

Chapitre 1

C++

1.1 Introduction

1.1.1 les concepts de la POO

Il existe différents paradigmes de programmation qui permettent de répondre à différents types de problèmes :

- programmation fonctionnelle (Lisp)
- programmation logique (Prolog)
- programmation par contraintes
- programmation procédurale
- programmation orientée objet (POO)

Du fait de nombreux atouts (modularité, encapsulation, polymorphisme), la programmation orientée objet constitue une technologie adéquate pour répondre aux exigences de la construction d'applications complexes.

On distingue 4 catégories de modes opératoires chez un programmeur :

- mode non structuré (Assembleur, Basic)
- mode structuré (sous-programmes)
- mode modulaire (modules, packages)
- mode objet

La programmation structurée (Pascal, C) est définie par la formule suivante (Wirth) :

$$\text{programme} = \text{algorithmes} + \text{structures de données}$$

en fait, la formule qui semble la plus appropriée est :

$$\text{programme} = \text{algorithmes}(\text{structures de données, paramètres})$$

La P.O.O repose sur le paradigme suivant :

$$\text{programme} = \text{structure de données.algorithmes}(\text{paramètres})$$

On réalise ainsi l'encapsulation de données et on donne naissance au concept d'objet, c'est à dire qu'on ne peut accéder aux données (appelées attributs) que par les algorithmes (appelés méthodes) définis pour chaque structure de données. Un objet (ou classe d'objets) est donc défini par attributs + méthodes. D'autre part, on ne se préoccupe pas des détails d'implémentation.

Le mode objet comprend des concepts qui permettent au programmeur de prendre une certaine distance par rapport au code.

- encapsulation de données
- héritage (ou hiérarchie d'objets)
- polymorphisme

1.1.2 historique

Le langage C++ a été développé par Bjarne Stroustrup [9] travaillant aux Bell Labs de 1980 à 1983. C++ fut commercialisé en 1985 et le livre *The C++ Programming Language* fut édité la

même année. Une standardisation du langage commença en 1990 et fut achevée en 1997 par le comité ANSI.

Le langage C++ est souvent considéré comme un *meilleur C* mais est en fait plus que cela. Il ne s'agit pas d'un langage objet pur car C++ est basé sur C, il en reprend donc les avantages et les inconvénients. C++ est un langage très subtile et donc très difficile à maîtriser mais il permet de passer progressivement de la programmation modulaire à la technologie objet.

1.2 Les différences avec le langage C

1.2.1 Fichiers et compilateur

En C++ les fichiers peuvent porter les extensions suivantes : `.c` ou `.cpp`. Les fichiers entête prennent quant à eux l'extension `.h` ou `.hpp`.

Le compilateur sous licence GNU porte le nom `g++` ou parfois `gpp`.

1.2.2 espace de nommage *namespace*

Pour des projets de grande envergure, des conflits de noms apparaissent pour certains noms de classes ou de variables. Le C++ offre un moyen de remédier à ce problème en limitant la portée des symboles à une zone appelée *espace de nommage* ou *namespace* :

```
namespace <nom> {
    declarations
}
```

Pour accéder à une déclaration de l'espace de nommage on peut utiliser l'opérateur de résolution de portée `::` ou alors utiliser l'instruction :

```
using namespace <nom> ;
```

En particulier `std` est l'espace de nommage réservé à la STL (Standard Template Library) dont nous reparlerons plus tard.

1.2.3 le qualificateur *const*

Le langage C++ introduit le qualificateur `const` qui indique qu'une donnée ne sera pas modifiée. Situé au niveau d'un sous-programme `const` indique qu'aucune donnée ne sera modifiée à l'intérieur du sous-programme. le qualificateur `const` peut également servir à :

- définir des constantes,
- spécifier qu'une méthode ne peut modifier des données membres,

```
const int N=5;
int R;

class A {
    int a;

    public:
        const int f(const int n) const {
            N=1; // interdit !!
            a=2; // interdit
            R=5; // autorise
        }
};
```

1.2.4 le qualificateur *enum*

Le mot clé `enum` permet de déclarer des énumérations ou un nouveau type (entier) associé à une liste de valeurs :

```
enum { zero, un, deux, trois };
enum Jour { lundi=1, mardi, mercredi, jeudi, vendredi, samedi, dimanche };

Jour j=mardi; // j=2;
int m=j;
```

1.2.5 gestion de la mémoire avec *new* et *delete*

Deux nouveaux opérateurs sont introduits pour gérer et simplifier l'allocation mémoire :

```
int *ptr;
int *tab;

ptr = new int;
tab = new int[20];

delete ptr;
delete [] tab; // ou delete tab
```

Ces opérateurs sont notamment utilisés pour allouer/désallouer des objets.

