
# INF 2005 Programmation orientée objet avec C++

## Texte 4



Version du 12 janvier 2015

# 1. Héritage et classe dérivée

L'héritage constitue l'un des aspects importants de la programmation orientée objet. Il permet de spécifier les relations entre les classes. C'est le mécanisme qui facilite la réutilisation du code d'un programme.

La dérivation de classe est la possibilité de définir une nouvelle classe à partir d'une classe existante appelée *classe de base* (ou encore *classe mère*). Cette nouvelle classe s'appelle la *classe dérivée* (ou encore *classe fille*) et ses membres ont accès aux données et aux méthodes de la classe de base. Une classe dérivée hérite de toutes les méthodes et des attributs de la classe de base et, en plus, elle pourra les modifier et définir ses propres méthodes.

Par exemple, imaginons un programme de gestion du personnel. Sans mécanisme d'héritage, nous aurions besoin de plusieurs classes, comme une classe `P_informatique` (personnel informatique) avec ses attributs et ses méthodes et une classe `P_comptable` (personnel de la comptabilité) avec ses méthodes et ces attributs, etc. Nous aurions donc dans ces classes autant d'attributs que de méthodes, ce qui rend le programme volumineux. Ces classes ont sans doute des attributs en commun, comme le nom, le prénom, la date de naissance, etc.

Avec les possibilités qu'offre l'héritage, vous pouvez identifier une classe, par exemple `Personne`, dont héritent les classes `P_informatique`, `P_comptable`. Cette classe sert de classe de base et possédera les attributs communs aux différentes classes dérivées comme le nom, le prénom, la date de naissance, etc.

## 1.1 La déclaration

La déclaration d'héritage en C++ se fait lors de la déclaration des classes dérivées. On fait suivre le nom de la classe fille par celui de la classe mère dont on désire qu'elle hérite. Puis on sépare les noms des deux classes par un deux-points `:` . On fait aussi précéder le nom de la classe de base par son type d'accès qui influe sur les contrôles d'accès des propriétés héritées.

La syntaxe est la suivante :

```
class nom_classe_derive : type_acces  nom_classe_base
```

Voyons un exemple de déclaration.

```
class mere
{
   private :
   int numero ;
    . . .
 };

class fille : public mere
  {
    private :
    float Identifiant_service ;
    . . .
   };
```

La définition des types d'accès restent les mêmes mots clés que ceux pour le contrôle d'accès des attributs et des méthodes des classes, c'est-à-dire `private`, `public`, `protected`. Si rien n'a été spécifié, le type par défaut est `private` pour une classe et `public` pour une structure.

Le tableau suivant illustre l'accessibilité des membres de la classe de base dans la classe dérivée.

**Tableau 1** Accessibilité des membres

| **Classe de base** | | | **Classe dérivée après héritage `public`** | | |
|---|---|---|---|---|---|
| Statut initial | Accès interne | Accès externe | Statut | Accès interne | Accès externe |
| `public` | oui | oui | `public` | oui | oui |
| `protected` | oui | non | `protected` | oui | non |
| `private` | oui | non | `private` | non | non |

| **Classe dérivée après héritage `protected`** | | | **Classe dérivée après héritage `private`** | | |
|---|---|---|---|---|---|
| Statut | Accès interne | Accès externe | Statut | Accès interne | Accès externe |
| `protected` | oui | non | `private` | oui | non |
| `protected` | oui | non | `private` | oui | non |
| `private` | non | non | `private` | non | non |

Mais C++ autorise le changement de statut pour un membre `public` ou `protected` dans la classe dérivée quel que soit le type d'héritage. Il suffit de déclarer le membre dans la classe dérivée avec le mot clé `using`. Ce changement est possible aussi bien pour les attributs que pour les méthodes.

La syntaxe est la suivante :

```
using nom_classe_base :: nom_methode ;
```

## 1.2 Les propriétés de l'héritage

Contrairement à certains langages comme le C# (C sharp de Microsoft), le langage C++ accepte l'héritage multiple, c'est-à-dire qu'une classe peut hériter de plusieurs classes de base. Ce concept de l'héritage multiple permet à une classe d'hériter de membres et de méthodes de plusieurs classes de base. Il suffit de faire suivre le nom de la classe dérivée des noms des différentes classes de base, accompagnés de leurs types d'accès, séparés par des virgules.

La syntaxe de l'héritage multiple est la suivante :

```
class nom_classe_derive : type_acces  nom_classe_base
nom_classe_base2, ...
```

Voici quelques propriétés liées à l'héritage.

- Une classe dérivée peut servir de classe de base pour une autre dérivation.
- Une classe dérivée peut modifier et ajouter des attributs.
- Une classe dérivée hérite des attributs (données et fonctions) de la classe de base sauf les constructeurs, les destructeurs et les opérateurs d'affectation.

## 1.3 Les constructeurs et les destructeurs

En langage C++, l'appel du constructeur d'une classe dérivée n'est réalisé qu'après l'appel du constructeur de la classe de base. La règle s'applique aussi à la classe de base si cette dernière hérite aussi d'une autre classe.

Quant au destructeur, l'appel s'effectue d'abord avec la classe dérivée, puis avec la classe de base. Par défaut, ce sont les constructeurs des classes de base qui sont appelés si aucune spécification n'est donnée lors de la création des classes.

Il est aussi possible lors de la définition d'un constructeur d'une classe dérivée de spécifier le constructeur de la classe de base avec les arguments suivants :

```
class classe_base
 { . . .
   classe_base(int B1) ;
  } ;
```

```
class classe_derive : private classe_base

{ . . .
  classe_derive(int D1,float D2) : classe_base(B1) ;
} ;

 void main()

 { classe_derive Deriv_1(5,10.2) ;
   . . .
 }
```

Ici, on précise l'appel du constructeur de la classe de base lors de l'écriture du constructeur de la classe dérivée. Dans l'appel du constructeur de `deriv_1`, c'est bien le constructeur de la classe dérivée qui sera appelé, mais après l'appel du constructeur de la classe de base.

Exemple :

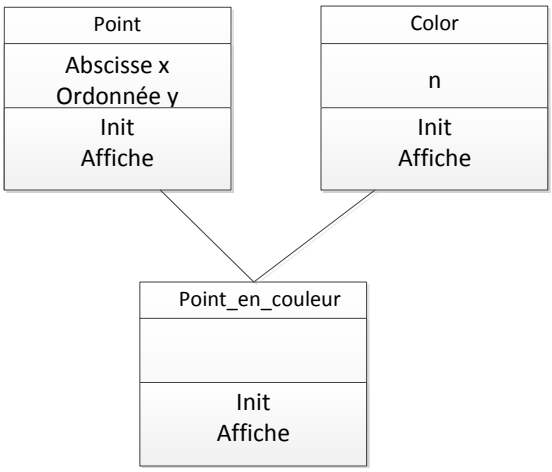


**Figure 1** Présentation des classes.

```
//#include <iostream.h> pour windows

// classe de base nommé Point de cette classe
// dérive une autre classe nommé point_en_Couleur
#include <iostream>
using namespace std;

class point
{
    protected:
```

```
            float x, y;
    public:
            void init(float, float);
            void affiche(void);
};

// classe de base , mere aussi de la classe //point_en_couleur

class color
{
    protected:
            short n;
    public:
            void init(short);
            void affiche(void);
};

//classe fille qui hérite de deux classe point et
//point_en_couleur
class point_en_couleur: public point, public color
{
      public:
           void init(float, float, short);
           void affiche(void);
};
 // ici aussi on utilise :: pour bien préciser et // lever toute
ambiguïté

void point::init(float abs, float ord)
{
     x = abs;
     y = ord;
}
// ici aussi on utilise :: pour bien préciser et // lever toute
ambiguïté sur affiche

void point::affiche(void)
{
     cout << '(' << x << ',' << y << ')';
}
void color::init(short couleur)
{
     n = couleur;
}
void color::affiche(void)
{
     cout << n;
}
```

```
void point_en_coleur::init(float abs, float ord, short couleur)

{
     point::init(abs, ord);
     coulor::init(couleur);
}

// ici on utilise :: pour dire à quel classe
// appartient la méthode pour éviter les //ambiguités

void point_en_couleur::affiche(void)
{
    point::affiche();
    cout << " avec la couleur ";
    color::affiche();
}

void main(void)
{
    point_en_couleur p;
    // p est de type point_en_couleur alors
    // il va appeler la fonction init de cette
    // même classe
    p.init(2.0, 3.0, 4);
    cout << "\n";

    // Ci-dessous c'est bien affiche de point qui
    // sera appelé

    p.point::affiche();
    cout << "\n";
    p.affiche();
    cout << "\n";
}
```

# 2. Fonctions virtuelles

Par l'intermédiaire du mécanisme des fonctions virtuelles, C++ permet de créer une liaison dynamique entre les fonctions membres; il met en place par ces procédés le polymorphisme. Ce concept permet d'appliquer une même opération à des objets de types différents, sans qu'il soit nécessaire d'en connaître le type exact.

Une fonction virtuelle est une méthode polymorphique. Elle permet aux classes dérivées de développer plusieurs versions d'une fonction de classe de base.

## 2.1 La déclaration

Pour déclarer qu'une méthode est virtuelle, on fait précéder sa déclaration de la directive `virtual`. Le mot clé placé devant la méthode permet d'indiquer au compilateur que la méthode est susceptible d'être redéfinie dans une classe dérivée.

Il n'est pas nécessaire de répéter le mot clé `virtual` dans la définition de la fonction, ni dans celle des fonctions des classes dérivées. La gestion se fait par le compilateur.

Pour redéfinir une fonction virtuelle dans la classe dérivée d'une classe de base, le nombre et le type d'arguments doivent être identiques dans les deux classes.

```
class article_Magasin

{
// ...
virtual void affiche() {
cout <<numero<<" "<< nom<<" " <<pht<<" "<<qte;
}
// ...
};

class Sport_Hiver: article

{
// ...
void affiche(){
article::affiche();
cout<<" la garantie de l'article est "<<Garantie_article;
}
// ...
};
```

La classe `article` déclare la fonction `affiche` comme virtuelle et définit une version de base. La classe `Sport_hiver` qui hérite de `article` déclare à nouveau `affiche` en la redéfinissant à sa manière.

Considérons la situation suivante :

```
class point class pointcol : public point
{    void affiche () ;  {     void affiche () ;
..... .....
} ;   } ;
..... point p ;
      pointcol pc ;
         point * adp = &p ;
```

Dans cet exemple, l'instruction `adp->affiche()` appelle la méthode `affiche` du type `point`. Mais si nous exécutons l'affectation (autorisée) `adp = & pc` le pointeur `adp` pointe maintenant sur un objet de type `pointcol`. Néanmoins, l'instruction `adp->affiche()` fait appel à la méthode `affiche` du type `point`, alors que le type `pointcol` dispose lui aussi d'une méthode `affiche`. En effet, à cause de la ligature statique, le type de l'objet est figé à la compilation.

Rendre virtuelles les méthodes permet de résoudre ce problème et de retrouver la vraie méthode.

```
class point
{  .....
  virtual void affiche () ;
  .....
} ;
```

L'instruction `virtual` indique au compilateur que les éventuels appels de la fonction *affiche* doivent utiliser une ligature dynamique et non plus une ligature statique. Autrement dit, la méthode appelée va dépendre seulement de la nature de l'objet réellement pointé.

`adp->affiche()` appelle non plus systématiquement la méthode `affiche` du type `point`, mais celle correspondant au type de l'objet réellement désigné par `adp` (ici `point` *ou* `pointcol`).

```
#include <iostream>
using namespace std ;
class D_Voiture
{
  protected:
  int H, L ;

 public:
  point (int L=0, int H=0)
      {L= longueur ; H=Honteur ;}

  virtual void affiche ()
   {cout <<"Les dimensions de la voiture sont :" << L <<
   " " << H << "\n" ;
   }
 };

class Couleur_Voiture : public D_Voiture
```

```
       {   short couleur ;
       public:
       Couleur_Voiture(int H=0, int L=0, short cl=1) :
       point (H, L)
       {
   couleur = cl ;
 }

  void affiche ()
           {cout << "Je suis une voiture \n" ;
 cout <<               " mes dimensions longueur et hauteur sont
respectivement
                         : " << L << " " << H ;
             cout << "et ma couleur est :" << couleur << "\n" ;

           }

        } ;

main()
{  D_Voiture V(8,4) ; D_Voiture * BM = &p ;
Couleur_Voiture VC (7,3,2) ; Voiture_Couleur * BMc = &pc ;
BM->affiche () ; BMc->affiche () ;
cout << "-----------------\n" ;
BM = BMc ; adp->affiche () ; adpc->affiche () ;
}
```

# 3. Héritage simple et  héritage multiple

Nous avons déjà vu que l'héritage est une technique qui facilite la réutilisation. Son objectif est de permettre la définition aisée de sous-types (sous-classes) correspondant à une spécification (une extension) de types (classes) existants. Voyons les caractéristiques générales du mécanisme de l'héritage et examinons un exemple pratique.

## 3.1 Caractéristiques

Lors de la création d'une classe, un programmeur n'est pas obligé de définir tous les attributs et méthodes de cette classe. Il peut utiliser les attributs et méthodes d'une ou de plusieurs classes existantes, appelées *classes de base* ou *superclasses*; cette technique consiste à faire hériter une classe d'un autre classe en C++ et, dans ce cas, la nouvelle classe porte le nom de *classe dérivée*.

Il est normalement possible d'ajouter des attributs et des méthodes aux classes dérivées, ce qui augmente généralement le nombre de membres qu'elles comportent. En outre, elles sont plus précises et représentent des groupes plus restreints d'objets que les classes de base. Chaque objet d'une classe dérivée en est aussi un de la superclasse. Toutefois, l'inverse n'est pas vrai : les objets d'une classe de base ne sont pas nécessairement des objets de ses classes dérivées.

Pour éviter de violer l'encapsulation de sa classe de base, une classe dérivée ne peut accéder aux membres privés de sa superclasse. Il faut donc définir une nouvelle forme d'accès qui autorise aux membres et amis des classes dérivées d'accéder aux membres non publics de leur classe de base : il s'agit de l'accès protégé (`protected`). Ce dernier constitue un niveau de protection intermédiaire entre l'accès public et l'accès privé. Autrement dit, seuls les membres et amis de la classe de base et les membres et amis des classes dérivées peuvent accéder aux membres protégés d'une classe.

