cependant, contrairement aux fonctions modèles, il faut explicitement indiquer la valeur du ou des types paramétrés au moment de la création des objets de cette classe.

```
Tableau<int> TabEntiers ;

Tableau<char> * pTabCaracteres ;
pTabCaracteres = new Tableau<char> ;
...
delete pTabCaracteres ;
```

Pour simplifier la notation employée pour créer ces objets, on peut avantageusement utiliser l'opérateur *typedef* :

```
typedef Tableau<int> TabInt ;
TabInt TabEntiers ;
```

En plus des types paramétrés, on peut utiliser la liste d'arguments d'un modèle pour définir des paramètres. La valeur de ces arguments devra être passée à l'instance du modèle, exactement de la même manière que pour les type paramétrés. De cette façon, la classe modèle dispose d'arguments qui peuvent ensuite être utilisé par les données et fonctions membres de la classe. Par ailleurs, ces variables peuvent avoir des valeurs par défaut comme les arguments des fonctions.

Dans le cas de la classe *Tableau*, un argument supplémentaire permettrait d'indiquer, par exemple, le nombre d'élément souhaité pour le tableau.

```
template <class T, int n = 10>
class Tableau
{
      private :
            T Valeurs[n] ;
            int indice ;

      public :
            Tableau() ;
            void Ajouter(T v) ;
} ;

template <class T, int n>
T Tableau<T>::Tableau()
{
      indice = 0 ;
}

template <class T, int n>
T Tableau<T>::Ajouter(T v)
{
      Valeurs[indice] = v ;
      indice++ ;
}
```

On dispose alors des appels suivants :

```
Tableau<int,20> TabEntiers ;

Tableau<char> * pTabCaracteres ;
pTabCaracteres = new Tableau<char> ;
...
delete pTabCaracteres ;
```

11.d Templates et membres statiques

Dans le cas des membres statiques, et plus particulièrement des données statiques, il faut utiliser le nom complet pour la définition hors classe, dont on rappelle qu'elle est toujours obligatoire dans ce cas.

```
template <class T>
class Liste
{
      public :
            static int n ;
} ;

template <class T>
int  Liste<T>::n = 10 ;
```

Une donnée statique existe pour chaque instance du modèle. Cela signifie, par exemple, que tous les objets de type *Liste<int>* partagent la même variable statique et que tous les objets de type *Liste<char>* partagent la leur.

12.Les exceptions

Outre les erreurs de compilations liées à une mauvaise programmation, de nombreux problèmes peuvent encore survenir lors de l'exécution d'un programme ; on parle *d'erreur d'exécution* (l'insuffisance de mémoire, la perte d'un fichier, la saisie non valide d'une entrée, …). Le rôle des programmeurs consiste à prévoir ces erreurs, à en informer l'utilisateur et éventuellement à mettre en œuvre des solutions de reprises et de correction de ces erreurs d'exécution.

12.a Mise en œuvre des exceptions

Les exceptions constituent une solution fiable et standardisée de gestion d'erreurs d'exécution. Pour mettre en œuvre la gestion des exceptions, vous devez respecter le canevas suivant :

- définir une classe d'exception ;
- lancer l'exception ;
- intercepter l'exception.

Définition d'une classe d'exception

Une classe d'exception correspond à une classe C++ qui peut fournir des informations sur une erreur. Il n'existe aucune contrainte particulière pour définir une telle classes.

La classe représentée ci-dessous représente une classe d'exception valide :

```
class Erreur
{
} ;
```

On peut avantageusement ajouter des données et fonctions membres pour permettre de définir une politique de gestion d'erreurs très évoluée.

Lancement d'une exception

Toute fonction souhaitant lancer une exception doit utiliser l'opérateur *throw* du langage C++. Cette opérateur doit être suivi par un objet créé à partir d'une classe d'exception. Cet opérateur permet de quitter la fonction qui l'utilise et d'informer la fonction appelante qu'une exception a été générée.

L'exemple suivant illustre l'utilisation de l'opérateur *throw* dans le cadre de la fonction *Positive* :

```
void Positive(int v)
{
      if (v<0)
            throw Erreur() ;
```

```
        cout << "Valeur positive" << endl ;
    }
```

La syntaxe utilisée permet de créer l'objet d'exception de façon anonyme, c'est-à-dire sans lui donner de nom. Cette syntaxe n'est intéressante que si l'on a pas besoin d'appeler des fonctions membres de la classe d'exception, sinon on préfèrera la syntaxe suivante :

```
    if (v<0)
        {
            Erreur e ;
            throw e;
        }
```

Toute fonction susceptible d'envoyer une ou plusieurs exceptions peut l'indiquer dans sa déclaration juste après sa liste d'arguments ; on parle alors de *spécification d'exception*.

```
    void Positive(int v) throw (Erreur, ...)
    {
        // Corps
    }
```

Cette syntaxe (bien qu'optionnelle pour les fonctions) permet d'une part au compilateur de contrôler la liste des exceptions envoyées par une fonction et d'autre part de permettre aux programmeurs d'identifier très rapidement la liste des exceptions pouvant être lancé par une fonction.

Interception d'une exception

En C++, il n'est pas obligatoire d'intercepter les exceptions. Cependant, le lancement d'une exception non interceptée provoque la fin de l'exécution du programme. Cela signifie que d'une manière générale, les programmes ont besoin d'intercepter et de traiter les exceptions qu'ils reçoivent.

Pour intercepter une exception, le C++ fournit une syntaxe basée sur l'utilisation du bloc *try* et de un ou plusieurs blocs *catch*. Il convient de respecter la syntaxe suivante :

```
    try
    {
        // Appels de fonctions pouvant générer des exceptions.
    }
    catch (ClasseException e)
    {
        // Ce bloc s'exécute pour une exception de type
ClasseException.
    }
    catch (...)
    {
        // Ce bloc s'exécute pour toutes autres exceptions !
    }
```

Après un bloc *try*, il faut au moins spécifier un bloc *catch*. La syntaxe particulière *catch(...)* permet d'intercepter toutes les exceptions renvoyées par les fonctions qui sont appelées dans le bloc *try*.

Lors de l'interception d'une classe d'exception précise, le bloc *catch* récupère directement une copie de l'objet créé et renvoyé par l'opérateur *throw*.

Si une exception est envoyée par une de ces fonctions appelées dans le bloc *try*, le mécanisme d'exception entraînera les étapes suivantes :

- Tous les objets créés dans le bloc *try* sont automatiquement détruits.
- Le programme sort du bloc *try* juste après la fonction qui a entraîné l'exception et n'exécute pas les instructions situés après cette fonction.

- Le C++ exécute dans l'ordre soit le bloc *catch* correspondant à l'exception interceptée s'il existe, soit le bloc *catch(...)*. Si aucune des conditions est remplies, cela entraîne la fin d'exécution du programme.
- Si un des blocs *catch* a été utilisé, le programme continue à exécuter les instructions après ce bloc s'il y en a.