1.2.6 l'opérateur de référence

Un nouveau sens est donné à l'opérateur `&` qui permet de manipuler les pointeurs sans utiliser la syntaxe traditionnelle `->` ou `*` :

```
int n = 1;
int& ref_n = n;
ref_n = 2;
cout << "n = " << n << endl; // n=2

int *p_int = new int;
*p_int = 1;
int& ref_p_int = *p_int;
ref_p_int = 3;
cout << "*p_int = " << *p_int << endl;
```

1.2.7 paramètres par défaut

La possibilité est offerte en C++ de donner une valeur par défaut uniquement aux paramètres les plus à droite dans la définition d'un sous-programme :

```
void fonction(int n, int m=1) {
    cout << n << " " << m << endl;
}

f(1,5);
f(1); // equivalent a f(1,1)
```

1.2.8 gestion des entrées/sorties

La STL introduit des flux (*stream*) sur lesquels on peut réaliser des entrées/sorties. Il existe des flux spécifiques :

- `cin` : flux d'entrée standard (clavier)
- `cout` : flux de sortie standard (écran)
- `cerr` : flux d'erreur standard
- `clog` : flux d'information standard

On manipule à *la Pascal* et non plus comme en C :

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

int n;
cin >> n; // lecture d'un entier
cout << "la valeur de n est " << n << endl;
```

1.2.8.1 flux de sortie

Des manipulateurs de flux sont définis qui permettent de formater l'affichage des données :

- `hex`, `oct`, `dec` : affichage en hexadécimal, octal ou décimal
- `scientific` : utilise la notation en virgule flottante
- `setprecision(int)` fixe le nombre de chiffres significatifs
- `setw(int)` donne la largeur minimale du champ
- `setfill(char)` précise le caractère de remplissage

1.2.8.2 flux d'entrée

Utilisation de la fonction `getline(char *, int max, char delim);`

1.2.9 Le typage

Afin de remédier aux déficiences du langage C concernant le typage, de nouveaux opérateurs sont introduits :

- `static_cast` pour la conversion d'un type vers un autre type
- `const_cast` pour supprimer le caractère constant d'une variable
- `dynamic_cast` pour les hierarchies d'objets
- `reinterpret_cast` pour conversion entre 2 types différents (int vers pointeur par exemple)

```
int i=1;
double d = static_cast<double>(i);

int f(int& n) {
    return n+1;
}

int g(const int& n) {
    return f(const_cast<int&>(n));
}
```

1.3 La classe ou comment réaliser l'encapsulation

1.3.1 déclaration

Le concept de classe (ou d'objet) permet d'encapsuler les données et les sous-programmes associés à une structure de données, c'est à dire que l'on restreint l'accès aux données et aux

sous-programmes afin d'éviter des modifications non voulues par des sous-programmes externes. Pour cela on introduit des environnements de protection :

- *public* : accès libre, les données peuvent être modifiées
- *private* : accès autorisé uniquement aux sous-programmes de la classe (par défaut)
- *protected* : accès autorisé uniquement aux sous-programmes d'une hiérarchie de classes.

```
class Nom-de-la-Classe {
    public:
        liste des déclarations publiques
    protected :
        liste des déclarations protégées
    private:
        liste des déclarations privées
};
```

1.3.2 constructeur

Le constructeur a pour but d'initialiser l'objet, il est déclaré en reprenant le nom de la classe et doit être déclaré dans la section *public* :

```
// fichier object.h

enum { UNKNOWN, INTEGER, FLOAT, LIST, ARRAY };

class Object {
    protected:
        int objectType;
    public:
        Object();
        Object(int type);
};
```

Le corps des sous-programmes est déclaré dans le fichier `.cpp`

```
// fichier object.cpp

Object::Object() {
    objectType=UNKNOWN;
}

Object::Object(int type) {
    objectType=type;
}
```

On remarquera que les sous-programmes sont précédés du nom de l'objet auquel ils se rapportent : `Object::`.

Une autre solution consiste à définir les sous-programmes au niveau de la définition de l'objet, c'est à dire dans le bloc `class`.

```

// fichier object.h

class Object {
protected:
    int objectType;
public:
    Object() { objectType=UNKNOWN; }
    Object(int type) { objectType = type; }
};

```

dans ce cas on n'a plus d'appel à un sous-programme mais le code du sous-programme est directement inséré à l'endroit de l'appel.

1.3.3 destructeur

Il a pour but de supprimer l'espace éventuellement alloué par l'objet.

