3. La programmation générique

3.1 Introduction

Observation : certaines fonctions ou classes possèdent le même fonctionnement (algorithmique) et diffèrent sur les types de données traités. Il sera intéressant de concevoir une seule fonction (ou classe), qui paramétrise les types de données (le type d'une variable par exemple devient lui même un paramètre). La bonne fonction ou classe sera instanciée au besoin (lors de l'appel de la fonction ou l'utilisation de la classe).

Condition : les opérateurs appliqués aux différents variables et/ou objets de types paramétrés, qu'on ne connait pas à priori, doivent être réalisables sur les types réels utilisés, dans le cas des objets leurs classes doivent surcharger les opérateurs en question.

3.2 Les templates (Patrons)

3.2.1 Les templates de fonctions

Soit l'exemple des deux variantes de la fonction min suivant :

```
1 int min(int a,int b) {
2     return a<b? a:b;
3 }
4 float min(float a,float b) {
5 return a<b?a:b;
6 }
```

On remarque qu'il s'agit du même algorithme qui peut être utilisé avec différents type de données. Le paramétrage du type se fait en définissant un template (patron, modèle) de la fonction min :

```
1 template <class T> T min (T a,T b) {
2     return a<b?a:b;
3 }
```

Le mot clé class désigne un type quelconque (qui peut être primitif ou classe). Il peut être substitué par le mot clé **typename** qui est plus significatif.

Conception des templates

1. Les templates sont des déclarations. Ils sont destinés au compilateur qui crée au besoin (appels aux fonctions avec des paramètres effectifs) les fonctions réelles adéquates, on ne peut pas créer un module objet à partir des templates.
2. Pour une meilleure organisation du code, il est préférable de mettre en œuvre deux fichiers d'entête : Le premier contient uniquement les déclarations des templates ayant la syntaxe suivante :

```
1     template <class T,class U,...> <typeretour> <nomfct> (T a,U b
      ,...);
```

Le deuxième contient le corps des différents templates déclarés.

Pour utiliser le template uniquement par son fichier de déclaration (le premier), ce dernier doit inclure le deuxième fichier par la clause

```
#include "deuxième.h"
```

Il suffit à l'utilisateur d'inclure le premier fichier par

```
#include "premier.h"
```

Si le compilateur trouve :

```
      int x=5;
      int y=6;
      int z=min(x,y);
      %
```

Le compilateur va instancier, d'une manière transparente, une fonction à partir du template précédemment définit en remplaçant le type **T** par **int**. On aura la forme de la fonction min avec des entiers comme paramètres et valeur de retour. Si le compilateur trouve les instructions suivantes :

```
          float z=8.5;
          float s=1.6;
          float res=min(z,t);

```

Le compilateur procèdera de la même façon : une fonction min est créée avec des paramètres **float** et une valeur de retour de type **float**.

— L'appel

```
          res=min(a,z) // a est de type int et z de type
    float

```

provoquera une erreur de compilation car le template ne peut pas générer la fonction min(int,float) étant que les deux paramètres ne sont pas du même type, ici il n'y a pas de conversion explicite entre types.

— Si on a :

```
          Complex c1(1.2,2.3);
          Complex c2(3.2,1.5);
          Complex res=min(c1,c2);

```

Cet appel provoquera une erreur de compilation indiquant que l'opérateur de comparaison '<' n'est pas applicable aux objets de type Complex. La classe Complex doit surcharger, dans ce cas, l'opérateur < pour que cet appel soit valable. L'opérateur peut être surcharger par :

```
1       friend bool operator <(Complex,Complex); //dans .h
2       bool operator < (Complex c1,Complex c2) { // dans .
      cpp
3           m1=sqrt(c1.rel*c1.rel+c1.img*c1.img); // modulo
      de c1
4           m2=sqrt(c2.rel*c2.rel+c2.img*c2.img);//modulo de
       c2
5           return m1<m2?m1:m2;
6       }
7
```

spécialisation des templates

Dans certains cas, on veut changer l'algorithme du template pour des types spécifiques. Pour le template précédent, si on écrit :

```
1 Char *nom1="CCCC";
2 char *nom2="AAAA";
3 Char *res=min(nom1,nom2);
```