La structure hiérarchique caractérisant l'héritage ressemble à celle d'un arbre, avec la classe de base au sommet de la hiérarchie. En C++, pour spécifier qu'une classe `classe1` est une classe dérivée d'une classe `classe2`, il suffit de définir `classe1` de la manière suivante :

```
class classe1 : <mode de dérivation> classe2 {
  ...
};
```

Le mode de dérivation le plus utilisé est le mode `public`, qui permet à `classe1` d'hériter des membres publics et protégés de la *classe2*. Il est alors possible de traiter les objets de `classe1` et ceux de `classe2` de la même façon. Il existe toutefois d'autres modes moins utilisés, qui sont le mode *privé (*`private`*)* et celui *protégé (*`protected`*).*

## 3.2 L'étude des différentes caractéristiques de la notion d'héritage

Voyons maintenant les différentes caractéristiques de l'héritage, soit la *dérivation*, la *redéfinition de méthodes*, le *polymorphisme* et la *méthode virtuelle* et la *liaison dynamique*.

### *3.2.1 La dérivation*

La dérivation est une technique qui consiste à faire hériter une classe d'une autre classe; en langage de programmation C++, on dit qu'une `classe B` est dérivée d'une `classe A` si elle en hérite. La `classe A` est alors appelée la *classe de base* ou

bien *superclasse* de la `classe B`. Dans l'exemple suivant, nous allons voir comment une classe `Boisson` peut hériter de la classe `Article`.

```
// Fichier Article1.h
// Définition de la classe Article
#ifndef ARTICLE1_H
#define ARTICLE1_H
#include <string.h>
class Article {
public:
            Article::Article (const char* nom, float prix, int
quantite      =      0,      float      tax      =      1.15):
prixArticle(prix),quantiteArticle(quantite),taxe(tax)
      {
            nomArticle= new char[strlen(nom)+1];
strcpy(nomArticle, nom);
      }
Article ::~Article(){ // Destructeur
            if (nomArticle)
nomArticle =""; }
      float prixsanstaxe () const{return prixArticle ;} // Prix
avant taxe
float prixavectaxe () const {return
(prixArticle*(taxe)*quantiteArticle);} // Obtenir le prix avec
taxe
protected:
            char* nomArticle; // Nom de l'article
float prixArticle; // Prix de l'article
int quantiteArticle; // Quantité en stock
float taxe; // Taxe appliquée
      };
#endif
```

La classe `Boisson` à présent :

```
// Fichier Boisson1.h
// Définition de la classe Boisson
#ifndef BOISSON1_H
#define BOISSON1_H
#include <Article1.h>
class Boisson : public Article {
public:
Boisson ::Boisson (const char* nom, float prix, int volume, int
quantite = 0): Article(nom,prix,quantite),volumeBoisson(volume)
{}
int Volume () const {return volumeBoisson;}
// Accès au volume de la boisson
protected:
```

```
int volumeBoisson; // Volume de la boisson
};
#endif
```

Nous voyons que la classe `Boisson`, qui est une sorte d'article, hérite des attributs et des méthodes définis dans la classe `Article`. Le statut de ces membres hérités est fixé par le mode de dérivation (dans ce cas le mode est public). La classe `Boisson` peut, elle aussi, définir de nouveaux attributs, comme ici le `volumeBoisson` et une nouvelle méthode appelée `Volume` qui permet d'avoir accès à cet attribut.

Lors de la création d'un objet de type `Boisson`, tous les constructeurs de la hiérarchie d'héritage sont appelés automatiquement, du plus général (la classe `Article`) au plus particulier (la classe `Boisson`).

Nous voyons, par exemple, que le type d'appel fait au constructeur de la classe `Article` est implanté dans la classe `Boisson`, en utilisant la même syntaxe que pour initialiser les attributs de classe. Si cet appel n'est pas spécifié, alors c'est le constructeur par défaut de la classe `Article` qui est invoqué.

Lors de la destruction d'un objet de type *Boisson*, les différents destructeurs sont appelés, mais dans l'ordre inverse de celui des constructeurs, c'est-à-dire de plus particulier au plus général. Dans notre exemple, c'est le destructeur de la classe *Boisson* qui est appelé en premier, puis celui de la classe *Article*.

Le langage de programmation C++ permet aussi d'utiliser l'*héritage multiple*, que nous verrons plus loin dans ce texte.

### *3.2.2 La redéfinition de méthodes*

La redéfinition d'une méthode est une des fonctionnalités de l'héritage. Supposons, pour illustrer cette technique, que nous souhaitons implanter une classe `BoissonAlcoolisee`. Cette classe hérite de la classe `Boisson` et introduit un autre attribut `taxeAlcool`. Cette classe hérite des attributs et des méthodes définis successivement dans les classes `Article` et `Boisson`. Cependant, le calcul à effectuer pour obtenir le prix avec taxe de la boisson doit être modifié de façon à prendre en compte la taxe sur les boissons alcoolisées.

L'héritage offre cette possibilité, car il permet de redéfinir la méthode `prixavectaxe` dans la classe `BoissonAlcoolisee`. Pour cela, l'en-tête de la méthode redéfinie doit être absolument identique à l'original. La méthode `prixavectaxe` originale reste

applicable sur un objet de la classe `BoissonAlcoolisee`, mais il faut pour cela la préfixer du nom de la classe où elle a été définie.

L'avantage de cette technique est qu'il est possible de disposer de deux méthodes `prixavectaxe` différentes sur des articles en fonction des spécificités exactes de l'article, en effectuant leur calcul de prix de manière différente, tout en conservant une homogénéité de nom. Mais lorsqu'une méthode surchargée est redéfinie dans une classe dérivée, la redéfinition masque toutes les définitions de la méthode de base, et pas seulement celles qui ont été redéfinies.

```
//Fichier BoissonAlcoolisee.h
#ifndef BOISSONALCOOLISEE_H
#define BOISSONALCOOLISEE_H
#include <Boisson1.h>
class BoissonAlcoolisee: public Boisson
{
public:
      BoissonAlcoolisee(const char* nom, float prix, int volume,
float taxeA, int quantite =0): Boisson(nom, prix, volume,
quantite), taxeAlcool(taxeA){}
// Redéfinition de la méthode prixavectaxe
      float prixavectaxe() const {return (prixArticle*(taxe +
taxeAlcool)*quantiteArticle);}
protected:
      float taxeAlcool;
};
#endif
```

```
//Fichier Boisson.cpp
#include <BoissonAlcoolisee.h>
// #include <conio.h> pas nécessaire avec Linux
#include <iostream>
using namespace std;
int main()
{
      Article banc ("unbanc", 250, 1); // Définir un objet
article
BoissonAlcoolisee vin ("unvin", 12, 1,5); // Un objet
BoissonAlcoolissee
cout<< "Le prix du banc est de "<<banc.prixavectaxe () <<endl;
cout<<"Le prix du vin est de "<< vin.prixavectaxe () <<endl;
```

```
cout<<"Le prix du vin sans alcool est de "<< vin.Article ::
prixavectaxe()<<endl;
// getch() ; pas nécessaire si Linux
return 0;
}
```

*3.2.3 Le polymorphisme*

Le polymorphisme est la capacité qu'ont les objets de classes différentes appartenant à une certaine hiérarchie de répondre différemment à l'appel d'une même fonction. Par exemple, si l'on définit une classe `BoissonAlcoolisee` qui dérive d'une classe `Article`, alors un objet de la classe `BoissonAlcoolisee` devient une version plus spécifique d'un objet de la classe `Article`. De ce fait, une opération pouvant s'exécuter sur un objet de `Article` peut aussi l'être sur un objet de `BoissonAlcoolisee`.

Le polymorphisme s'implante par les fonctions virtuelles. Lorsqu'une requête est faite par l'intermédiaire d'un pointeur ou d'une référence pour exécuter une fonction virtuelle, C++ choisit de faire appel à la fonction appropriée de la classe dérivée. Toutefois, lorsqu'une fonction non virtuelle est définie dans une classe de base et redéfinie dans une classe dérivée, si cette fonction est appelée à partir d'un pointeur de la classe de base, la version de la classe de base est exécutée; dans le cas contraire, la version de la classe dérivée est utilisée. Ce comportement est dit non polymorphique.

Considérons l'exemple suivant qui utilise une classe de base `Article` et une classe dérivée `BoissonAlcoolisee`.

```
// #include <iostream.h> pour windows
#include <iostream>
using namespace std; // non nécessaire dans le cas de windows
#include <Article.h>
#include <BoissonAlcoolisee.h>
//#include <conio.h> nécessaire si vous êtes sur windows
int main()
{
      // Avec les objets
// Avec les pointeurs
Article *psucre = new Article ("Sucre", 25);
BoissonAlcoolisee *pbiere =new BoissonAlcoolisee ("Biere", 12,
1,5);
cout<<psucre ->prixavectaxe () <<endl;// Article::prixavectaxe
()
cout<<pbiere ->prixavectaxe () <<endl;
// BoissonAlcoolisse::prixavectaxe ()
psucre= pbiere;
cout<<psucre ->prixavectaxe () <<endl;// Article::prixavectaxe
```

```
()
pbiere= (BoissonAlcoolisee*) psucre;// Juste, car psucre pointe
//vers un alcool
cout<<pbiere ->prixavectaxe () <<endl;
// BoissonAlcoolisse::prixavectaxe ()
// getch(); pas nécessaire avec linux
return 0;
}
```

Nous constatons qu'on ne peut accéder aux membres spécifiques de la classe `BoissonAlcoolisse` par un pointeur ou une référence de type `Article`. Les seuls membres accessibles sont donc ceux qui sont définis dans la classe `Article`. Un exemple pratique de cette remarque se trouve dans la commande :

```
cout<<psucre ->prixavectaxe () <<endl; //Article::prixavectaxe
(),
```

même si `psucre` pointe vers un objet de type `BoissonAlcoolisee`, c'est bien la méthode `Article :: prixavectaxe()` qui est appelée.

Si la conversion implicite marche très bien du type `BoissonAlcoolisee` vers le type `Article`, ce n'est pas le cas de la conversion inverse (type `Article` vers le type `BoissonAlcoolisee`) qui nécessite une conversion de type explicite (`cast`). Autrement appelée conversion explicite, cette technique permet de forcer une conversion. Cette situation est illustrée dans la ligne suivante :

```
pbiere= (BoissonAlcoolisee*) psucre
```

D'autre part, pour réaliser cette conversion, le programmeur doit être sûr qu'elle a un sens, c'est-à-dire que `psucre` pointe effectivement vers une boisson alcoolisée.

### *3.2.4 La méthode virtuelle et la liaison dynamique*

L'héritage et le polymorphisme sont des fonctionnalités très puissantes pour représenter et manipuler non seulement des objets, mais également des familles d'objets. La liaison statique des méthodes à la compilation constitue une entrave à la bonne maniabilité de ces familles d'objets. Jusqu'à présent nous avons toujours utilisé des méthodes statiques. Cependant le langage de programmation C++ permet de bénéficier d'une liaison dynamique.

Pour ce faire, il suffit de définir les méthodes utilisées comme virtuelles. Leur définition s'effectue en préfixant la méthode par le mot clé `virtual` lors de sa première déclaration. Une méthode virtuelle définie dans la classe de base le reste en effet dans

toutes les classes dérivées de la hiérarchie. Dès lors, il n'est plus nécessaire de convertir au fur et à mesure les différents articles. La méthode qui correspond au type d'objet manipulé est dynamiquement déterminée.

La déclaration d'une méthode virtuelle se fait de la manière suivante dans la définition de la classe `Article` de notre exemple :

```
virtual float prixavectaxe() const;
```

Elle signifie simplement que `prixavectaxe` est une fonction virtuelle constante qui ne prend pas d'argument et retourne une valeur de type `float`. Par ailleurs, on parle de fonction virtuelle pure lorsque la valeur de cette fonction est initialisée à `zéro`. Ainsi, la fonction `prixavectaxe` peut être déclarée virtuelle pure de la manière suivante :

```
virtual float prixavectaxe () const = 0;
```

Une classe où sont définies une ou plusieurs fonctions virtuelles pures est dite abstraite.

Il existe deux mécanismes permettant à un programme de déterminer à quelle classe s'applique une fonction virtuelle.

- Le premier utilise un pointeur pour choisir dynamiquement, au moment de l'exécution du programme, la fonction à exécuter : il s'agit de la *liaison dynamique (dynamic binding)*. Par exemple, si l'on définit un pointeur `pbiere` vers une classe `BoissonAlcoolisee`, l'application de la fonction virtuelle `prixavectaxe` de la classe `BoissonAlcoolisee` se fait de la manière suivante :

```
pbiere -> prixavectaxe();
```

- Quant au second mécanisme, il utilise une variable de référence à l'objet sur lequel on veut exécuter la méthode : il s'agit de la liaison statique (*static binding*). Ainsi, si l'on déclare `Objetbiere` un objet d'une classe `BoissonAlcoolisee`, alors l'exécution de la méthode en rapport avec cet objet se fait de la manière suivante :

```
Objetbiere.prixavectaxe();
```

## 3.3 L'héritage multiple

Jusqu'à présent, nous avons montré comment implémenter l'héritage simple. Étudions maintenant le cas de l'héritage multiple par lequel une classe peut hériter de plusieurs

classes de base. Cette particularité encourage des formes intéressantes de réutilisation de logiciels, mais elle peut aussi engendrer des ambiguïtés de toutes sortes.

Les deux exemples un peu plus bas illustrent une implémentation d'héritage multiple avec deux classes de base et une classe dérivée.

Exemple 1 :

```
// Fichier GetNumeroSerie1.H
// Définition de la classe GetNumeroSerie1
#ifndef GETNUMEROSERIE1_H
#define GETNUMEROSERIE1_H
class GetNumeroSerie {
public:
GetNumeroSerie (int x) {valeur = x;}
int getData() const { return valeur; }
protected:
int valeur; //Accessible par les classes dérivées de
Getnumeroserie1
};
#endif
```

La classe de base contient un attribut protégé (*valeur*), un constructeur qui le rajuste et une fonction publique `getData` qui retourne la valeur de l'attribut en question.