```

class Vector {
private:
    int size_max;
    int size; // nombre maximal d'elements
    int *tab; // pointeur sur un tableau alloué dynamiquement
public:
    // constructeur par défaut
    Vector() {
        size_max=20;
        size=0;
        tab=NULL;
    }
    // constructeur avec taille du vecteur
    Vector(int s) : size(0) {
        tab=new int[size_max=s];
    }
    // destructeur
    ~Vector() {
        delete [] tab;
    }
    // append value
    void append(int v) {
        if (size<size_max) tab[size++]=v;
    }
};

Vector v1(10);

```

1.3.4 constructeur par recopie

Ce constructeur est utilisé lors de la création d'un nouvel objet qui est initialisé par un objet existant :

```

Object o1;
Object o2(o1);

```

S'il n'existe pas explicitement de constructeur par recopie défini pour une classe, on réalise une recopie des champs de l'objet existant vers le nouvel objet. Ce comportement *par défaut* pose de nombreux problèmes notamment pour les objets de type *conteneur* :

```
Vector v1(10);
Vector v2(v1);
```

Dans ce cas `v1` et `v2` pointent sur le même espace alloué par `v1`. De plus, lorsque l'objet `v1` est supprimé, `v2` pointe sur une zone de mémoire désallouée.

On définit le constructeur par recopie sous la forme suivante :

```
class Vector {
public:
    Vector(Vector& v) {
        // allocation
        size=v.size;
        tab=new int[size];
        // recopie des valeurs
        for (int n=0;n<v.size;++i) tab[i]=v.tab[i];
    }
};
```

1.3.5 objet copie d'un autre objet

Il est possible en C++ d'affecter des objets entre-eux, notamment lorsque l'on a besoin de créer une copie temporaire d'un objet :

```
Vector v1(10); // vecteur de 10 elements
Vector v2;
...
v2=v1;
```

Ce cas de figure ressemble au constructeur par recopie. Pour ne pas rencontrer les mêmes problèmes on doit redéfinir l'opérateur d'affectation pour l'objet concerné :

```
class Vector {
    ...
    Vector& operator=(Vector& v);
};

Vector& Vector::operator=(Vector& v) {
    // empeche l'affectation de l'objet sur lui-meme
    if (this!=&v) {
        // supprimer l'espace alloue precedemment
        if (tab!=NULL) delete tab;
        // allouer un nouveau tableau
        tab=new int[size=v.size];
        // recopier les elements
        for (int i=0;i<size;++i) tab[i]=v.tab[i];
    }
    return *this;
}
```

1.3.6 tableaux d'objets

Lors de l'allocation d'un tableau d'objets, chacun des objets est initialisé avec le constructeur approprié. S'il n'en existe aucun, une erreur est générée.

```

Vector *tab;

// allocation avec initialisation par constructeur par défaut
tab=new Vector[10];

// allocation avec initialisation par constructeur avec taille
tab=new Vector[10](10);

```

1.3.7 attributs statiques

La création d'un nouvel objet engendre la création d'un nouvel espace mémoire. Il en résulte une impossibilité de garder trace, par exemple, du nombre d'objets alloués. Pour palier ce problème, on peut utiliser des champs statiques :

```

// fichier compte.h
class Compteur {
private:
    static int compte;

public:
    Compteur() { ++compte; }
    ~Compteur() { --compte; }
    int getCompte() { return compte; }
};

// fichier compte.cpp
int Compteur::compte=0;

```

Ici, la variable `Compteur::compte` n'est définie qu'une seule fois et est commune à toute instance de la classe.

1.3.8 fonctions et classes amies

1.3.8.1 fonctions amies

Il est parfois nécessaire de faire en sorte qu'une fonction puisse avoir accès aux champs protégés et privés d'une classe. Pour cela il suffit de déclarer au niveau de la définition de la classe, le prototype de la fonction précédé du mot clé `friend`. Les fonctions amies permettent de résoudre des problèmes liés à la définition de fonctions membres qui nécessitent parfois d'accéder aux attributs de deux classes différentes. Un exemple concret est celui de deux classes `Vecteur` et `Matrice` pour lesquelles on désire définir une fonction produit qui réalise le produit d'un `Vecteur` avec une `Matrice` :

- une première solution consisterait à déclarer et implanter cette fonction au niveau de la classe `Vecteur` et de la classe `Matrice`, mais cela va à l'encontre du principe de réutilisabilité du code.
- une deuxième solution consisterait à créer une nouvelle classe `ProdVecteurMatrice` qui hériterait de `Vecteur`.
- une troisième solution consiste à créer une fonction amie, indépendante des deux classes `Vecteur` et `Matrice` mais capable d'accéder rapidement à leurs champs privés et protégés sans passer par des accesseurs.