Le résultat sera le pointeur ayant l'adresse minimum et non l'ordre alphabétique des chaines. Si on veut que la fonction min effectue une comparaison alphabétique des chaines, on doit ajouter avec le template une spécialisation de cette fonction comme suit :

```
1 Char * min(char *a,char *b) {
2     if(strcmp(a,b)<0)
3         return a;
4     else return b;
5 }
```

— On peut procéder à des spécialisations partielles : Template <class T> T* min (T *a,int *b) ...,
— La forme spécialisée est toujours **plus prioritaire** que le template et prise en compte en premier par le compilateur.

surcharge des templates

On peut surcharger un template en définissant ainsi des familles de templates :

```
1 template <Class T, class U> T fct1(T a,U b,T c);
2 template <class T,class U> T fct2(T a,T b,U c);
```

3.2.2 Les templates de classes

De la même manière que les fonctions, ils existent des classes avec les mêmes données membres (attributs) et même fonctions membres (méthodes) ayant les mêmes algorithmes

mais diffèrent uniquement pour les types de données utilisés. L'exemple le plus connu est celui des structures dynamiques file, pile, vecteurs,etc. qui ont la même structure mais peuvent traiter et contenir des données de type quelconque (entier, réel, objets ...) **Exemple illustratif :**

```
class IntArray {
private:
    int *array;
    int size;
public:
    IntArray(int);
    void print();
    int getElement(int);
    int getSize();
    void setElement(int,int);
    ~IntArray();
};
```

Avec l'implémentation suivante :

```
IntArray::IntArray(int s){
    array=new int[size]; size=s;
}
void IntArray::print(){
    for(int i=0;i<size;i++) cout<<array[i]<<" ";
    cout <<'\n';
}
void IntArray::setElement(int index,int value){
    if (index<size)
        array[index]=value;
}
int IntArray::getElement(int index){
    return array[index];
}
IntArray::~IntArray(){
    delete array;
}
int IntArray::getSize(){
    return size;
}
```

On remarque que cette classe modélise des tableaux d'entiers crées dynamiquement. Le code de cette même classe est valable pour un autre type de donnée (des réels par exemple) en modifiant uniquement le type entier par un autre type. Si on veut écrire le template de la classe IntArray, on remplace int par le type générique T (dans le premier fichier d'entête) :

```
template <class T> class IntArray { private: T *array; int size;
public:
IntArray(int); void print(); T getElement(int); int getSize(); void
    setElement(int,T);
~IntArray();
};
```

Dans le deuxième fichier, On ajoute le code des différentes fonctions membres de la manière suivante :

```
#include <iostream>
#include "Generic.h" // Generic.h est supposé contenir la déclaration
                     // de la classe template IntArray précédente.
using namespace std;
template <class T> IntArray<T>::IntArray(int s){
    array=new T[size];
    size=s;
}
template <class T> void IntArray<T>::print(){
    for(int i=0;i<size;i++)
        cout<<array[i]<<" ";
    cout <<'\n';
}
template <class T> void IntArray<T>::setElement(int index,T value){
    if (index<size) array[index]=value;
}
template <class T> T IntArray<T>::getElement(int index){
    return array[index];
}
template <class T> IntArray<T>::~IntArray(){
    cout<<"suppression"<<'\n';
}
template <class T> int IntArray<T>::getSize(){
    return size;
}
```

La déclaration de la classe patron IntArray et la définition de ses fonctions membres doivent être incluses dans un programme pour que le compilateur génère les classes réelles correspondantes. On peut les mettre dans un seul fichier header (.h) ou séparer la déclaration et l'implémentation du template dans deux fichiers et l'un fait l'inclusion de l'autre. Une troisième solution avec la possibilité d'exportation permise par la norme C++ (liaison avec un fichier objet contenant des déclaration exportables).