Exemple 2 :

```
// Fichier GetLettre1.h
// Définition de la classe GetLettre
#ifndef GETLETTRE1_H
#define GETLETTRE1_H
class GetLettre {
public:
GetLettre (char c) { lettre = c; }
char getData() const { return lettre; }
protected:
char lettre; //Accessible par les classes dérivées de GetLettre
};
#endif
```

La seconde classe est définie de façon similaire, à la différence que son attribut protégé (`lettre`) est de type `char`. Il en résulte que la fonction publique `getData` de `GetLettre` retourne la valeur de `lettre` qui est aussi de type `char`.

Par ailleurs, pour spécifier qu'une classe dérivée hérite de plusieurs classes, on fait suivre son nom d'un deux-points : et de la liste des superclasses séparées par des

virgules. Par exemple, si l'on définit une classe `CleLogiciel` qui hérite publiquement des classes `GetNumeroSerie` et `GetLettre`, sa déclaration se fait de la manière suivante :

```
class CleLogiciel:public GetNumeroSerie, public GetLettre
```

Le prochain exemple illustre la définition complète d'une classe, ainsi que de ses fonctions membres. La première classe contient, entre autres, un attribut privé (`entier`) et une fonction publique (`getEntier`) qui lit cet attribut (durée en jours de la licence).

Notons aussi que le constructeur de la classe `CleLogiciel` fait appel aux constructeurs de `GetNumeroSerie` et de `GetLettre`, en utilisant une syntaxe d'initialisation bien particulière. Encore une fois, les constructeurs de la classe de base ne sont pas appelés dans l'ordre où ils sont définis, mais dans l'ordre de spécification de l'héritage.

```
// Fichier CleLogiciel1.h
// Définition de la classe dérivée qui hérite
// des classes de base GetNumeroSerie et GetLettre.
#define CLELOGICIEL_H
#define CLELOGICIEL_H
#include <GetLettre1.h>
#include <GetNumeroSerie1.h>
// Héritage multiple
class CleLogiciel : public GetNumeroSerie, public GetLettre {
friend ostream &operator<<(ostream &, const CleLogiciel &);
public:
CleLogiciel (int, char, int);
int getEntier() const;
private:
int entier; // Attribut privé de la classe CleLogiciel
};
#endif
```

```
// Fichier Clelogiciel.cpp
// Définition des fonctions membres de la classe dérivée
// #include <iostream.h> utilisé pour windows
#include <iostream>
using namespace std;
#include <CleLogiciel1.h>
// Le constructeur de la classe CleLogiciel appelle les
constructeurs des
// classes GetNumeroSerie et GetLettre.
CleLogiciel:: CleLogiciel (int i, char c, int f)
      : GetNumeroSerie (i), GetLettre(c)
// Appel du constructeur de chaque classe de base
```

```
{ entier = f; }
// Retourner la valeur de entier
int CleLogiciel::getEntier() const { return entier; }
// Afficher tous les attributs membres de CleLogiciel
ostream &operator<<(ostream &output, const CleLogiciel &d)
{
      output << " l'entier: " << d.valeur << endl
            << " le caractère: " << d.lettre << endl
<< " le nombre de jours de la licence: " << d.entier;
      return output; // Concaténation des valeurs à afficher
}
```

D'autre part, l'opérateur surchargé d'insertion de la classe `CleLogiciel` utilise la notation point **.** pour accéder à l'objet `d` de `CleLogiciel` et pour afficher à l'écran la valeur de ses attributs : `valeur`, `lettre` et `entier`.

Cette fonction est déclarée amie de `CleLogiciel`, de sorte que l'opérateur `<<` peut directement accéder au membre privé entier. De plus, elle peut respectivement accéder aux membres protégés `valeur` de `GetNumeroSerie` et `lettre` de `GetLettre`.

Examinons maintenant le programme principal. D'abord, les objets `numeroserie` de `GetNumeroSerie` et `premierelettre` de `GetLettre` sont créés et initialisés respectivement au nombre `6707` et au caractère `T`. Ensuite, l'objet `d` de la classe `CleLogiciel` est créé et initialisé pour contenir l'entier `60`, le caractère `A` et le second entier `120`.

Le contenu des objets de chaque classe est affiché en appelant la fonction membre `getData` de chaque objet. D'ailleurs, même s'il existe deux fonctions `getData`, les appels ne sont pas ambigus puisqu'ils se réfèrent directement à la version de `numeroserie` et à celle de `premierelettre` de `getData`. Ainsi, l'ambiguïté créée par la fonction `getData` peut être résolue en utilisant l'opérateur point, comme dans :

```
d.GetNumeroSerie »:: getData ()
```

Pour afficher la valeur numérique de l'attribut `valeur` et `d`.

```
GetLettre::getData()
```

Pour afficher le caractère représenté par `lettre`. Par contre, la valeur du second entier est affichée sans ambiguïté avec la commande :

```
d.getEntier()
```

Ensuite, l'adresse de l'objet `d` de `CleLogiciel` est affectée au pointeur `GetNumeroSeriePtr` de `GetNumeroSerie` et la valeur numérique de *valeur* est affichée en faisant appel à `getData` de `GetNumeroSerie` par l'intermédiaire de`GetNumeroSeriePtr`.

Cette opération se répète avec le pointeur de `GetLettre`, ce qui donne lieu à l'affichage de la valeur de `lettre` qui est `T`.

Pour illustrer l'héritage multiple :

```
// Programme principal de l'héritage multiple
// #include <iostream.h> à utiliser pour windows
#include <iostream>
using namespace std;
#include <GetLettre1.h>
#include <GetNumeroSerie1.h>
#include <CleLogiciel1.h>
main ()
{
GetNumeroSerie n1(10), *n1Ptr;
// Création d'objets de la classe GetNumeroSerie
GetLettre l2('Z'), *l2Ptr;
// Création d'objets de la classe GetLettre1
CleLogiciel d(7, 'A',3);
// Création d'objets de la classe CleLogiciel1
// Afficher les données membres de la classe de base
cout << "L'objet b1 contient le nombre entier "
      << n1.getData() << endl
<< "L'objet b2 contient le caractère "
<< l2.getData() << endl
<< "L'objet d contient :" << endl << d << endl << endl;
// Afficher les attributs des objets de la classe CleLogiciel1
// Opérateur point résout ambiguïté causée par getData
cout << "Les attributs de CleLogiciel1 peuvent être "
      << " accessibles individuellement :" << endl
<< " l'entier " << d. GetLettre::getData() << endl
<< " le caractère " << d. GetLettre::getData() << endl
<< " le nombre entier " << d.getEntier() << endl << endl;
cout << "La classe CleLogiciel peut être traitée comme un"
      << "objet de l'une des classes de base :" << endl;
// Traiter la classe CleLogiciel comme un objet de Base1
n1Ptr = &d;
cout << "n1Ptr->getData() donne " << n1Ptr->getData() << endl ;
// Traiter la classe CleLogiciel1comme un objet de Base2
l2Ptr = &d;
cout << "l2Ptr->getData() donne " << l2Ptr->getData() << endl;
return 0;
```

```
}
```

Réponses :

```
L'objet b1 contient le nombre entier 10
L'objet b2 contient le caractère Z
L'objet d contient :
      l'entier 7
le caractère A
le nombre de jours de la licence 3
Les attributs de CleLogiciel1 peuvent être accessibles
individuellement :
      l'entier 7
le caractère A
le nombre entier 3
La classe CleLogiciel1 peut être traitée comme un objet de l'une
des classes de base :
      n1Ptr->getData() donne 7
l2Ptr->getData() donne A
```

## 3.4 Étude de cas : l'héritage d'interface et d'implémentation

L'exemple qui suit présente la hiérarchie entre les classes `Point`, `Rectangle` et `Cube`, en mettant l'accent sur une classe de base abstraite appelée `Figure`, contenant deux fonctions virtuelles pures : `afficherNomFigures` et `afficher`.

De plus, cette classe `Figure` contient deux fonctions virtuelles `aire` et `volume` héritées par `Point`, classe dérivée de `Figure`. Quant à la classe `Rectangle`, elle possède sa propre implémentation de la fonction `aire`, mais hérite de la fonction `volume` de la classe `Point`. La classe `Cube` contient sa propre implémentation de la fonction `volume` et hérite de la fonction `aire` de la classe `Rectangle`.

La classe de base `Figure` se compose de quatre fonctions virtuelles publiques, mais ne contient pas de données membres. Les fonctions `afficherNomFigures` et `afficher` y sont déclarées virtuelles pures et, de ce fait, sont redéfinies dans chacune des classes dérivées. Quant aux fonctions `aire` et `volume`, elles y sont définies pour retourner la valeur numérique `0.0`. Ces fonctions doivent être redéfinies dans les classes dérivées dont les calculs d'aire et de volume sont non nuls.

Voici le code de la classe `Figure` :

```
// Fichier Figure.h
// Définition de la classe de base Figure
#ifndef FIGURE_H
```

```
#define FIGURE_H
class Figure {
public:
      virtual float aire () const {return 0.0;}
virtual float volume () const {return 0.0; }
virtual void afficherNomFigures() const = 0;
// Fonction virtuelle pure
virtual void afficher() const = 0;
// Fonction virtuelle pure
};
#endif
```

La classe `Point` dérive de la classe `Figure` avec héritage public. Puisqu'un point n'a pas d'aire, ni de volume, les fonctions `aire` et `volume` de `Figure` n'y sont pas redéfinies : la classe `Point` en hérite telles qu'elles sont définies dans `Figure`.

Par contre, les fonctions virtuelles pures `afficherNomFigures` et `afficher` de `Figure` sont redéfinies dans `Point`. De plus, pour affecter de nouvelles coordonnées `x` et `y` à `Point`, on utilise la fonction `set`, alors que pour retourner ces valeurs, on utilise la fonction `get`.

Voici le code de la classe `Point` :

```
// Fichier Point1.h
// Définition de la classe Point
#ifndef POINT1_H
#define POINT1_H
#include <iostream>
#include <Figure.h>
class Point : public Figure {
      friend ostream &operator<<(ostream &, const Point &);
public:
      Point(float = 0, float = 0); // Constructeur par défaut
void setPoint(float, float);
float getX() const { return x; }
float getY() const { return y; }
virtual void afficherNomFigures() const {cout << "Point: "; }
virtual void afficher () const;
private:
      float x, y; // Coordonnées x et y de Point
};
#endif
```

Voici le code des fonctions membres de la classe `Point` :

```
// Fichier Point1.cpp
// Définition des fonctions membres de la classe Point.
// #include <iostream.h> utilisé pour windows
#include <iostream>
using namespace std;
#include <Point1.h>
Point::Point(float a, float b) { setPoint(a, b); }
void Point::setPoint(float a, float b)
{
      x = a;
y = b;
}
void Point::afficher() const { cout << "[" << x << ", " << y <<
"]"; }
ostream &operator<<(ostream &output, const Point &p)
{
      p.afficher(); // Affichage de l'objet à l'écran
return output; // Autorise la concaténation des données à
afficher
}
```

La classe `Rectangle`, illustrée aux figures 5.3.4-4 et 5.3.4-5, dérive de la classe `Point` avec héritage public. Puisqu'un rectangle n'a pas de volume, la fonction `volume` de `Point`, qui est implicitement la même que celle qui est définie dans la classe `Figure`, n'y est pas redéfinie.

Toutefois, l'aire d'un rectangle est généralement non nulle, de sorte que la fonction `aire` définie dans `Figure` pour retourner une valeur nulle doit être redéfinie. Quant aux fonctions virtuelles `afficherNomFigures` et `afficher` elles doivent être redéfinies dans `Rectangle`, sinon elles seront héritées de `Point`. En outre, la classe `Rectangle` ne contient pas d'attributs privés.

Voici le code de la classe `Rectangle` :

```
// Fichier Rectangle1.h
// Définition de la classe Cercle
#ifndef RECTANGLE1_H
#define RECTANGLE1_H
#include <Point1.h>
class Rectangle : public Point {
      friend ostream &operator<<(ostream &, const Rectangle &);
public:
      // Constructeur par défaut
Rectangle(float longueur = 0.0, float largeur = 0.0);
virtual float aire() const;
```

```
float getlongueur() const { return longueur; }
float getlargeur() const { return largeur; }
virtual void afficher() const;
virtual void afficherNomFigures() const { cout << "Rectangle: ";
}
private:
      float longueur ;
float largeur ;
};
#endif
```

Voici le code des fonctions membres de la classe `Rectangle` :

```
// Fichier Rectangle1.cpp
// Définition des fonctions membres de la classe Rectangle
// #include <iostream.h> à utiliser pour windows
// #include <iomanip.h> à utiliser pour windows
#include <iostream>
using namespace std;
#include <iomanip>
#include <Rectangle1.h>
Rectangle::Rectangle(float a, float b)
      {longueur=a, largeur=b;}
// Appel du constructeur de la classe de base
float Rectangle::aire() const { return longueur*largeur; }
void Rectangle::afficher() const
{
cout << "["<< getlongueur() << ", " << getlargeur()<< "]";
}
ostream &operator<<(ostream &output, const Rectangle &r)
{
      r.afficher();
// Affichage de l'objet à l'écran
return output;
// Autorise concaténation lors de l'affichage
}
```

Quant à la classe `Cube`, elle dérive de la classe `Rectangle` avec interface publique. Dans ce cas, les fonctions `aire`, `volume`, `afficherNomFigures` et `afficher` sont toutes redéfinies. De plus, pour rajuster la valeur de la hauteur, on utilise la fonction `set` , alors que pour retourner cette valeur, on utilise la fonction `get`.