```

class Vecteur {
    ...
    friend Vecteur *produit(Vecteur *v, Matrice *m);
};

class Matrice {
    ...
    friend Vecteur *produit(Vecteur *v, Matrice *m);
};

Vecteur *produit(Vecteur *v, Matrice *m) { ... }

```

1.3.8.2 classes amies

Une classe peut être amie d'une autre classe, dans ce cas pour déclarer la situation d'amitié il faut que la classe amie ait été définie au préalable.

```

#include "classe_amie.h"

class Une_classe {
    ...
    public:
        class Classe_amie;
};

```

1.3.9 opérateurs

Les classes définissent des objets et dans un langage orienté objet, les objets peuvent être manipulés comme des types simples dans d'autres langages. Il est possible en C++ d'associer un certain nombre d'opérateurs comme les opérateurs arithmétiques ainsi que `new`, `delete`, `[]`, `()`, `->`, `,`.

Dans le cas des opérateurs arithmétiques, deux solutions s'offrent à nous (et il faut bien sûr en choisir une seule) :

- surcharger l'opérateur associé à la classe
- ou utiliser une fonction amie

1.3.9.1 surdéfinition de l'opérateur

```

// surcharge de l'opérateur +
class Integer {
protected:
    int value;
public:
    Integer() { value=0; }
    Integer(int v) { value=v; }

    Integer operator+(const Integer& a);
};

Integer Integer::operator+(const Integer& a) {
    Integer sum(value+a.value);
    return sum;
}

```

1.3.9.2 fonction amie opérateur

```
// utilisation d'une fonction amie
class Integer {
    ...
    friend Integer operator+(Integer& a, Integer& b);
};

Integer operator+(Integer& a, Integer& b) {
    Integer sum(a.value+b.value);
    return sum;
}
```

Si on écrit le code suivant :

```
Integer a(1), b(2), c;

c=a+b;
```

alors le code est évalué

- dans le cas d'une surcharge comme : `c=a.+(b)`
- dans le cas d'une fonction amie : `c=+(a,b)`

1.3.9.3 Les opérateurs d'incrément / décrémentation

On peut également redéfinir les opérateur de pré et post incrément

```
class Integer {
    ...
    Integer& operator++; // pre incrementation
    Integer operator++(int); // post incrementation
};

Integer& Integer::operator++() {
    ++value;
    return *this;
}

Integer Integer::operator++(int) {
    ++(*this);
}
```

1.3.10 utilisation de l'opérateur []

```
class Vector {
    ...
public:
    int& operator[] (int n);
};

int& Vector::operator[] (int n) {
    return tab[n];
}
```

1.3.11 passage des objets en argument

Supposons que nous disposions d'une classe vecteur et que nous définissions une fonction somme de deux vecteurs sous la forme suivante :

```
int somme(Vector v, Vector w) {
    int s=0;
    for (int i = 0; i < v.getSize(); ++i) {
        v[i]=v[i]+w[i];
        s+=v[i];
    }
    return s;
}
```

alors le vecteur v ne sera pas modifié car une copie de v sera réalisée lors de l'appel à la fonction. Il faut ici faire appel à l'une des solutions suivantes :

```
// utilisation d'un pointeur
int somme(Vector *v, Vector *w) {
    int s=0;
    for (int i = 0; i < v->getSize(); ++i) {
        (*v)[i] = (*v)[i] + (*w)[i];
        s += (*v)[i];
    }
    return s;
}

somme(&v1,&v2);
```

```
// utilisation de l'operateur de reference
int somme(Vector &v, Vector &w) {
    int s=0;
    for (int i = 0; i < v.getSize(); ++i) {
        v[i] = v[i] + w[i];
        s += v[i];
    }
    return s;
}

somme(v1,v2);
```

1.4 Héritage, hierarchies de classes

La notion d'héritage permet de définir des hierarchies de classes. Pour rappel :
La définition de l'héritage en C++ s'opère sous la forme suivante :