Pour créer des classes réelles à partir de la classe template précédente on utilise la syntaxe :

```
IntArray <int> tab(5); // création d'une classe IntArray avec int comme
    T
for(int i=0;i<tab.getSize();i++){
    tab.setElement(i, i*2);
}
tab.print();
```

Si on veut la classe IntArray avec le type float :

```
IntArray <float> tab(5);
for(int i=0;i<tab.getSize();i++)
```

```
3       tab.setElement(i,(float) i*0.25);
4   tab.print();
```

Pour utiliser la classe IntArray avec des objets de la classe Complex :

```
1   IntArray <Complex> tab(5);
2   for(int i=0;i<tab.getSize();i++)
3       tab.setElement(i,Complex(i*0.25,i*0.25));
4   cout <<"taille du tableau:"<<tab.getSize()<<'\n'; tab.print();
```

La compilation va générer une erreur dans la fonction membre print() indiquant que l'opérateur « n'est pas applicable sur des objets de la classe Complex. Pour y remédier, on doit surcharger l'opérateur « dans la classe Complex en ajoutant en premier lieu dans Complex.h :

```
1   friend ostream & operator <<(ostream &,Complex); //
```

L'opérateur « doit retourner une référence à un objet de la classe ostream (flux de sortie qui peut être envoyé à un autre flux de sortie pour composer plusieurs flux utilisables par cout). Le premier paramètre de la fonction sert comme valeur de retour, le deuxième est l'objet pour lequel l'opérateur est appliqué, par exemple on envoi dans le premier paramètre les données membres de l'objet Complex. On ajoute dans Complex.cpp :

```
1   ostream & operator << (ostream & sortie, Complex p) {
2       sortie << "<" << p.rel << "," << p.img << ">" ;
3       return sortie ;
4   }
```

R Il faut veiller dans la conception des templates de classes à ce que les opérateurs utilisés soient applicables aux types génériques employés dans l'utilisation du template.

3.3 La Standard Template Library (STL)

La bibliothèque standard du langage C++ inclut un ensemble de classes générique regroupées sous le nom de STL (standard Template Library) introduite par Alexander Stépanove (le père de la généricité). Concepts de la STL :

- Conteneur : Un Objet qui contient d'autres objets.
- Itérateur : Un objet utilisé pour le parcours d'un conteneur.
- Algorithme : fonction utilisée pour le traitement des éléments d'un conteneur, ces fonctions manipulent les itérateurs et ne touchent pas directement aux éléments du conteneur.
- Complexité : coût d'un algorithme en fonction de la taille n du conteneur :
  - Instantané : $O(1)$
  - Logarithmique : $O(log(n))$
  - Polynomiale : $O(N^k)$