Voici le code de la classe `Cube` :

```
// Fichier Cube1.h
// Définition de la classe Cube
#ifndef CUBE1_H
```

```
#define CUBE1_H
#include "Rectangle1.h"
class Cube : public Rectangle
{
friend ostream &operator<<(ostream &, const Cube &);
public:
// Constructeur par défaut
Cube(float h = 0.0,
float x = 0.0, float y = 0.0);
void setHauteur(float);
virtual float aire() const;
virtual float volume() const;
virtual void afficherNomFigures() const { cout << "Cube: ";
}
virtual void afficher() const;
private:
float hauteur; // Hauteur de cube
};
#endif
```

Voici le code des fonctions membres de la classe `Cube` :

```
// Fichier Cube1.cpp
// Définition des fonctions membres et amies de la classe Cube
// #include <iostream.h> utilisé pour windows
#include <iostream>
#include <iomanip>
using namespace std;
//#include <iomanip.h> à utiliser pour windows
#include <Cube1.h>
Cube::Cube(float h, float x, float y)
: Rectangle(x, y) // Appel du constructeur de la classe de base
{ setHauteur(h) ; }
void Cube::setHauteur(float h) { hauteur = h ;}
float Cube::aire() const
{
// Aire de Cube
return 6 * Rectangle::aire() ;
}
float  Cube::volume()  const  {  return  1/6*Rectangle::aire()  *
hauteur; }
void Cube::afficher() const
{
cout << "["<< getX() << ", " << getY()<< "]" << "Hauteur=" <<
hauteur;
}
ostream &operator<<(ostream &output, const Cube& c)
{
```

```
c.afficher(); // Affichage de l'objet à l'écran
return output; // Autorise la concaténation à l'affichage
}
```

Le programme principal (voir plus bas) commence par déclarer et initialiser les objets `point` de la classe `Point`, `rectangle` de `Rectangle` et `cube` de `Cube`. La fonction `afficherNomFigures` est appelée pour afficher les attributs de chaque objet, de manière à montrer que les objets sont correctement initialisés. Ensuite, un tableau `tableauFigures` de `typeFigure` * est déclaré et utilisé pour contenir les adresses des objets des classes dérivées.

Autrement dit, l'adresse de `point` (objet de Point) est affectée à `tableauFigures[0]`, celle de rectangle à `tableauFigures[1]`, celle de cube à `tableauFigures[2]`. En outre, une boucle for est utilisée pour exécuter les opérations suivantes :

```
tableauFigures[i]->afficherNomFigure();
tableauFigures[i]->afficher();
tableauFifures[i]->aire();
tableauFigures[i]->volume();
```

Voici le code du programme principal :

```
// Programme principal de la hiérarchie formée par Point,
Rectangle, Cube
//#include <iostream.h> utilisé dans un environnement Windows
//#include <iomanip.h> utilisé dans un environnement Windows
//#include <conio.h> utilisé dans un environnement Windows
#include <iostream>
using namespace std;
#include <iomanip>
#include <Figure.h>
#include <Point1.h>
#include <Rectangle1.h>
#include <Cube1.h>
main()
{
      Point point (6, 7); // Créer l'objet point
Rectangle rectangle (11, 8); // Créer l'objet rectangle
Cube cube (20, 20, 20); // Créer l'objet cube
point.afficherNomFigures(); // Liaison statique
cout << point << endl;
rectangle.afficherNomFigures(); // Liaison statique
cout << rectangle << endl;
cube.afficherNomFigures(); // Liaison statique
```

```
cout << cube << "\n\n";
Figure *tableauFigures[3];// Tableau de pointeurs de type
                          // Figure *
tableauFigures[0] = &point; // Contient l'adresse de l'objet de
Point
tableauFigures [1]= &rectangle; // Contient l'adresse de l'objet
//de Rectangle
tableauFigures[2] = &cube; // Contient l'adresse de l'objet
                           // de Cube
                           //  Affichage du nom de la forme,
//l'aire et le volume de chaque objet
                           // vers lequel pointe le tableau en
//utilisant la liaison dynamique.
for (int i = 0; i < 3; i++) {
            tableauFigures[i]->afficherNomFigures();
cout << endl;
tableauFigures[i]->afficher();
cout << "\nAire = " << tableauFigures[i]->aire()<< "\nVolume = "
<< tableauFigures[i]->volume()<< endl << endl;
      }
getch();
return 0;
}
```

Voici les résultats de l'exécution de ce programme qui illustre le polymorphisme :

```
Point : [6, 7]
Rectangle : [11, 8]
Cube : [20, 20]; Hauteur = 20
Point :
      [6, 7]
Aire = 0
Volume = 0
Rectangle :
      [11, 8]
Aire = 88
Volume = 0
Cube :
      [20, 20]; Hauteur = 20
Aire = 2400
Volume = 0
```

# 4. Surcharge d'opérateurs

Les surcharges d'opérateurs permettent de redéfinir certains opérateurs qui existent déjà dans le langage de programmation. La surcharge consiste donc à donner plusieurs significations à un même opérateur dans un programme. Dans cette section, nous verrons quand et comment surcharger des opérateurs, et nous présenterons plusieurs programmes illustrant cette notion de surcharge des opérateurs.

## 4.1 Principes de base

Dans la pratique, la surcharge d'opérateurs offre aux développeurs de logiciels une souplesse supplémentaire qui leur permet non seulement de définir de nouveaux types, mais surtout de manipuler les objets avec des expressions concises. Même si le langage de programmation C++ ne permet pas d'inventer de nouveaux opérateurs, il permet de surcharger la plupart des opérateurs existants. Autrement dit, lorsque ces derniers sont utilisés avec des objets, ils peuvent avoir un sens qui dépend du type de ces objets. Par exemple, l'opérateur d'affectation `=` agit différemment, selon qu'il traite des variables de type `int`, `float`, `double` ou `construit`.

En C++, la surcharge d'un opérateur se fait en déclarant et en définissant une fonction formée du mot réservé `operator`, suivi du symbole de l'opération à surcharger. Par exemple, la fonction `operator+` peut être utilisée pour surcharger l'opérateur de l'addition (`+`). La surcharge est particulièrement appropriée pour des classes mathématiques qui exigent qu'un nombre important d'opérateurs soient surchargés pour assurer la consistance des classes lors de leur manipulation.

La surcharge d'opérateurs en C++ facilite l'utilisation d'expressions concises aussi bien pour des types définis par le programmeur que pour des types construits à partir de la collection d'opérateurs de C++. Une telle pratique n'est cependant pas automatique : le programmeur doit explicitement préciser l'opérateur à surcharger avant de l'utiliser.

## 4.2 Limitations

La plupart des opérateurs de C++ peuvent être surchargés, sauf les cinq opérateurs suivants : le *point* `.` , le *point suivi d'un astérisque* `.*`, le *double deux-points* `::` , le *point d'interrogation suivi du deux-points* `?:` , ainsi que `sizeof`. Notons que la surcharge ne change pas certaines caractéristiques telles que l'ordre de priorité ou l'associativité de ces opérateurs. De plus, il faut conserver la pluralité (unaire et binaire) de l'opérateur

initial. Ainsi, nous pouvons surcharger un opérateur `+`, qu'il soit unaire ou binaire, mais nous ne pouvons pas définir de `/` unaire ou de `-` binaire.

Toutefois, certaines opérations sont interdites s'il y a surcharge. Ainsi, on ne peut utiliser d'arguments par défaut avec des opérateurs surchargés. En outre, il n'est pas permis de créer de nouveaux opérateurs dans le but de les surcharger : seulement les opérateurs existants peuvent être surchargés. Soulignons aussi que la surcharge d'opérateurs fonctionne seulement avec des objets ou une combinaison d'objets de type *construit* ou *défini* par l'utilisateur. Par ailleurs, la surcharge d'une combinaison d'addition et d'affectation ne correspond pas nécessairement à celle de l'opérateur `+=`. Par exemple, si l'on considère la surcharge des opérateurs `+` et `=` dans l'expression :

```
objet2 = objet2 + objet1;
```

même si cette expression peut s'écrire de la manière suivante :

```
objet2 += objet1;
```

L'opérateur `+=` n'est pas nécessairement surchargé.

## 4.3 Étude de cas

Pour terminer cette section sur la surcharge des opérateurs, présentons deux cas : la surcharge d'opérateurs binaires et celle des opérateurs d'insertion-d'extraction.

### *4.3.1 La surcharge d'opérateurs binaires*

Les fonctions de surcharge d'opérateurs peuvent être considérées comme membres (surcharge d'opérateur interne) ou non membres (normalement amis) des classes qui les utilisent. Lorsque les opérateurs binaires sont déclarés comme des fonctions non statiques surchargées, ils peuvent être définis avec un ou plusieurs arguments. Par exemple, si l'opérateur `+=` est surchargé pour effectuer l'addition de deux objets de même type de *données*, il peut prendre un ou deux arguments.

Supposons `objet1` et `objet2`, deux objets d'une classe quelconque, par exemple une classe permettant d'additionner deux objets d'une classe `Complexe`, alors l'expression `x += y` peut être traitée comme si l'on avait la commande :

```
Objet1.operator+= (objet2)
```

Cela amène à faire appel à la fonction membre `operator+=` qui prend un argument et qui est déclarée de la manière suivante :

```
class Complexe {
public:
Complexe &operator+= (const Complexe &);
...
};
```

D'autre part, lorsque l'opérateur binaire `+=` est surchargé comme une fonction non membre (surcharge d'opérateur externe), il prend deux arguments dont l'un est un objet ou une référence à un objet. Dans ce cas, si `objet1` et `objet2` sont des objets de la classe `Complexe` définie précédemment ou des références à ses objets, alors la commande « `objet1 += objet2` » est traitée comme s'il s'agissait de :

```
operator+= (objet1, objet2)
```

faisant appel à la fonction non membre et amie `operator+=`, déclarée de la manière suivante :

```
class Complexe{
  friend Complexe &operator+= (const Complexe &, const Complexe
&);
  ...
};
```

Ces deux définitions de la classe `Complexe` nous montrent comment définir l'opérateur binaire `+=` surchargé, comme fonction membre et non membre d'une classe donnée.

### *4.3.2 La surcharge des opérateurs d'extraction et d'insertion*

Le langage C++ est capable de contrôler les entrées-sorties de données standardisées au moyen des opérateurs d'extraction-d'insertion (*extraction/insertion operators*), représentés par les opérateurs `>>` et `<<`.

Ces opérateurs sont surchargés à l'intérieur même des bibliothèques fournies avec le compilateur, de façon à pouvoir traiter tout type de données, incluant les chaînes de caractères et les adresses. Ils peuvent aussi être surchargés comme des fonctions non membres (surcharge externe) pour agir sur des types construits, c'est-à-dire définis par l'utilisateur. De plus, lorsqu'elle est surchargée, la fonction `operator<<` est précédée de `ostream&`, alors que la fonction `operator>>` est précédée de `istream &`.

Le prochain programme illustre la surcharge de ces opérateurs pour manipuler les données d'une classe `Complexe`, définie par un utilisateur, en supposant que l'entrée des parties *imaginaire* et *réelle* se fasse correctement.

```
// Surcharge des opérateurs d'insertion/d'extraction
```

```
// #include <iostream.h> pour Windows
// #include <conio.h> pour windows pas nécessaire avec linux
#include <iostream>
using namespace std;

class Complexe {
public:
      friend ostream &operator<<(ostream &, const Complexe &);
friend istream &operator>>(istream &, Complexe &);
private:
      float imaginaire; // Partie imaginaire
float reel; // Partie réelle
};
//  Surcharge  de  l'opérateur  d'insertion  (ne  peut  être  une
fonction membre).
ostream &operator<<(ostream &sortie, const Complexe &nombreC)
{
      sortie << nombreC.reel << " +"<<"i" << nombreC.imaginaire;
return sortie; // Permet l'opération « cout << a << b »;
}
// Surcharge de l'opérateur d'extraction
istream &operator>>(istream &entree, Complexe &nombreC)
{
      entree>> (nombreC.reel); // Entrée de la partie réelle
entree>> (nombreC.imaginaire); // Entrée de la partie imaginaire
return entree; // Permet l'opération « cin >> a >> b »;
}
int main()
{
      Complexe complexe1; //Ccréer un objet de type complexe
cout << "Entrez un nombre complexe parties reelle et imaginaire
"<< endl;
cin >> complexe1; // Fait appel à la fonction operator>> par
                  // l'appel de operator>> (cin, complexe1).
cout << "Voici le complexe1 :" << endl
<< complexe1 << endl; // Fait appel à la fonction operator<< par
                      // l'appel de operator<<(cout, complexe1).
return 0;
}
```

Dans ce programme, la fonction d'extraction `operator>>` prend comme arguments une référence à `istream (entree)` et une référence à la classe `Complexe (complexe1)`, et retourne la référence de `istream`. Elle est utilisée pour enregistrer le nombre complexe sous la forme `partie réelle +i partie imaginaire`

Ainsi, lorsque le compilateur rencontre l'expression :

```
Cin >> complexe1
```

dans la fonction `main`, il lance automatiquement la fonction :

```
operator>>(cin, complexe1);
```

Cette dernière fait alors la lecture de la partie réelle et de la partie imaginaire du nombre complexe. Ainsi, la fonction `operator>>` permet de lire des objets de la classe `Complexe` ou des objets d'autres types. Par exemple, si l'on reçoit comme entrées deux objets de la manière suivante :

```
Cin >> complexe1>> complexe2;
```

l'expression `cin >> complexe1` est d'abord exécutée en appelant la fonction :

```
operator>>(cin, complexe1);
```

Cet appel retourne une référence à `cin`, de sorte que la portion restante de l'expression soit interprétée comme `cin >> complexe1`. Cela est exécuté par l'appel :

```
operator>>(cout, complexe1);
```

Dans le même ordre d'idée, l'opérateur d'insertion `operator<<` prend comme arguments une référence à `ostream (output)` et une référence à la classe `Complexe (complexe1)` pour retourner la référence de `ostream`. Ainsi, lorsque le compilateur rencontre l'expression :

```
cout << complexe1
```

dans la fonction `main`, il lance automatiquement la fonction :

```
operator<<(cout, complexe1);
```

qui affiche alors les parties réelle et imaginaire du nombre complexe.

Pour finir, notons que les fonctions `operator>>` et `operator<<` sont déclarées comme fonctions amies de `Complexe`. Dans la pratique, les opérateurs surchargés d'entrées-sorties peuvent être déclarés comme membres amis d'une classe, s'ils doivent directement accéder aux membres non publics de cette classe, et ceci, dans un souci de programmation efficace.

# 5. Fonctions et classes amies

Le langage C++ propose une manière particulière d'accéder aux membres privés d'autres classes. Cela permet de limiter à une seule classe l'accès à des données privées et d'éviter de les rendre publiques et disponibles pour tout objet ou classe.