```
class A {
    ...
};

class B : public A {
    ...
};
```

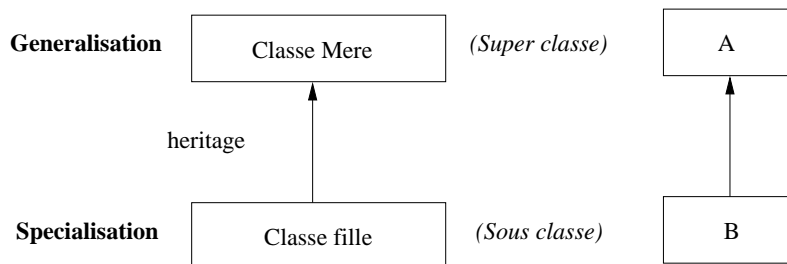


FIG. 1.1 – La notion d’héritage

1.4.1 Les constructeurs et destructeurs hérités

Soit le code suivant :

```
class A {
    int a;
public:
    A(int n) { a=n; }
    ~A();
};

class B : public A {
    int b;
public:
    B(int n, int m) : A(n) { b=m; }
};
```

lors de la création d’une instance de la classe B, le constructeur de la classe A est appelé préalablement.

1.4.2 héritage multiple

L’héritage multiple intervient lorsqu’une classe hérite d’autres classes qui possèdent une classe mère commune.

Le modèle le plus simple d’héritage multiple est le modèle en *diamant étroit* (cf. fig 1.2). Un modèle un peu plus complexe est appelé *diamant étendu*. C++ est un langage qui permet de prendre en compte l’héritage multiple.

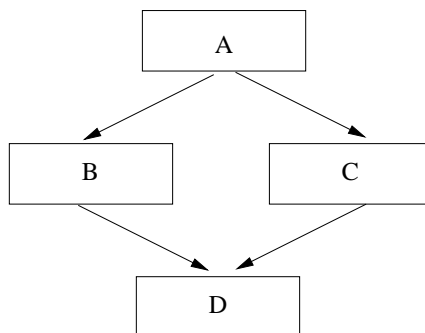


FIG. 1.2 – Héritage multiple simple (dit diamant étroit)

Dans ce cas de figure, si la classe **A** possède des attributs, ils sont dupliqués dans les classes **B** et **C**. La classe **D** hérite donc de deux jeux de données issus de **A**. Il faudra, pour remédier à ce problème, déclarer :

```
class B : public virtual A { ... };  
class C : public virtual A { ... };  
class D : public B, public C { ... };
```

1.4.3 les fonctions virtuelles

Les fonctions virtuelles permettent de redéfinir le comportement d'une méthode associée à une hiérarchie de classes.

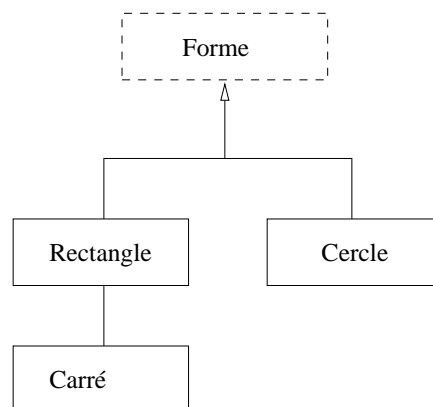


FIG. 1.3 – Exemple d'héritage multiple et fonctions virtuelles

Sur le diagramme de classes de la figure 1.3, on veut définir l'aire associée à chaque classe. Pour cela on utilise une méthode *virtuelle* dont le comportement pourra être modifié en fonction de la forme à laquelle elle se rapporte :

```

class Forme {
protected:
    int x, y;
public:
    Forme() { x=y=0; }
    Form(int xpos, int ypos) { x=xpos; y=ypos; }
    virtual int aire() { return 0; }
};

class Rectangle : public Forme {
protected:
    int width, height;
public:
    Rectangle(int x, int y, int w, int h) : Forme(x,y) { width=w; height=h; }
    int aire() { return width*height; }
};

class Carre : public Rectangle {
protected:
    int size;
public:
    Carre(int x, int y, int s) : Rectangle(x,y,s,s) { }
};

```

En particulier ici, il n'est pas nécessaire de redéfinir la méthode `aire` pour la classe `Carre` car elle a le même comportement que pour un rectangle.

1.4.4 les fonctions virtuelles pures et les classes abstraites

Les fonctions virtuelles pures permettent de définir des classes abstraites. Dans l'exemple précédent, la classe `Forme` ne possède pas d'aire, on a donc défini la fonction `aire` de manière à ce qu'elle retourne 0. Si on ne veut pas que le programmeur puisse manipuler des instances de la classe `Forme`, il suffit de définir une classe abstraite `Forme` en déclarant au moins une de ses méthodes *virtuelle pure* :

```

class Forme {
    int x, y;
public:
    Forme() { x=y=0; }
    Form(int xpos, int ypos) { x=xpos; y=ypos; }
    virtual int aire() = 0;
};