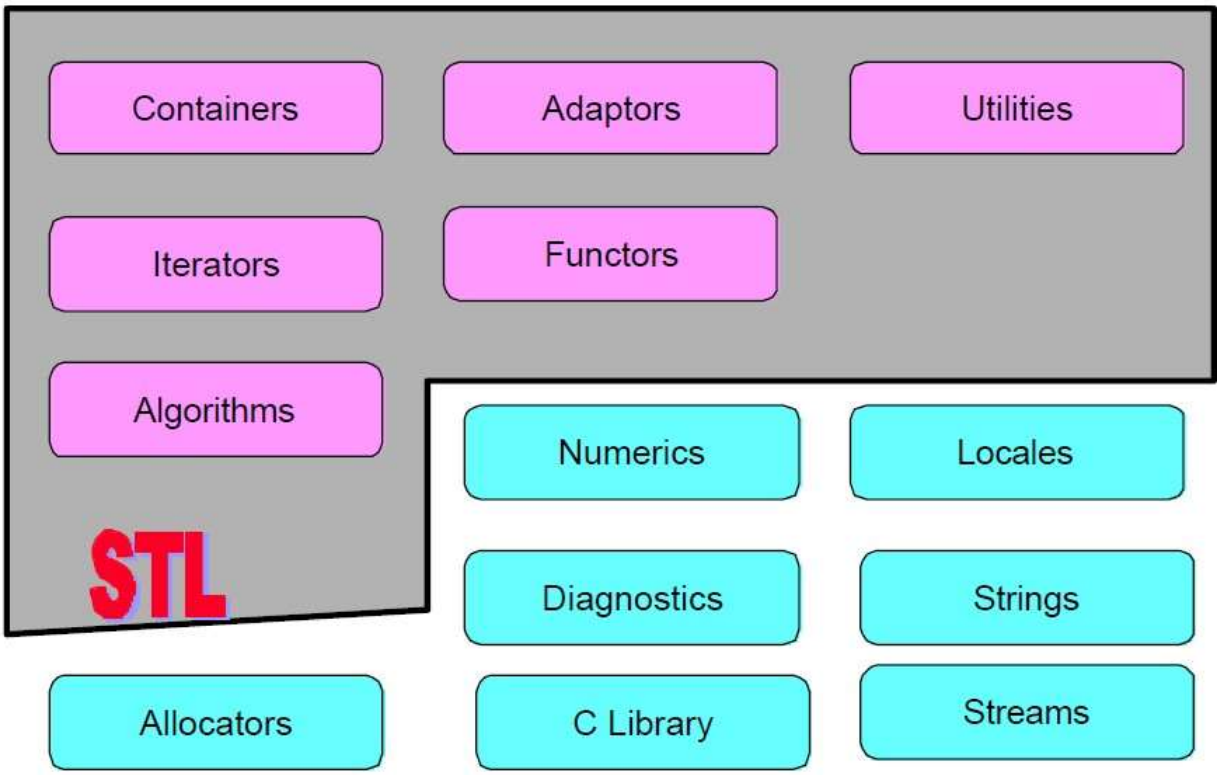

FIGURE 3.1 – Composants de la STL

— ...

La figure 3.1 illustre les différents composants de la STL standard de C++.

On peut classer les conteneurs en deux familles :

1. Conteneurs séquentiels : comme vector,deque, list où les éléments sont stockés dans un ordre décidé par l'utilisateur (par exemple l'ordre d'insertion).
2. Conteneur associatifs : comme set, multiset, map et multimap où les éléments sont stockés et triés selon un ordre associé au type (en général <).

La figure 3.2 présente les principaux conteneurs de la STL.

Opérations communes aux conteneurs :

si TC est un conteneur alors :

— **TC<types> c** : crée un conteneur c vide.
— **TC c1(c2)** : crée un conteneur c1 de même type que c2 et copie ses éléments.
— **TC c(beg,end)** : crée un conteneur á partir de deux itérateurs et copie les éléments compris entre [beg,end].
— **c.size()** : retourne le nombre d'éléments dans le conteneur.
— **c.empty()** : retourne vrai si le conteneur est vide.
— **c.max_size()** : retourne le nombre maximal d'éléments pouvant être contenus dans le conteneur.
— **c1==c2, c1 !=c2, c1<c2, c1>c2, c1<=c2, c1>=c2** : compare deux conteneurs (ordre lexicographique).
— **c1=c2** : copie tous les éléments de c2 dans c1.
— **c1.swap(c2), swap(c1,c2)** : échange les éléments de c1, c2.