## 5.1 Les fonctions amies

En langage C++, l'unité de protection est la classe, et non pas l'objet. Cela signifie qu'une fonction membre d'une classe peut accéder à tous les membres privés de n'importe quel objet de sa classe. En revanche, ces membres privés restent inaccessibles à n'importe quelle fonction membre d'une autre classe ou à n'importe quelle fonction indépendante.

La notion de fonction amie, ou plus exactement de « déclaration d'amitié », permet de déclarer dans une classe les fonctions que l'on autorise à accéder à ses membres privés (données ou fonctions).

Pour rendre une fonction amie d'une classe, il suffit d'inclure son interface précédée du mot clé *friend* au sein de la déclaration de la classe.

Nous présentons ci-dessous un exemple d'une fonction indépendante, amie d'une classe *A*.

```
class A
{
 ......
 friend --- fct (-----) ;
 .....
}
```

Dans cet exemple, la fonction `fct` ayant le prototype spécifié est autorisée à accéder aux membres privés de la classe `A`.

L'exemple suivant illustre une fonction membre d'une classe `B`, amie d'une autre classe `A`.

```
class A
{
  .....
  friend --- B:fct (-----) ;
  .....
} ;
```

La fonction `fct`, membre de la classe `B`, ayant le prototype spécifié, est autorisée à accéder aux membres privés de la classe `A`.

Pour qu'il puisse compiler convenablement la déclaration de `A`, donc en particulier la déclaration d'amitié relative à `fct`, le compilateur devra connaître la déclaration de `B`, mais pas nécessairement sa définition.

Généralement, la fonction membre `fct` possédera un argument ou une valeur de retour de type `A`, ce qui justifiera sa déclaration d'amitié. Pour compiler sa déclaration, au sein de la déclaration de `A`, il suffira au compilateur de savoir que `A` est une classe; si sa déclaration n'est pas connue à ce stade-ci, on pourra se contenter de :

```
class A ;
```

En revanche, pour compiler la définition de `fct`, le compilateur devra posséder les caractéristiques de `A`, donc disposer de sa déclaration.

## 5.2 Les classes amies

Il arrive parfois que l'on souhaite accorder des accès plus restreints que les accès publics, protégés ou privés proposés par défaut. Pour accéder aux membres privés d'une classe, par exemple, une fonction définie à l'extérieur du champ de visibilité de cette classe doit être déclarée comme son *amie*. Une classe entière peut être déclarée membre ami ou classe amie d'une autre classe.

La syntaxe de sa déclaration se fait en faisant précéder son nom du mot réservé `friend` dans la définition de la classe amie. Autrement dit, pour déclarer que `Classe2` est amie de `Classe1`, il suffit d'inclure dans la définition de `Classe1` une déclaration de la forme :

```
friend Classe2;
```

Comme toutes les classes, les classes amies ont aussi des propriétés. Présentons maintenant quelques propriétés des classes amies.

Pour qu'une classe `B` soit amie d'une classe `A`, la classe `A` doit explicitement déclarer la classe `B` comme amie. De plus, le lien d'amitié entre les classes `A` et `B` n'est ni transitif, ni symétrique.

Autrement dit, si `A` est une classe amie de `B` et que `B` est une classe amie de `C`, on ne peut conclure que `B` est une classe amie de `A`, ou que `C` est une classe amie de `B`, ou que `A` est une classe amie de `C`.

Le code suivant illustre la déclaration et l'utilisation d'une fonction amie `MettreAjour` qui accède aux attributs privés d'une classe `BulletindeNotes` pour ajuster ses données. Notons que, par convention, la déclaration des membres amis se fait avant celle des membres publics et privés d'une classe.

```
// Accès aux membres privés d'une classe par une fonction amie.
#include <iostream> // sans le .h si Linux
using namespace std; //nécessaire pour Linux
class BulletindeNotes
{
friend void MettreAjour(BulletindeNotes &, float);
// Déclaration de fonction amie
public:
      ulletindeNotes () {x = 0;} // Constructeur
void afficherNotes () const {cout << x << endl;} // Sortie
private:
      float x; // Membre privé
};
// Peut modifier les membres privés de BulletindeNotes puisque
// MettreAjour est déclarée fonction amie de Bulletin de note
void MettreAjour (BulletindeNotes &c, float val)
{
c.x = val; // Commande légale puisque MettreAjour est
           // une fonction amie de Bulletin de note
}
int main ()
{
BulletindeNotes bulletin;
cout << " bulletin.x après instanciation: ";
bulletin. afficherNotes ();
cout <<" bulletin.x après l'appel de la fonction amie
MettreAjour: ";
MettreAjour (bulletin, 18); // Mettre à jour x avec un ami
bulletin. afficherNotes ();
return 0;
}
```

## 5.3 L 'utilisation du pointeur *this*

On a parfois besoin de désigner, à l'intérieur d'une fonction membre, l'objet qui est manipulé par cette fonction. Comment peut-on le désigner alors qu'il n'existe aucune variable qui le représente dans la fonction membre?

Pour ce faire, le langage de programmation C++ utilise un pointeur particulier appelé `this`. Ce pointeur constitue un argument implicite lors de l'appel des membres de cet

objet. Son type dépend également du type de l'objet et de la fonction membre dans laquelle il est utilisé; de plus. Enfin, il ne peut être utilisé qu'à l'intérieur des fonctions non statiques.

Le code qui suit illustre l'utilisation explicite du pointeur `this` à l'intérieur de la fonction `afficheruneTension` de la classe `Afficheur`.

```
// Utilisation du pointeur this pour faire appel à des membres
d'un objet
// #include <iostream.h> sans .h avec linux
#include <iostream>
using namespace std; // nécessaire si Linux

// #include <conio.h> nécessaire si Windows
class Afficheur
{
public:
Afficheur (int = 0); // Constructeur par défaut
void afficheruneTension () const;
private:
int uneTension;
};
Afficheur::Afficheur(int a) { uneTension = a; } // Constructeur
void Afficheur::afficheruneTension () const
{
cout << " uneTension = " << uneTension << endl
      << " this-> uneTension = " << this-> uneTension << endl
<< " (*this). uneTension = " << (*this). uneTension << endl;
}
int main ()
{
Afficheur autreTension (120);
autreTension.afficheruneTension();
// getch(); pas nécessaire si Linux
return 0;
}
```

Résultats :

```
uneTension = 120
this-> uneTension = 120
(*this). uneTension = 120
```

Supposons un voltmètre ou un appareil qui mesure la tension pour accéder à une donnée privée `uneTension`.

Dans un premier temps, la fonction membre `afficheruneTension` affiche directement `uneTension`. Ensuite, le programme utilise deux notations différentes pour accéder à `uneTension` :

```
this-> uneTension
(*this). uneTension
```

L'utilisation des parenthèses dans la dernière notation spécifie l'ordre de priorité des opérations; sans les parenthèses, cette commande serait interprétée comme `*(this. uneTension)` et serait retenue comme une erreur de syntaxe par le compilateur.

# 6. Flots et fichiers

En C++, les entrées-sorties sont les opérations qui provoquent la lecture ou l'écriture des données dans le programme. Ces fonctions sont liées à l'écran, aux fichiers, au clavier, par exemple la saisie des données de programme par le clavier. En C++, ces opérations se font en utilisant les flots (*stream*).

Un flot est un canal sur lequel on peut écrire ou lire des données et qui établit une communication avec un périphérique comme le clavier (entrée de données). Plusieurs flots peuvent être reliés au même périphérique. Leur rôle est de rendre le programme indépendant des contraintes liées aux périphériques auxquels ils sont reliés.

Il existe trois types de flots :

- les flots d'entrée : qui permettent de fournir des informations;
- les flots de sortie : qui permettent de recevoir des informations;
- les flots d'entrée et de sortie qui permettent de faire les deux.

Les classes pour la gestion des flots sont nombreuses. Il existe une dizaine de ces classes réparties dans les bibliothèques `fstream`, `strstrea`, `iostream` et `iomanip`. Dans cette section, nous introduisons certaines classes, notamment les classes `istream` (flot d'entrée), `ostream` (flot de sortie) et `iostream` (qui permet les deux).

Avant d'étudier ces classes, nous présentons quatre flots prédéfinis qui permettent les entrées-sorties standards :

- `cin` : est un flot d'entrée de la classe `istream`; il permet de récupérer les saisies de l'utilisateur. Il est relié au clavier. C'est l'équivalent de `scanf` du langage C.

- `cout` : est un flot de sortie de la classe `ostream`; il permet d'afficher du texte. Il est relié à l'écran. C'est l'équivalent de `printf` du langage C.

- `cerr` : est un flot de sortie de la classe `ostream`; il permet d'afficher des messages d'erreur du programme. En l'utilisant, tout objet écrit dans le flot est immédiatement écrit à l'écran.

- `clog` : est un flot de sortie de la classe `ostream`; il travaille de la même façon que `cerr` à la différence qu'ici tout objet écrit dans le flot est écrit par paquet à l'écran.

On manipule ces flots en utilisant des opérateurs d'insertion `<<` et d'extraction `>>`. Ainsi, pour écrire un texte à l'écran, on envoie les données avec un flot à l'aide de l'opérateur d'insertion `<<`. On peut cumuler plusieurs envoie de données. Pour lire un texte saisi au clavier, on consulte le flot `cin` et l'opérateur d'extraction `>>`.

L'exemple suivant présente une utilisation des flots de saisie et de lecture.

```
#include<iostream> // sans le .h si Linux
using namespace std;
void main()
{
int Age ; /*déclaration de vos variables*/
cout << "Bonjour \n" << " donnez votre âge : " ;
cin >> Age ;
cout << "Votre âge est de " << Age << " ans\n" ;
}
```

## 6.1 L'étude de quelques classes

Les flots sont gérés par un ensemble de classes. Nous vous en présentons quelques-unes.

### *6.1.1 La classe* `ios`

La classe `ios` représente la classe de base de toutes les classes qui gèrent les flots d'entrée et de sortie. Toutes les classes de flots héritent de cette classe.

### *6.1.2 La classe* `ostream`

La classe `ostream` permet d'effectuer les écritures de flot sur un fichier en sortie. Ces sorties peuvent être formatées ou non formatées. Ce flot surcharge l'opérateur `<<` pour tous les types prédéfinis du langage C++. Pour introduire de nouveaux types, il faut la surcharger.

En plus de l'opérateur `<<`, voici deux autres méthodes de la classe `ostream` :

- `put (char c)` insère un caractère dans le flot.
- `write(char *pc, int i)` insère une chaîne de caractères dans le flot.

### *6.1.3 La classe* `istream`

La classe `istream` est dédiée aux entrées formatées ou non formatées. L'objet `cin` est un objet de la classe `istream` défini par défaut qui surcharge l'opérateur `>>` pour tous les types de base de C++. Le principe reste le même que celui pour le flot `cout`.

En plus de l'opérateur `>>`, voici d'autres méthodes de la classe `istream` :

- `get(char c)` lit un caractère dans le flot.
- `getline(char *pc, int i, char c)` lit une chaîne de `(i-1)` caractères dans le flot à moins que le caractère délimiteur `c` ne soit rencontré avant la lecture des `(i-1)`$^{e}$ caractère.
- `gcount()` compte le nombre de caractères après l'appel de la méthode `getline`.
- `read(char *ptr, int taille)` lit la taille de caractères sur le flot et les range à partir de l'adresse `ptr`.

### *6.1.4 La classe* `iostream`

La classe `iostream` permet de réaliser des entrées sorties « conversationnelles » (écran-clavier).

## 6.2 Le statut d'erreur d'un flot

Chaque flot comprend des bits interprétés comme un entier, formant ce qu'on appelle le statut d'erreur du flot.

Quatre valeurs sont définies dans la classe `ios` :

- `eofbit` renvoie 1 en cas de fin de fichier (le flot n'a plus de caractères disponibles).
- `failbit` renvoie 1 si la prochaine opération sur le flot ne pourra pas aboutir.
- `badbit` renvoie 1 si le flot est dans un état irrécupérable.
- `goodbit` vaut 0 s'il n'y a pas d'erreur sur le flot.

Aucune autre opération n'est possible lorsqu'un flot est dans un état d'erreur. Il faut le corriger en remettant le bit d'erreur à zéro à l'aide de la fonction `clear ()`.

Pour accéder aux bits d'erreur, on peut utiliser les cinq fonctions suivantes :

- `eof()` renvoie la valeur de `eofbit`.
- `bad()` renvoie la valeur de `badbit`.
- `fail()` renvoie la valeur de `failbit`.
- `good()` renvoie 1 si aucun bit du statut d'erreur n'est activé sinon il envoie une valeur nulle.
- `rdstate()` renvoie le statut d'erreur du flot.
- `clear()` remet à zéro l'indicateur d'erreur du flux.

Il est aussi possible d'utiliser les mécanismes d'exception pour la remontée des erreurs en C++.

## 6.3 Action sur le statut de formatage

Lorsque vous utilisez la commande `printf` en programmation C, vous spécifiez le type de la variable que vous voulez afficher. On peut en faire autant avec C++. Le mot formatage veut dire réglementer, contrôler le contenu des suites de caractères que vous voulez afficher. Par exemple, afficher des entiers selon la base que vous désirez (octal, hexadécimal, etc.) ou bien contrôler les nombres à virgule flottante.

Pour formater un flot, on peut utiliser, soit des `<<` manipulateurs non paramétriques, soit des fonctions membres.

- Les manipulateurs non paramétriques s'emploient sous la forme `flot << manipulateur ou flot >> manipulateur`.

- Les manipulateurs paramétriques sont comme des fonctions qui s'emploient sous la forme :
    - `istream & manipulateur (argument)` ou
    - `ostream & manipulateur (argument).`

**Tableau 5.6.3-1** Les manipulateurs non paramétriques

| **Manipulateurs non paramétriques** | **Description** |
|---|---|
| `dec` | Active le bit de conversion base 10 |
| `hex` | Active le bit de conversion base 16 |
| `oct` | Active le bit de conversion base 8 |
| `endl` | Provoque un retour chariot |
| `ends` | Provoque une fin de chaîne |
| `flush` | Vide le flot |
| `ws` | Ignorer les espaces dans le flot |

**Tableau 5.6.3-2** Les manipulateurs paramétriques

| **Manipulateurs paramétriques** | **Description** |
|---|---|
| `setbase (int)` | Définit la base de conversion |
| `setprecision (int)` | Précise le nombre de décimal flottant dans le flot |
| `setw (int)` | Précise la largeur de la chaîne envoyée sur le flot (nombres de caractères affichés) |

## 6.4 Les fichiers

C++ possède des classes pour gérer l'échange de données entre les fichiers sur disques et le programme. Ces classes appartiennent à la bibliothèque `fstream` et possèdent certaines propriétés comme la longueur, le début du fichier, la fin du fichier, le mode d'ouverture du fichier, etc.