// on ne peut pas déclarer !!
Forme forme;
// mais par contre on peut utiliser :
Forme *ptr_forme;
// mais pas
ptr_forme=new Forme();

```

1.4.5 la force du C++ : fonctions virtuelles + typage dynamique

L'utilisation du typage dynamique associé à des classes déclarant des fonctions virtuelles permet d'utiliser toute la puissance du langage C++ .

Avec l'exemple précédent on obtient :

```

Carre c(0,0,10);
Forme *ptr_forme = &c;
Forme& ref_forme = c;

cout << "aire " << ptr_forme->aire() << endl;
cout << "aire " << ref_forme.aire() << endl;

```

L'exemple le plus simple pour comprendre ce que peut apporter le typage dynamique associé aux fonctions virtuelles consiste à considérer le problème suivant : on désire stocker dans un tableau (ou une liste), un ensemble de formes et calculer la somme des aires de toutes les formes.

```

Forme *tab[100];
tab[0] = new Carre(0,0,10);
tab[1] = new Rectangle(10,5,6,2);
...
tab[99] = new Carre(-5,-2,1);

int somme=0;
for (int i=0;i<100;++i) {
    somme += tab[i]->aire();
}

```

question subsidiaire : *que se passe-t-il si on ne déclare pas la fonction aire comme virtuelle ?*

Réponse : C'est la fonction aire de la classe Forme qui sera utilisée uniquement !

1.5 Généricité en C++

Un problème très largement rencontré en programmation classique consiste à implémenter une structure de données pour des types différents : par exemple, en C, si on désire définir une structure de liste chaînée pour les entiers, les chaînes et les réels, il faut écrire 3 modules différents de listes chacun adapté au type de données :

```

typedef struct { ... } ListInt;
typedef struct { ... } ListStr;
typedef struct { ... } ListFlt;

int ListIntGet(ListInt *l, int n);
int ListStrGet(ListStr *l, int n);
int ListFltGet(ListFlt *l, int n);

```

On peut apporter une solution à ce problème en utilisant une liste qui manipule des pointeurs du type `void *`, mais dans ce cas on est obligé d'allouer de la mémoire et on ne sait pas sur quel type de structure on pointe.

La généricité permet de définir un modèle (ou canevas) qui pourra être adapté en fonction des besoins des utilisateurs. En C++ on utilise le mot clé *template* pour définir des fonctions ou des classes génériques.

1.5.1 fonctions génériques

Voici un exemple de déclaration de fonction générique :

```
template <class T> T &min(T &a,T &b) {  
    return (a)<(b)?(a):(b);  
}  
  
int c=min(3,4);  
float f=min(3.14, 6.6666);
```

Celle-ci peut ensuite être appliquée à n'importe quel type/classe qui se conforme aux opérations utilisées dans la fonction. Le type de T est déterminé à l'exécution.

1.5.2 classes génériques

```

template <int maxSize,class T> Vector {
protected:
    int size_max;
    int size;
    T *tab;
public:
    Vector();
    Vector(int s);
    T& operator[](int index);
    int getSize();
};

template <int maxSize,class T>
Vector<maxSize,T>::Vector() {
    tab=new T[size_max=maxSize];
    size=0;
}

template <int maxSize,class T>
Vector<maxSize,T>::Vector(int s) {
    if (s>maxSize) s=maxSize;
    tab=new T[size_max=s];
    size=0;
}

template <int maxSize,class T>
T& Vector<maxSize,T>::operator[](int index) {
    return tab[index];
}

template <int maxSize,class T>
inline int Vector<maxSize,T>::getSize() {
    return size;
}

int i;
Vector<20,int> v1(10);

for (i=0;i<v1.getSize();++i) {
    v1[i]=i;
}
for (i=0;i<v1.getSize();++i) {
    cout << v1[i] << endl;
}
Vector<20,Vector<50,int> > v2(5);

```

A noter :

- pour certains compilateurs, l'utilisation de template requiert que les fonctions membres soient déclarées uniquement dans le fichier entête,
- ne pas oublier l'espace dans `Vector<20,Vector<50,int> >` sinon le compilateur comprendra `>>` qui est le symbole de redirection des flux d'entrée.

1.6 Les exceptions

Les exceptions sont un mécanisme de signalisation et de récupération sur erreur. Alors que dans les langages classiques une division par zéro produit un arrêt du programme, les exceptions permettent d'intercepter et de traiter les erreurs.

En C++ une exception est représentée par une classe qui comprend généralement une méthode `watch()` qui renvoie un message d'erreur. La levée et l'interception de l'exception se font comme en Java grâce aux instructions `throw` et `try / catch` :