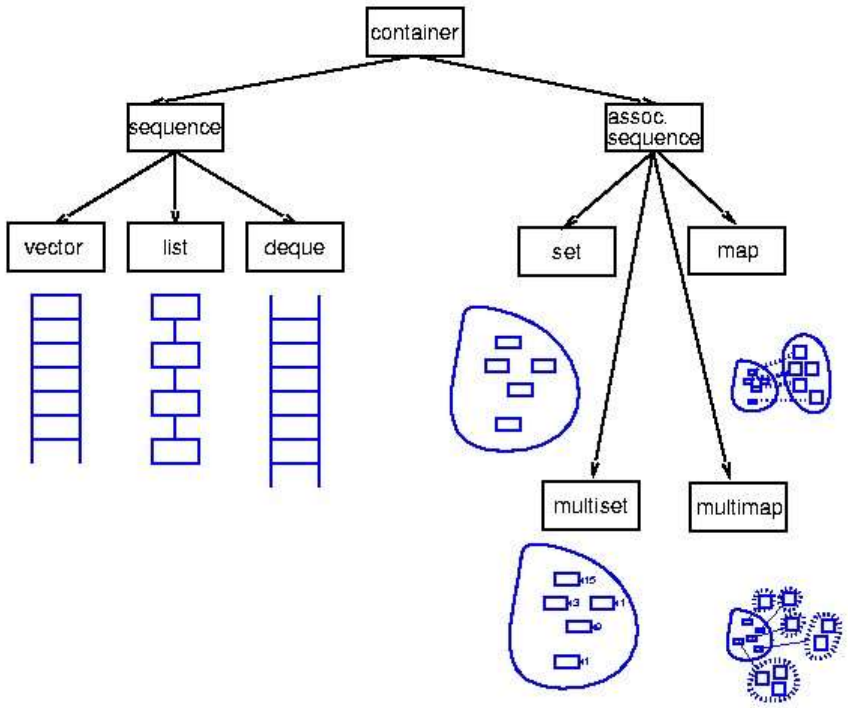

FIGURE 3.2 – Composants de la STL

— **c.begin()** : un itérateur vers le premier élément de c.
— **c.end()** :un itérateur vers la position après le dernier élément de c.
— **c.rbegin()** : un itérateur inverse vers le dernier élément de c.
— **c.rend()** : un itérateur inverse vers la position avant le premier élément de c.
— **c.insert(pos,elem)** : insère une copie de elem à la position pos.
— **c.erase(beg,end)** : efface les éléments de l'intervalle [beg,end].
— **c.clear()** : efface tous les éléments du conteneur.

3.3.1 Les conteneurs séquentiels

Le template Vector

C'est une représentation d'un tableau dynamique et des opérations permettant de manipuler les données qui y sont stockés.

Caractéristiques :

— L'accès à un élément se fait en O(1),
— L'insertion à la fin se fait en O(1), sauf exception,
— L'insertion ailleurs que àla fin se fait en O(n),
— Un vecteur se comporte d'une façon proche d'un tableau, avec une syntaxe plus riche,
— Un vecteur peut se comporter aussi comme une pile avec les fonctions **push_back()** et **pop_back()**.

Exemple :

```
1 #include <iostream>
2 #include <vector>
```

```
using namespace std;
int main(){
    vector<int> vi(100),vj;
    for (int i=0; i<100; ++i){
        vi[i]=100-i;
        vj.push_back(i); }

    vi.resize(128);
    for (int i=100; i<128; ++i) vi[i]=100-i;
}
```

Le template list

Une liste est un conteneur qui organise les éléments dans une liste doublement chainée comme le montre la figure 3.3.

L'insertion dans une liste se fait en O(1), l'accès à un élément en O(n).

Il n'y a pas d'opérateur d'accès direct aux éléments d'une liste, tout accès doit être séquentiel. Pas de fonction at() où d'opérateur []. Il n'y a pas également de taille pré-allouée pour une liste, donc pas de capacity() où resize().

Exemple 1 :

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> lst; // une liste vide
    lst.push_back( 5 ); lst.push_back( 6 ); lst.push_back( 1 ); lst.
    push_back( 10 );
    lst.push_back( 7 ); lst.push_back( 8 ); lst.push_back( 4 ); lst.
    push_back( 5 );
    lst.pop_back(); // enleve le dernier élément et supprime l'entier 5
    cout << "La liste lst contient " << lst.size() << " entiers : \n";
}
```

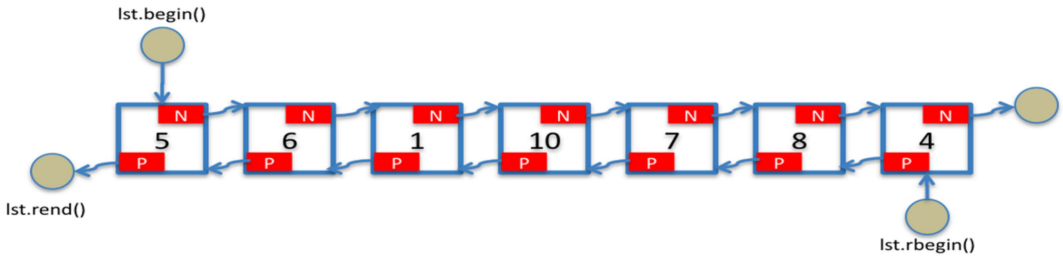

FIGURE 3.3 – Conteneur séquentiel list de la STL

Exemple 2 :

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