- `ofstream` dérive de `ostream` pour la gestion de fichier en écriture. Elle permet de créer un flot de sortie associé à un fichier.

Exemple :

```
ofstream flot (char * nomfich, mode_ouverture)
```

- `ifstream` dérive de `istream` pour la gestion de fichiers en lecture. Elle permet de créer un flot d'entrée associé à un fichier.

Exemple :

```
ifstream flot (char * nomfich, mode_ouverture)
```

- `fstream` pour gérer les fichiers en écriture et en lecture.
- La fonction `close` permet de fermer un fichier.

**Tableau 5.6.4** Les modes d'ouverture de fichier

| **Bit de mode d'ouverture** | **Action** |
|---|---|
| `ios::in` | Ouverture en lecture (obligatoire pour la classe `ifstream`) |
| `ios::out` | Ouverture en écriture (obligatoire pour la classe `ofstream`) |
| `ios::app` | Ouverture pour ajouter des données (écriture en fin de fichier) |
| `ios::ate` | Se place en fin de fichier après ouverture |
| `ios::trunc` | Si le fichier existe, son contenu est perdu |
| `ios::binary` | Le fichier est ouvert en mode *binaire* |

Les exemples suivants présentent respectivement un programme pour saisir des données (nom, numéro de téléphone de 10 personnes) dans un fichier (appelé contact) et un programme pour afficher le contenu du fichier contact en sautant une ligne entre chaque personne.

Exemple 1 :

```
# include <fstream.h>
# include <iostream>
using namespace std;
main ( )
{
// Déclaration de variables permettant du programme
string tel;
string nom;
ofstream fichrep ("contact.txt"); // ouverture en
// écriture

for(int i=0; i<10; i++)
{
      cout << "\nPersonne "<< i+1 << ":\n"
cout << " donnez le nom? ";
cin >> nom;
fichrep <<nom << " ";
cout << "\nTelephone?";
cin >> tel;
fichrep << tel << "\n";
}
fichrep.close();
}
```

Exemple 2 :

```
# include <fstream.h>
// # include <conio.h> // nécessaire si Windows
# include <cstring.h>

main( )
{
string nom;
string tel;
ifstream fichrep ("repertoire.txt"); // ouvrir le
                                // fichier en lecture
fichrep >> nom >> tel;
while (! fichrep.eof( )) // tant qu'on est pas arrivé
                         // à la fin du fichier…
{
  cout << nom << " \t"<< tel << "\n"; // … on affiche
  fichrep >> nom >> tel; // on lit les données
}

fichrep.close( ); // Fermeture du fichier
// getch( );non nécessaire en Linux
```

```
}
```

# 7. Piles et arbres en C++

Les listes, les piles et les arbres sont largement utilisés en programmation pour manipuler et gérer les structures de données. Dans cette partie, nous étudions quelques cas d'utilisation de ces outils utilisés en programmation C++.

## 7.1 Les listes

Une liste chaînée est une collection de nœuds reliés par des pointeurs de lien. On accède à une liste chaînée par l'intermédiaire d'un pointeur sur le premier nœud de la liste. Les nœuds suivants sont accessibles par le membre pointeur de lien stocké dans chacun des nœuds. Le premier élément de la liste s'appelle tête de liste et le dernier élément de la liste s'appelle queue de liste.

Il existe trois types de listes de base qui sont :

- les listes simplement chaînées;
- les listes doublement chaînées;
- les arborescences.

Dans une liste simplement chaînée, chaque nœud pointe vers le suivant, jamais vers l'arrière. Pour retrouver un nœud, on commence par le début et l'on passe de nœud en nœud. Une liste doublement chaînée permet d'avancer ou de reculer dans la chaîne. Quant à l'arborescence, il s'agit d'une structure complexe construite à partir de nœuds, chacun d'eux pointant dans deux directions ou plus.

Les nœuds internes assureront le suivi des données contenues dans la liste. Vous pouvez enregistrer n'importe quel type de données dans la liste étant donné que ce ne sont pas elles qui seront reliées, mais les nœuds qui contiennent les données.

### *7.1.1 Les composants d'une liste chaînée*

Les listes chaînées sont composées de nœuds. La classe du nœud sera elle-même abstraite. Nous utilisons trois sous-types : un nœud de tête qui est chargé de gérer la tête de la liste, un nœud de queue et zéro nœud interne ou plus.

L’exemple suivant présente une liste chaînée avec différentes fonctions de manipulation de ses nœuds.

```
#include <iostream>
using namespace std;
enum { kPlusPetit, kPlusGrand, kIdentique};
class Donnees
    {
      public:
        Donnees(int val): maValeur(val){}
        ~Donnees(){}
        int Compare(const Donnees &);
        void Affiche() { cout << maValeur << endl; }
      private:
        int maValeur;
    };

// La classe ci-dessous Donnees permet d’insérer dans la liste
// chaînées. Et cette classe
// possède deux méthodes. Une méthode qui affiche la valeur :
//  Affiche  (affiche  la  valeur)  et  compare  qui  renvoie  la
position // relative

int Donnees::Compare(const Donnees & AutresDonnees)
    {
       if (maValeur < AutresDonnees.maValeur)
          return kPlusPetit;
       if (maValeur > AutresDonnees.maValeur)
          return kPlusGrand;
       else
          return kIdentique;
    }
//  Compare  permet  de  décider  de  l'endroit  dans  la  liste  où
placer // un objet particulier.

// Déclarations anticipées
    class Noeud;
    class NoeudTete;
    class NoeudQueue;
    class NoeudInterne;

    // TDA représentant l'objet noeud dans la liste
    // Chaque classe dérivée doit surcharger Insere et Affiche
    class Noeud
    {
      public:
        Noeud(){}
        virtual ~Noeud(){}
```

```
        virtual Noeud * Insere(Donnees * lesDonnees)=0;
        virtual void Affiche() = 0;
      private:
    };

    // Voici le noeud contenant l'objet réel
    // Ici, l'objet est de type Donnees
    // Nous verrons comment le rendre plus général
    // lorsque nous traiterons des modèles
    class NoeudInterne: public Noeud
    {
      public:
        NoeudInterne(Donnees * lesDonnees, Noeud * suivant);
        ~NoeudInterne(){ delete leSuivant; delete mesDonnees; }
        virtual Noeud * Insere(Donnees * lesDonnees);
        // Déléguer !
        virtual void Affiche() { mesDonnees->Affiche();
                                 leSuivant->Affiche(); }

      private:
        Donnees * mesDonnees;  // Les données
        Noeud * leSuivant;    // Pointe vers le prochain noeud
    };

    // Le constructeur se contente d'initialiser
    NoeudInterne::NoeudInterne(Donnees  *  lesDonnees,  Noeud  *
suivant):
      mesDonnees(lesDonnees),leSuivant(suivant)
    {
    }

    // L'essentiel de la liste
    // Lorsque vous placez un nouvel objet dans la liste
    // il est passé au noeud, qui décide
    // où il va et l'insère dans la liste
    Noeud * NoeudInterne::Insere(Donnees * lesDonnees)
    {

       // Le nouveau est-il plus grand ou plus petit que moi ?
       int resultat = mesDonnees->Compare(*lesDonnees);


       switch(resultat)
       {
       // Par convention, s'il est pareil que moi, il passe en
1er
       case kIdentique:
       case kPlusGrand: // Les nouvelles données viennent avant
```

```
moi
          {
             NoeudInterne * donneesNoeud =
                   new NoeudInterne(lesDonnees, this);
             return donneesNoeud;
          }

       // Il est plus grand que moi, le passer au noeud
       // suivant et le laisser gérer.
       case kPlusPetit:
          leSuivant = leSuivant->Insere(lesDonnees);
          return this;
       }
       return this;
    }

    // Le noeud de queue n'est qu'une sentinelle

    class NoeudQueue : public Noeud
    {
      public:
        NoeudQueue(){}
        ~NoeudQueue(){}
        virtual Noeud * Insere(Donnees * lesDonnees);
        virtual void Affiche() { }

      private:

    };

    //  Si  les  données  viennent  à  moi,  elles  doivent  être
insérées
    // avant moi car je suis la queue et RIEN ne vient après
    Noeud * NoeudQueue::Insere(Donnees * lesDonnees)
    {
       NoeudInterne * donneesNoeud =
              new NoeudInterne(lesDonnees, this);
       return donneesNoeud;
    }

    // Le noeud de tête n'a pas de données, il se contente de
    // pointer vers le début de la liste

    class NoeudTete : public Noeud
    {
      public:
        NoeudTete();
        ~NoeudTete() { delete leSuivant; }
```

```
    virtual Noeud * Insere(Donnees * lesDonnees);
    virtual void Affiche() { leSuivant->Affiche(); }
  private:
    Noeud * leSuivant;
};

// Dès sa création, la tête crée la queue
NoeudTete::NoeudTete()
{
   leSuivant = new NoeudQueue;
}

// Rien ne vient avant la tête,
// donc passer les données au noeud suivant
Noeud * NoeudTete::Insere(Donnees * lesDonnees)
{
   leSuivant = leSuivant->Insere(lesDonnees);
   return this;
}

// J'ai tout le mérite et ne fais rien
class ListeChainee
{
  public:
    ListeChainee();
    ~ListeChainee() { delete maTete; }
    void Insere(Donnees * lesDonnees);
    void AfficheTout() { maTete->Affiche(); }
  private:
    NoeudTete * maTete;
};

// À la naissance, je crée le noeud de tête
// Celui-ci crée le noeud de queue
// Une liste vide pointe donc vers la tête qui
// pointe vers la queue et ne contient rien
ListeChainee::ListeChainee()
{
   maTete = new NoeudTete;
}

// Déléguer, déléguer, déléguer
void ListeChainee::Insere(Donnees * pDonnees)
{
   maTete->Insere(pDonnees);
}

// Programme de test
```

```
int main()
{
   Donnees * pDonnees;
   int val;
   ListeChainee ll;

   // Demander à l'utilisateur de produire des valeurs
   // les placer dans la liste
   for (;;)
   {
      cout << "Insérer une valeur (0 pour arrêter) : ";
      cin >> val;
      if (val == 0)
         break;
      pDonnees = new Donnees(val);
      ll.Insere(pDonnees);
   }

   // Parcourir la liste et afficher les données
   ll.AfficheTout();
   return 0;  // ll est hors de portée, donc détruit !
}
```

Lorsqu'on étudie de plus près le code précédent, la première chose à noter est la ligne `enum { kPlusPetit, kPlusGrand, kIdentique}` qui fournit trois valeurs constantes : `kPlusPetit, kPlusGrand, kIdentique`.Chaque objet qui pourrait être contenu dans cette liste chaînée doit fournir une méthode `Compare()`. Ces constantes seront le résultat renvoyé par la méthode `compare()`.

Cet exemple commence par la création d'une classe `donnees` qui comprend deux méthodes dont la méthode `compare` et une méthode `affiche` qui affiche la valeur de l'objet `donnees`.

Lorsque la classe `ListeChainee` est créée, on appelle le constructeur qui permet d'allouer un objet de type `NoeudTete` et d'affecter l'adresse de cet objet au pointeur tête de la liste chaînée ainsi déclarée. Cette affectation appelle le constructeur `NoeudTete` qui, à son tour, alloue un `NoeudQueue` et affecte son adresse au pointeur `lesuivant` du noeud de tête. La création du noeud de queue appelle le constructeur `NoeudQueue` qui est déclaré `inline` et qui ne fait rien.

Le programme principal commence par une boucle sans fin *(for (,,)).* L'utilisateur doit alors entrer des valeurs qui seront ajoutées à la liste chaînée puis 0 lorsqu'il termine. Si

la valeur saisie est 0, alors le programme interrompt la boucle, sinon un objet `donnees` est crée et inserrer dans la liste.

Par la méthode `insere`, la liste chaînée délègue la responsabilité de l'insertion de l'objet à son noeud de tête.

Au cours du processus, les données ont été insérées dans la liste. La liste les a passées à sa tête. La tête les a passées à l'élément vers lequel elle pointe. Dans ce cas, la tête pointait vers la queue. Un nouveau nœud interne est immédiatement créé et initialisé pour qu'il pointe vers la queue. La queue a renvoyé l'adresse du nouveau noeud à la tête, qui a réaffecté son pointeur `lesuivant` de sorte qu'il pointe vers le nouveau noeud.

## 7.2 Les piles

Une pile est une structure de données abstraite *dynamique* contenant des éléments homogènes (de type non précisé). Il permet de stocker des valeurs avec une stratégie LIFO (). Des méthodes permettent de gérer la pile :

- `void ajouter_au_sommet()` pour ajouter une valeur à la pile.
- `int prendre_au_sommet ()` pour relire la dernière valeur ajoutée.
- d'enlever la dernière valeur ajoutée.
- de tester si la pile est vide.

Dans une structure pile, on ne « connaît » que le dernier élément empilé, c'est-à-dire le sommet de la pile.

Dans le prochain exemple, considérons un programme qui ajoute les *valeurs (-4, -3, -2 et -1)* dans une pile.

La classe `Pile` ci-dessous fait appel à trois méthodes :

- un constructeur de la classe;
- une méthode `ajouter_au_sommet` qui permet d'ajouter un élément dans la pile;
- une méthode `prendre_au_sommet` qui permet de lire un élément en tête de la pile.