```
class Exception {
public:
    Exception(char *s) { message=s; }
    string watch() const { return message; }
protected:
    string message;
};

int x, y ,z;
cout << "entrer deux nombres entiers : ";
cin >> x >> y;
try {
    if (y==0) {
        throw(Exception("division par zero"));
    } else {
        z=x/y;
    }
} catch(Exception e) {
    cout << "Exception : " << e.watch() << endl;
}
```

Autre exemple en utilisant la STL :

```

#include <exception>
using namespace std;

class Integer {
protected:
    int value;
public:
    Integer(int v) {
        value=v;
    }
    void divide(int v);
    class DivideByZero : public exception
    {
        public:virtual const char *what(void) const throw() {
            return "Division by zero";
        }
    };
};

void Integer::divide(int v) {
    if (v==0) throw DivideByZero();
    value/=v;
}

int main() {
    Integer i(10);
    try {
        i.divide(0);
    } catch(Integer::DivideByZero &e) {
        cout << "Exception : " << e.what() << endl;
    }
    return 0;
}

```

1.7 La STL

La STL (*Standard Template Library*) est une librairie C++ standard fournie avec la plupart des compilateurs C++, notamment le GCC (*GNU C Compiler*). Elle a été développée par Alexander Stepanov et Meng Lee en 1992 aux Hewlett Packard Lab à Palo Alto, CA. La STL fut adoptée par le comité ANSI/ISO C++ en 1994. la STL utilise de manière abondante la généricité .

L'implantation de la STL est laissée libre et suivant les versions il est nécessaire d'inclure les fichiers entête comme en C (<string.h>) ou sous forme plus générale <string>.

La STL se révèle difficile à utiliser pour le néophyte car elle repose sur une vision particulière (non objet) de l'implantation des containers en ce sens que les algorithmes qui agissent sur les containers sont externes à ceux-ci, alors que dans une approche objet on a tendance à faire en sorte que les algorithmes soient définis au sein de la classe qui les utilise. En outre, l'utilisation de la généricité est omniprésente.

Les algorithmes interagissent avec les containers par l'intermédiaire d'itérateurs qui ont pour but de manipuler les objets détenus par les containers.

La conception de la STL repose sur 5 composants principaux :

- *les containers* (ou conteneurs en français) sont des objets qui offrent une organisation capable de stocker d'autres objets
- *les algorithmes* qui sont des sous-programmes capables de travailler sur des containers.
- *les itérateurs* sont des objets capables d'accéder aux objets des containers

- les *objets fonction* qui sont des classes redéfinissant les opérateurs
- les *adaptateurs* (adaptors) sont des composants qui offrent des interfaces

1.7.1 La classe string

Elle permet de manipuler les chaînes de caractères bien plus simplement qu'en langage C.

```
string nom, prenom, s;
nom = "Richer";
prenom = "Jean-Michel";
s = nom + prenom;
s.insert(0, "Dr ");
```

1.7.2 Containers

Les containers sont divisés en deux catégories :

- les *Sequence Containers* stockent les objets de manière linéaire : Vector, Dequeue, List
- les *Associative Containers* offrent la possibilité d'un accès rapide grâce à l'utilisation de clés : Set, Multiset, Map, Multimap

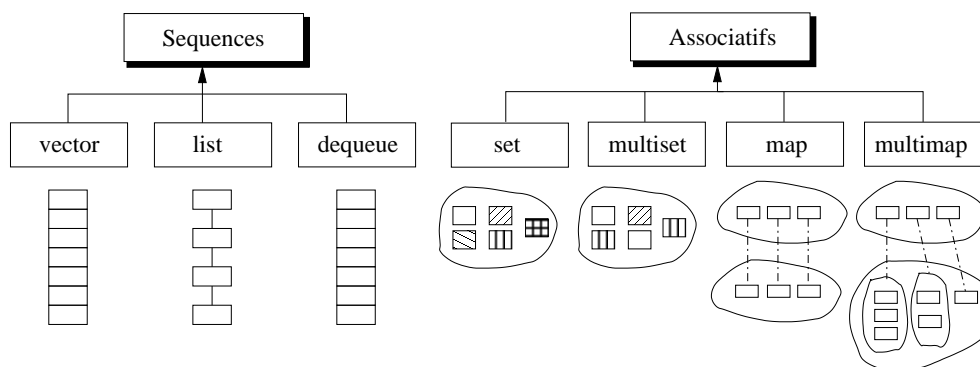


FIG. 1.4 – Standard Template Library

1.7.3 les itérateurs

Avec la STL chaque container `Contain` associé à un type `T` possède deux types d'itérateurs :

- `Contain::iterator` manipulant `T *`
- `Contain::const_iterator` manipulant `const T *`

Les containers définissent les méthodes suivantes afin d'être manipulés par les itérateurs :

- `iterator begin()`
- `iterator end()`
- `const_iterator begin() const`
- `const_iterator end() const`

Les itérateurs redéfinissent les opérateurs `++`, `==`, `*` et `->`.

1.7.4 La classe list

Les listes autorisent l'ajout d'éléments à tout endroit de la liste mais ne permettent qu'un accès séquentiel aux différents éléments.