```
int main()
{
    list<int> lst; // une liste vide
    lst.push_back( 5 ); lst.push_back( 6 ); lst.push_back( 1 ); lst.push_back( 10 );
    lst.push_back( 7 ); lst.push_back( 8 ); lst.push_back( 4 ); lst.push_back( 5 );
    lst.pop_back(); // enleve le dernier élément et supprime l'entier 5
    cout << "La liste lst contient " << lst.size() << " entiers : \n";
    // utilisation d'un itérateur pour parcourir la liste lst
    for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
        cout << ' ' << *it <<endl; // parcours direct
    for (list<int>::reverse_iterator it = lst.rbegin(); it != lst.rend(); ++it)
        cout << ' ' << *it <<endl; // parcours inversé
    // afficher le premier et le dernier élément
    cout << "Premier element:" << lst.front() << "  Dernier element : " << lst.back() << endl;
    // utilisation d'un itérateur pour parcourir la liste lst
    for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
        cout << ' ' << *it <<endl; // parcours direct
    for (list<int>::reverse_iterator it = lst.rbegin(); it != lst.rend(); ++it)
        cout << ' ' << *it <<endl; // parcours inversé
    // afficher le premier et le dernier élément
    cout << "Premier element:" << lst.front() << " Dernier element : " << lst.back() << endl;
```

3.3.2 Les conteneurs associatifs

Le template table associatif (map)

Les maps permettent d'associer une clé à une donnée. Chaque élément d'un map est constitué d'une paire <clé, donnée>. L'interêt des maps réside dans le fait que la complexité algorithmique d'accès est de l'ordre de O(log(n)) insertion ou recherche.

Exemple :

```
#include <iostream>
#include <iomanip>
#include <map>
#include <string>
using namespace std;
int main() {
    map<string,unsigned int> nbJoursMois;
    nbJoursMois["janvier"] = 31;
    nbJoursMois["février"] = 28;
    nbJoursMois["mars"] = 31;
    nbJoursMois["avril"] = 30;
    //...
    cout << "La map contient " << nbJoursMois.size() << " elements : \n";
    for (map<string,unsigned int>::iterator it=nbJoursMois.begin(); it!=nbJoursMois.end(); ++ it)
        cout << it->first << " -> \t" << it->second << endl;
```

```

```

R Le fait d'accéder à une clé via l'opérateur [] insère cette clé dans la map (avec le constructeur par défaut pour la donnée). Ainsi l'opérateur [] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find.

La STL offre la structure pair (structure générique) qui permet de créer des paires de valeurs :

```
1       pair<int,int> p=make\_pair(3,2); pair<char,int> q=make\_pair('B',3);
```

Certaines fonctions des maps (comme find) retournent une pair de valeur (un objet de la classe pair).

3.4 La Boost Library C++

La BoostLibrary c++, est une bibliothèque non standard de C++, certains de ses fonctions ont été intégré dans le standard. Elle offre à l'utilisateur le moyen de manipuler :

- Les Threads,
- Les Matrices, le calcul algébrique les graphes mathémaiques, les aléatoires ...,
- Les Lambda-calculs,
- La méta-programmation,
- Les fichiers et répertoires,
- La sérialisation,etc.

Lien : "http ://www.boost.org/"

Documentation : "http ://www.boost.org/doc/libs/1_53_0/libs/libraries.htm"