Quant à la fonction `abort()`, elle permet d'interrompre brutalement un programme en signalant une erreur grave.

```
class Pile {
int taille, *contenu, nb;
public:
Pile(int taille_max);
~Pile();
virtual void ajouter_au_sommet(int k);
virtual int prendre_au_sommet();
    };

// Cette méthode ci-dessous qui est le constructeur  permet
// d'initialiser la pile.

      Pile::Pile (int taille_max) {
taille = taille_max;
contenu = new int[taille];
nb = 0;
             }

      Pile::~Pile()
         {
delete[] contenu;
         }

      void Pile::ajouter_au_sommet(int k)

 {
if(nb < taille) { contenu[nb] = k; nb++; }
else abort();
 }

int Pile::prendre_au_sommet()
        {
if(nb > 0) { nb--; return contenu[nb]; }
else abort();
 }

int main(int argc, char **argv)

{
Pile pile(10);
pile.ajouter_au_sommet(-1);
pile.ajouter_au_sommet(-2);
pile.ajouter_au_sommet(-3);
pile.ajouter_au_sommet(-4);
cout << pile.prendre_au_sommet() << endl;
```

```
cout << pile.prendre_au_sommet() << endl;
cout << pile.prendre_au_sommet() << endl;
cout << pile.prendre_au_sommet() << endl;
 return 0;
 }
Résultat :

-4
-3
-2
-1
```

Résultat :

```
-4
-3
-2
-1
```

## 7.3 Les arbres

Un arbre est une structure constituée d'un ensemble d'éléments appelés nœuds. Les successeurs immédiats d'un élément sont les fils, alors que l'ensemble des éléments auxquels on peut accéder en suivant les branches issues d'un élément donné constitue sa descendance. L'élément dont un élément est le fils est son père (ou son parent), alors que deux éléments ayant le même père seront dits frères. Un élément sans descendance est une feuille de l'arbre.

Voici la règle de construction de l'arbre :

- Tout arbre a un et un seul nœud sans parent : c'est le nœud racine de l'arbre.
- Chaque nœud a au plus deux nœuds fils.
- Chaque nœud qui n'est pas la racine a exactement un nœud parent.
- Un nœud qui n'a pas de nœuds fils est appelé une feuille.

Enfin, on appelle profondeur de l'arbre, le nombre de branches allant de la racine au nœud le plus bas de l'arbre (le nœud le plus loin de la racine, c'est-à-dire le nœud ayant le plus d'ancêtres).

Comme pour les listes, un arbre est représenté par une structure. Il est implémenté sous la forme de plusieurs nœuds. Chaque nœud connaît ses successeurs et contient une valeur.

```
struct Noeud
{
   int valeur;
   Noeud*fils_droite;
   Noeud*fils_gauche;
};
Noeud* arbre = NULL;
```

Nous allons maintenant présenter les fonctions qui vont nous permettre d'exploiter un arbre.

La fonction `Placer` permet de placer un nœud dans l'arbre ainsi déclaré :

```
void Placer(Noeud* noeud)
{
      Noeud* courant = arbre;
      Noeud* precedent = NULL;

      // Si l'arbre est vide, ça va très vite :
      if(arbre == NULL)
      {
            arbre = noeud;
            return;
      }

      // On se fraye un chemin jusqu'a une place vide
      while(courant)
      {
            precedent = courant;
            if(noeud->valeur < courant->valeur)
                  courant = courant->fils_gauche;
            else
                  courant = courant->fils_droite;
      }

      // si on trouve une place libre, et
      // precedent pointe vers le parent du
      // noeud à replacer.
      if(noeud->valeur < precedent->valeur)
            precedent->fils_gauche = noeud;
      else
            precedent->fils_droite = noeud;
}
```

La fonction `Ajouter` permet l'ajout d'un élément à un arbre. Cette fonction crée un nœud et fait appel à la fonction `Placer` afin de mettre le nœud créé à la place convenable.

```
void Ajouter(int valeur)
{
	// Création de notre nouveau noeud en mémoire
	Noeud* nouveau = new Noeud;
	nouveau->valeur = valeur;
	nouveau->fils_droite = NULL;
	nouveau->fils_gauche = NULL;

	// Puis on laisse Placer() le mettre à la bonne place
	Placer(nouveau);
}
```

La fonction `Rechercher` permet de chercher l'emplacement d'un élément dans l'arbre.

```
Noeud* Rechercher(int valeur)
{
	Noeud* courant = arbre;

	while(courant)
	{
		if(valeur == courant->valeur)
			return courant;
		    else if(valeur < courant->valeur)
			courant = courant->fils_gauche;
		else
			courant = courant->fils_droite;
	}

	return NULL;
}
```

La fonction `Supprimer` permet de supprimer un nœud de l'arbre. Cette fonction fait appelle aussi à la fonction `Placer` qui permet de replacer un nœud à la place du nœud supprimé dans l'arbre.

```
void Supprimer(Noeud* noeud)
{
	Noeud* droite = noeud->fils_droite;
	Noeud* gauche = noeud->fils_gauche;
	Noeud* courant = arbre;

	// Cas délicat : si on supprime la racine?
	if(noeud == arbre)
	{
		arbre = droite;
		if(gauche) Placer(gauche);
		delete noeud;
```

```
            return;
      }

      while(courant)
      {
            if(courant->fils_droite == noeud
                  || courant->fils_gauche == noeud)
                  break;

            if(noeud->valeur >= courant->valeur)
                  courant = courant->fils_droite;
            else
                  courant = courant->fils_gauche;
      }

      // Courant pointe maintenant vers le noeud précédent le
      // noeud à supprimer.
      if(courant->fils_droite == noeud)
            courant->fils_droite = droite;
      else
            courant->fils_gauche = droite;

      // Et puis on replace l'autre fils du noeud disparu

      if(gauche) Placer(gauche);

      // Enfin, on libère l'objet noeud de ses obligations
      delete noeud;
}
```

La fonction `Afficher` permet l'affichage des éléments de l'arbre. Elle est structurée comme suit :

```
void Afficher(Noeud* racine)
{
      if(racine->fils_gauche) Afficher(racine->fils_gauche);
      cout << racine->valeur << endl;
      if(racine->fils_droite) Afficher(racine->fils_droite);
}
```

Toutes ces fonctions peuvent être appelées dans la fonction `main` comme suit :

```
int main(void)
{
      Ajouter(10);
      Ajouter(4);
      Ajouter(15);
      Ajouter(2);
```

```
	Ajouter(16);
	Ajouter(1);
	Ajouter(9);
	Ajouter(14);

	Afficher(arbre);

	Noeud* n = Rechercher(4);
	Supprimer(n);

	Afficher(arbre);

	return 0;
}
```

Résultat du programme :

```
1
2
4
9
10
14
15
16
1
2
9
10
14
15
16
```

# 8. Bibliothèque STL

La bibliothèque STL (*Standard Template Library*) est certainement l'un des atouts très important du langage C++. Cette bibliothèque fournit un ensemble de composantes C++ bien structurées qui fonctionnent de façon cohérente et peuvent aussi être adaptées facilement.

En effet, il est possible d'utiliser les structures de données proposées par STL avec des algorithmes personnels, les algorithmes de la bibliothèque avec des structures de données personnelles, ou d'utiliser toutes les composantes STL. Lors de sa conception,

l'accent a été mis sur l'efficacité et sur l'optimisation des composantes, ce qui en fait un outil très puissant.

Voyons les généralités liées à STL. La bibliothèque STL contient cinq types de composantes : des *conteneurs*, des *itératifs*, des *algorithmes*, des *objets fonctions* et des *adaptateurs*. Nous nous intéressons dans cette partie aux deux premières composantes.

## 8.1 Les conteneurs

Les conteneurs (*containers*) sont des objets qui permettent de stocker d'autres objets : les listes, les tableaux, les ensembles, etc. Ils sont décrits par des classes génériques représentant les structures de données logiques les plus couramment utilisées. Ces classes sont dotées de méthodes permettant de créer, de copier, de détruire ces conteneurs, d'y insérer, de rechercher ou de supprimer des éléments. La gestion de la mémoire, c'est-à-dire l'allocation et la libération de la mémoire, est contrôlée directement par les conteneurs, ce qui facilite leur utilisation.

L'exemple suivant illustre l'utilisation de conteneurs.

```
// Exemple d'utilisation des conteneurs
#include <iostream>   //include <iostream.h> si windows
//#include <conio.h> // pas nécessaire si Linux
using namespace std;
#include <vector.h>
#include <list.h>
int main ()
{
vector<int> tableauEntiers; // Crée un tableau d'entiers vide
list<int> listeEntiers; // Crée une liste d'entiers vide
int unEntier;
// Saisie des entiers
cout << "Saisir le prochain entier (-1 pour finir) : ";
cin >> unEntier;
while (unEntier != -1)
{
      tableauEntiers.push_back(unEntier);
listeEntiers.push_back (unEntier);
cout << "Saisir le prochain entier (-1 pour finir) : ";
cin >> unEntier;
// Nombre d'éléments des conteneurs
cout << "Il a y " << tableauEntiers.size () << " éléments dans
le tableau" << endl;
cout << "Il a y " << listeEntiers.size () << " éléments dans la
```

```
liste" << endl;
// Accès à des éléments
cout << "Premier élément du tableau : "<< tableauEntiers.front
() << endl;
cout << "Premier élément de la liste : "<< listeEntiers.front ()
<< endl;
int milieu = tableauEntiers.size () / 2;
cout << "Élément de milieu de tableau : "<< tableauEntiers
[milieu] << endl;
// getch();pas nécessaire si vous utilisez Linux
}
```

Résultat :

```
Saisir le prochain entier (-1 pour finir) : 4
Saisir le prochain entier (-1 pour finir) : 5
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : 7
Saisir le prochain entier (-1 pour finir) : 6
Saisir le prochain entier (-1 pour finir) : 3
Saisir le prochain entier (-1 pour finir) : -1
Il a y 6 éléments dans le tableau
Il a y 6 éléments dans la liste
Premier élément du tableau : 4
Premier élément de la liste : 4
Élément de milieu de tableau : 7
```

Quelques remarques sur les conteneurs et sur l'exemple présenté ci-dessus.

- Un certain nombre de méthodes sont disponibles sur tous les types de conteneurs, ce qui permet d'homogénéiser leur utilisation. C'est le cas par exemple de la méthode `push_back` qui insère un nouvel élément à la fin d'un conteneur.

- D'autres méthodes ou opérateurs sont disponibles en fonction du type de conteneur utilisé. L'opérateur `[]` est disponible sur les objets de type `vector`, mais pas sur ceux de type `list` : il permet d'accéder directement à un élément.

- L'utilisateur n'a pas à se soucier de l'allocation ou de la libération de la mémoire. C'est vrai lors de l'insertion d'éléments et aussi à la fin du programme : aucune instruction particulière n'est nécessaire pour restituer la mémoire occupée par les conteneurs. À la sortie du bloc dans lequel ils sont définis, leur destructeur se charge de libérer toutes les ressources occupées.

- Les conteneurs peuvent manipuler n'importe quel type de données, à partir du moment où la classe correspondante est dotée d'un certain nombre de méthodes nécessaires à STL (pour une classe `X`) :
  - `X()` : un constructeur par défaut,
  - `X(const X&)` : un constructeur par copie,
  - `operator=(const X&)` : l'opérateur d'affectation,
  - `operator==(const X&)` : l'opérateur d'égalité,
  - `operator<(const X&)` : l'opérateur inférieur (utile uniquement pour les tris).

Les différents types de conteneurs disponibles sont :

- `vector` : conteneur implantant les tableaux, qui autorise les accès directs sur ses éléments. Les opérations de mise à jour (insertion, suppression) sont réalisées en un temps constant à la fin du conteneur, et en un temps linéaire (dépendant du nombre d'éléments) aux autres endroits.
- `list` : conteneur implantant les listes doublement chaînées, dédié à la représentation séquentielle de données. Les opérations de mise à jour sont effectuées en un temps constant à n'importe quel endroit du conteneur.
- `deque` : conteneur similaire au conteneur `vector`, effectuant de plus les opérations de mise à jour en début de conteneur en un temps constant.
- `set` : conteneur implantant les ensembles où les éléments ne peuvent être présents qu'en un seul exemplaire.
- `multiset` : conteneur implantant les ensembles où les éléments peuvent être présents en plusieurs exemplaires.
- `map` : conteneur implantant des ensembles où un type de données appelé *clé* est associé aux éléments à stocker. On ne peut associer qu'une seule valeur à une clé unique. On appelle aussi ce type de conteneur *tableau associatif.*
- `multimap` : conteneur similaire au conteneur `map` supportant l'association de plusieurs valeurs à une clé unique.

- `stack` : conteneur implantant les piles, qui sont des listes spéciales, dites *LIFO* (*last in first out*).

- `queue` : conteneur implantant les files, qui sont des listes spéciales, dites *FIFO* (*first in first out*).

### *8.1.1 Les listes*

Comme nous l'avons vu précédemment, les listes sont des structures de données à accès indirect. Elles sont formées d'un ensemble d'éléments souvent de même type ou de type différent. Dans la programmation en C et en C++, les listes jouent un grand rôle.

L'avantage des listes est qu'elles sont dynamiques, ce qui assure moins de ressources lors des variations de la taille des structures. Un autre avantage est qu'il est facile d'ajouter, de supprimer, d'ordonner les éléments dans n'importe quelle partie de la liste, aussi bien au milieu, au début qu'à la fin de celle-ci.

Une liste chaînée est une suite de couples formés d'un élément et de l'adresse (référence) vers l'élément suivant dans la liste.

La classe `template list` est certainement l'une des plus importantes, car, comme son nom l'indique, elle implémente une structure de liste chaînée d'éléments, ce qui est sans doute l'une des structures les plus utilisées en informatique. Cette structure est particulièrement adaptée pour les algorithmes qui parcourent les données dans un ordre séquentiel.

Les listes offrent donc la plus grande souplesse possible sur les opérations d'insertion et de suppression des éléments, en contrepartie de quoi les accès sont restreints à un accès séquentiel.

Ici, l'insertion et la suppression des éléments en tête et en queue de liste peuvent se faire sans recherche, ce sont évidemment les opérations les plus courantes. Par conséquent, la classe `template list` propose des méthodes spécifiques permettant de manipuler les éléments qui se trouvent en ces positions.

L'insertion d'un élément peut donc être réalisée respectivement en tête et en queue de liste avec les méthodes `push_front` et `push_back`. Inversement, la suppression des éléments situés en ces emplacements est réalisée avec les méthodes `pop_front` et `pop_back`.