```
#include <list>
#include <iterator>
```

```

#include <iostream>
#include <stdlib.h>
#include <unistd.h>

using namespace std;

typedef list<int> listInt;

int return_zero() { return 0; }

void cube(int n) { cout << n*n*n << endl; }

int main() {
    listInt l;
    unsigned int i;

    srand(getpid());
    for (i=0;i<10;i++) l.push_back(rand()%100);
    l.insert(l.end(),-1);
    generate_n(front_inserter(v),10,return_zero);
    listInt::iterator iter;
    for (iter=l.begin();iter!=l.end();++iter) cout << *iter << endl;
    for_each(l.begin(),l.end(),cube);
    return 0;
}

```

1.7.5 La classe vector

```

#include <vector>
#include <iterator>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>

using namespace std;

typedef vector<int> vectorInt;

int main() {
    unsigned int i;
    vectorInt v;

    srand(getpid());
    for (i=0;i<10;i++) v.push_back(rand()%100);

    vectorInt::iterator iter;
    for (iter=v.begin();iter!=v.end();++iter) {
        cout << *iter << endl;
    }
    sort(v.begin(),v.end());
    for (i=0;i<v.size();i++) cout << v[i] << endl;
    reverse(v.begin(),v.end());
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout," "));
}

```

```

return 0;
}

```

1.7.6 Les algorithmes

Il existe de nombreux algorithmes qui manipulent les containers :

- algorithmes non mutables : ils ne modifient pas les données
 - `for_each` : applique une fonction sur chacun des éléments du container sans modifier les données
 - `find`, `find_if` retourne l'itérateur qui correspond à la position d'un élément recherché
 - `search`
 - `count` compte le nombre d'éléments qui ont une valeur donnée
- algorithmes mutables : ils peuvent modifier les données ou la nature des données
 - `copy`
 - `fill`
 - `replace`
 - `reverse`
 - `remove`, `remove_if`
 - `generate_n` assigne le résultat de la fonction passée en paramètre
- algorithmes de tri
 - `sort`
 - `sort_heap`
 - `binary_search`
 - `remove`, `remove_if`
- les containers de séquences : permettent de stocker les objets sous forme d'une suite d'objets, l'efficacité du container dépend alors de l'implantation choisie (liste, vecteur, file)
- les containers associatifs : alors que l'on accède aux objets des containers de séquences par leur index, les containers associatifs permettent d'accéder aux objets par l'intermédiaire d'une *clé* qui peut être un objet.

1.7.7 quelques exemples

Conversion en majuscule

```

void convert_to_uppercase() {
    string s;

    s = "Voici un Texte, a Convertir en Majuscule";
    cout << s << endl;
    transform(s.begin(), s.end(), s.begin(), (int(*)(int)) toupper);
    cout << s << endl;
}

```

Somme des éléments d'un tableau

```

void sum_elem() {
    int tab[] = {10, -2, 8, 7, -3 };

    int sum = accumulate(&tab[0], &tab[5], 0);
    cout << "sum = " << sum << endl;
}

```

Tri

```
int tab[] = { 9, 8, 7, 2, 3, 5, 4 };
sort(&tab[0], &tab[7]);
vector<int> v(&tab[0], &tab[7]);
sort(v.begin(), v.end());
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

tokenizer

```
string s = " exemple de phrase a parser ";
istringstream stream(s);
string t;

int n=1;
while (stream >> t) {
    cout << n << ": " << t << endl;
    ++n;
}
```

mapping

```
map<string,int> occ;
string s="voici un exemple de phrase qui compte les occurrences des mots de la phrase";

istringstream stream(s);
string t;
while (stream >> t) {
    if (occ.find(t) == occ.end()) {
        occ[t] = 0;
    }
    ++occ[t];
}
map<string,int>::iterator iter;
for (iter = occ.begin(); iter != occ.end(); ++iter) {
    cout << (*iter).first << " = " << (*iter).second << endl;
}
```