Toutes ces méthodes ne renvoient aucune valeur, aussi l'accès aux deux éléments situés en tête et en queue de liste peut-il être réalisé respectivement par l'intermédiaire des

accesseurs `front` et `back`, qui renvoient tous deux une référence (éventuellement constante si la liste est elle-même constante) sur ces éléments.

```
#include  <iostream>
#include  <list>
using  namespace  std;
typedef  list <int>  li;
int  main (void)
{
li l1;
l1.push_back(2);
l1.push_back(5);
cout << "Tête : " << l1.front() << endl;
cout << "Queue : " << l1.back() << endl;
l1.push_front(7);
cout << "Tête : " << l1.front() << endl;
cout << "Queue : " << l1.back() << endl;
l1.pop_back();
cout << "Tête : " << l1.front() << endl;
cout << "Queue : " << l1.back() << endl;
return 0;
}
```

**Tableau 5.8.1.1** Méthodes spécifiques des listes

| **Méthode** | **Fonction** |
|---|---|
| **remove**(const T &) | Permet d'éliminer tous les éléments d'une liste dont la valeur est égale à la valeur passée en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste. |
| **remove_if**(Predicat) | Permet d'éliminer tous les éléments d'une liste qui vérifient le prédicat unaire passé en paramètre. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé. La complexité de cette méthode est linéaire en fonction du nombre d'éléments de la liste. |
| **unique**(Predicat) | Permet d'éliminer tous les éléments pour lesquels le prédicat binaire passé en paramètre est vérifié avec |

| Méthode | Fonction |
| --- | --- |
| | comme valeur l'élément courant et son prédécesseur.<br><br>Cette méthode permet d'éliminer les doublons successifs dans une liste selon un critère défini par le prédicat. Par souci de simplicité, il existe une surcharge de cette méthode qui ne prend pas de paramètres, et qui utilise un simple test d'égalité pour éliminer les doublons. L'ordre relatif des éléments qui ne sont pas supprimés est inchangé, et le nombre d'applications du prédicat est exactement le nombre d'éléments de la liste moins un si la liste n'est pas vide. |
| **splice**(iterator position, list<T, Allocator> liste, iterator premier, iterateur dernier) | Injecte le contenu de la liste fournie en deuxième paramètre dans la liste courante à partir de la position fournie en premier paramètre. Les éléments injectés sont les éléments de la liste source identifiés par les itérateurs premier et dernier. Ils sont supprimés de la liste source à la volée.<br><br>Cette méthode dispose de deux autres surcharges, l'une ne fournissant pas d'itérateur de dernier élément et qui insère uniquement le premier élément, et l'autre ne fournissant aucun itérateur pour référencer les éléments à injecter.<br><br>Cette dernière surcharge ne prend donc en paramètre que la position à laquelle les<br><br>éléments doivent être insérés et la liste source elle-même. Dans ce cas, la totalité de la liste source est insérée en cet emplacement. Généralement, la complexité des méthodes splice est proportionnelle au nombre d'éléments injectés, sauf dans le cas de la dernière surcharge, qui s'exécute |

| Méthode | Fonction |
|---|---|
| | avec une complexité constante. |
| **sort**(Predicat) | Trie les éléments de la liste dans l'ordre défini par le prédicat binaire de comparaison passé en paramètre. Encore une fois, il existe une surcharge de cette méthode qui ne prend pas de paramètre et qui utilise l'opérateur d'infériorité pour comparer les éléments de la liste entre eux. L'ordre relatif des éléments équivalents (c'est-à-dire des éléments pour lesquels le prédicat de comparaison n'a pas pu statuer d'ordre bien défini) est inchangé à l'issue de l'opération de tri.<br><br>On indique souvent cette propriété en disant que cette méthode est stable. La méthode sort s'applique avec une complexité égale à N×ln(N), où N est le nombre d'éléments de la liste. |
| **merge**(list<T,<br>Allocator>,<br>Predicate) | Injecte les éléments de la liste fournie en premier paramètre dans la liste courante en conservant l'ordre défini par le prédicat binaire fourni en deuxième paramètre. Cette méthode suppose que la liste sur laquelle elle s'applique et la liste fournie en paramètre sont déjà triées selon ce prédicat, et garantit que la liste résultante sera toujours triée. La liste fournie en argument est vidée à l'issue de l'opération.<br><br>Il existe également une surcharge de cette méthode qui ne prend pas de second paramètre et qui utilise l'opérateur d'infériorité pour comparer les éléments des deux listes. La complexité de cette méthode est proportionnelle à la somme des tailles des deux listes ainsi fusionnées. |

| Méthode | Fonction |
|---|---|
| **reverse** | Inverse l'ordre des éléments de la liste. Cette méthode s'exécute avec une complexité linéaire en fonction du nombre d'éléments de la liste. |

Nous présentons ci-dessous l'utilisation de différentes `templates` pour manipuler les listes.

```
#include <iostream>
#include <functional>
#include <list>
using namespace std;
typedef list<int> li;

void print(li &l)
{
    li::iterator i = l.begin();
    while (i != l.end())
 {

cout << *i << " ";
++i;
 }
cout << endl;
return ;
}

bool parity_even(int i)
{
return (i & 1) == 0;
}

int main(void)
{
// Construit une liste exemple :
li l;
l.push_back(2);
l.push_back(5);
l.push_back(7);
l.push_back(7);
l.push_back(3);
l.push_back(3);
l.push_back(2);
l.push_back(6);
l.push_back(6);
```

```
l.push_back(6);
l.push_back(3);
l.push_back(4);
cout << "Liste de départ :" << endl;
print(l);
li l1;
// Liste en ordre inverse :
l1 = l;
l1.reverse();
cout << "Liste inverse :" << endl;
print(l1);
// Trie la liste :
l1 = l;
l1.sort();
cout << "Liste triée : " << endl;
print(l1);
// Supprime tous les 3 :
l1 = l;
l1.remove(3);
cout << "Liste sans 3 :" << endl;
print(l1);
// Supprime les doublons :
l1 = l;
l1.unique();
cout << "Liste sans doublon :" << endl;
print(l1);
// Retire tous les nombres pairs :
l1 = l;
l1.remove_if(ptr_fun(&parity_even));
cout << "Liste sans nombre pair :" << endl;
print(l1);
// Injecte une autre liste entre les 7 :
l1 = l;
li::iterator i = l1.begin();
++i; ++i; ++i;
li l2;
l2.push_back(35);
l2.push_back(36);
l2.push_back(37);
l1.splice(i, l2, l2.begin(), l2.end());
cout << "Fusion des deux listes :" << endl;
print(l1);
if (l2.size() == 0)
cout << "l2 est vide" << endl;
return 0;
}
```

*8.1.2 Les piles*

Les piles sont des structures de données qui se comportent, comme leur nom l'indique, comme un empilement d'objets. Elles ne permettent donc d'accéder qu'aux éléments situés en haut de la pile, et la récupération des éléments se fait dans l'ordre inverse de leur empilement. En raison de cette propriété, on les appelle également couramment *LIFO*, acronyme de l'anglais « *last In first Out* » (dernier entré, premier sorti).

La classe adaptatrice définie par la bibliothèque standard C++ pour implémenter les piles est la classe `template stack`. Cette classe utilise deux paramètres `template` : le type des données lui-même et le type d'une classe de séquence implémentant au moins les méthodes `back`, `push_back` et `pop_back`. Il est donc parfaitement possible d'utiliser les conteneurs `list`, `deque` et `vector` pour implémenter une pile à l'aide de cet adaptateur. Par défaut, la classe `stack` utilise une `deque`, et il n'est donc généralement pas nécessaire de spécifier le type du conteneur à utiliser pour réaliser la pile.

L'interface des piles se réduit au strict minimum, puisqu'elles ne permettent de manipuler que leur sommet. La méthode *push* permet d'empiler un élément sur la pile, et la méthode `pop` de l'en retirer. Ces deux méthodes ne renvoient rien, l'accès à l'élément situé au sommet de la pile se fait donc par l'intermédiaire de la méthode `top`.

L'exemple suivant présente les méthodes de manipulation d'une pile.

```
// Utilisation d'une pile
#include <iostream>
#include <stack>
using namespace std;

int main(void)
{
typedef stack<int> si;
// Crée une pile :
si s;
// Empile quelques éléments :
s.push(2);
s.push(5);
s.push(8);
// Affiche les éléments en ordre inverse :
while (!s.empty())
{
cout << s.top() << endl;
s.pop();

}
```

```
return 0;
  }
```

### 8.1.3 Les files

Les files sont des structures de données similaires aux piles, à la différence près que les éléments sont mis les uns à la suite des autres au lieu d'être empilés. Leur comportement est donc celui d'une file d'attente où tout le monde serait honnête (c'est-à-dire que personne ne doublerait les autres). Les derniers entrés sont donc ceux qui sortent également en dernier, d'où leur dénomination de *FIFO* (de l'anglais *first in first out*).

Les files sont implémentées par la classe `template queue`. Cette classe utilise comme paramètre `template` le type des éléments stockés ainsi que le type d'un conteneur de type séquence pour lequel les méthodes `front`, `back`, `push_back` et `pop_front` sont implémentées. En pratique, il est possible d'utiliser les listes.

Les méthodes fournies par les files sont les méthodes `front` et `back`, qui permettent d'accéder respectivement au premier et au dernier élément de la file d'attente, ainsi que les méthodes `push` et `pop`, qui permettent respectivement d'ajouter un élément à la fin de la file et de supprimer l'élément qui se trouve en tête de file.

```
// Utilisation d'une file
#include <iostream>
#include <queue>
using namespace std;
int main(void)
{
typedef queue<int> qi;
// Crée une file :
qi q;
// Ajoute quelques éléments :
q.push(2);
q.push(5);
q.push(8);
// Affiche récupère et affiche les éléments :
while (!q.empty())
{
cout << q.front() << endl;
q.pop();
}
return 0;
}
```

*8.1.4 Les files de priorités*

Enfin, la bibliothèque standard fournit un adaptateur permettant d'implémenter les files de priorités. Les files de priorités ressemblent aux files classiques, mais ne fonctionnent pas de la même manière. En effet, contrairement aux files normales, l'élément qui se trouve en première position n'est pas toujours le premier élément qui a été placé dans la file, mais celui qui dispose de la plus grande valeur. C'est cette propriété qui a donné son nom aux files de priorités, car la priorité d'un élément est ici donnée par sa valeur. Bien entendu, la bibliothèque standard permet à l'utilisateur de définir son propre opérateur de comparaison, afin de lui laisser spécifier l'ordre qu'il veut utiliser pour définir la priorité des éléments.

La classe `template` fournie par la bibliothèque standard pour faciliter l'implémentation des files de priorité est la classe `priority_queue`. Cette classe prend trois paramètres `template` : le type des éléments stockés, le type d'un conteneur de type séquence permettant un accès direct à ses éléments et implémentant les méthodes `front`, `push_back` et `pop_back`, et le type d'un prédicat binaire à utiliser pour la comparaison des priorités des éléments.

Comme les files de priorités se réorganisent à chaque fois qu'un nouvel élément est ajouté en fin de file, et que cet élément ne se retrouve pas forcément en dernière position s'il est de priorité élevée, accéder au dernier élément des files de priorité n'a pas de sens. Il n'existe donc qu'une seule méthode permettant d'accéder à l'élément le plus important de la pile : la méthode top. En revanche, les files de priorité implémentent effectivement les méthodes push et pop, qui permettent respectivement d'ajouter un élément dans la file de priorité et de supprimer l'élément le plus important de cette file.

Exemple :

```
#include <iostream>
#include <queue>
using namespace std;
// Type des données stockées dans la file :
struct A
{
int k; // Priorité
const char *t; // Valeur
A() : k(0), t(0) {}
A(int k, const char *t) : k(k), t(t) {}
};
// Foncteur de comparaison selon les priorités :
```

```
class C
{
public:
bool operator()(const A &a1, const A &a2)
{
return a1.k < a2.k ;
}
};
int main(void)
{
// Construit quelques objets :
A a1(1, "Priorité faible");
A a2(2, "Priorité moyenne 1");
A a3(2, "Priorité moyenne 2");
A a4(3, "Priorité haute 1");
A a5(3, "Priorité haute 2");
// Construit une file de priorité :
priority_queue<A, vector<A>, C> pq;
// Ajoute les éléments :
pq.push(a5);
pq.push(a3);
pq.push(a1);
pq.push(a2);
pq.push(a4);
// Récupère les éléments par ordre de priorité :
while (!pq.empty())
{
cout << pq.top().t << endl;
pq.pop();
}
return 0;
}
```

## 8.2 Les itérateurs

Les *itérateurs* sont une généralisation des pointeurs, ce qui permet au programmeur de travailler avec des conteneurs différents de façon uniforme. Ils permettent de spécifier une position à l'intérieur d'un conteneur. Ils peuvent être incrémentés ou déréférencés (à la manière des pointeurs utilisés avec l'opérateur de déréférencement `*`), et deux *itérateurs* peuvent être comparés. Tous les conteneurs sont dotés d'une méthode `begin` qui renvoie un *itérateur* sur le premier de leurs éléments, et d'une méthode `end` qui renvoie un *itérateur* à l'endroit qui se trouve juste après le dernier de leurs éléments. On ne peut ainsi pas déréférencer l'*itérateur* renvoyé par la méthode `end`. Étudions un exemple d'utilisation des *itérateurs*.

```
#include <iostream>
#include <list>
int main()
 {
   list<int> lesEntiers;
    // Ici, des instructions pour initialiser la
   // liste des entiers
    ...
   // Affichage des éléments contenus dans la liste
    list<int>::iterator it;
      for (it = lesEntiers.begin(); it != lesEntiers.end();
it++)
    cout << *it << endl;
 }
```

Ces *itérateurs* sont dotés de méthodes permettant de les manipuler. Il existe une hiérarchie d'*itérateurs*, qui n'est pas liée à un quelconque héritage.

- Les *itérateurs* d'entrée (*input iterators*) : permettent d'accéder séquentiellement à des sources de données. Cette source peut-être un conteneur, un *flot*.
- Les *itérateurs* de sortie (*output iterators*) : permettent de préciser la destination où seront stockées les données. Cette source peut-être un conteneur, un *flot*.