Séance 7

Templates + STL

Template

- Les templates permettent à une fonction ou une classe d'utiliser les types différents :
  - Performance : code compact et efficace
  - Temps de codage
  - Clarté
- Programmation générique
  - Algorithmes génériques
  - Polymorphisme paramétrique

Fonction template

```
// dans un fichier.h

template <typename T>    // or template <class T>
T min(const T & a, const T & b) {
     return (a < b ? a : b);
}

// dans main.cpp
double minimum1;
minimum1=min<double>(d1, d2);
int mininum2;
min2=min<int>(i1,i2);
Personne pmin;
pmin=min<Personne>(p1,p2)          // il faut surcharger « operator< »
```

Fonction templale

```
// dans un fichier.h
template<typename T1, typename T2>
T2 calculerMoyenne(T1 tableau[], int taille) {
    T2 somme  = 0.0;
    for(int i = 0; i<taille; ++i)
            somme += tableau[i];
    return somme/taille;
}

// dans main.cpp
double avg;
avg = calculerMoyenne< int, double, >(tab,5);
```

Votre première classe C++ : ***Vector.h***

```
class Vector {
    private:
        double* elem;
        unsigned int size;
    public:
        Vector(unsigned int s);
        ~Vector();
        double & operator[](unsigned int i);
        unsigned int getSize() const;
};
```

Classe template : vector.h

```
template<typename T, int size>
class Vector {
    private:
        T* elem;
         unsigned int size;
    public:
        Vector(unsigned int s);
        ~Vector() { delete[] elem; }
        // ... copy and move operations ...
        T& operator[](unsigned int i);
        const T& operator[](unsigned int i) const;
        unsigned int getSize() const { return size; }
};
```

vector.cpp

```
template<typename T>
Vector<T>::Vector(unsigned int s) {
    elem = new T[s];
    size = s;
}
```

Vector.h

```
template<typename T>
const T& Vector<T>::operator[](unsigned int i) const {
    if (getSize()<=i II getSize() > size )
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

main.cpp

```
Vector<char> vc(200); // vecteur de caractères
Vector<string> vs(17); // vecteur de strings
Vector<list<int>> vli(45); // vecteur de listes
```

Pourquoi Standard Template Library (STL) ?

"*Don’t reinvent the wheel !"*

(Stroustrup)

- Algorithmes et structures de données fondamentaux
- Librairies bien codées, testées et optimisées

Standard Template Library (STL)

- Conteneurs : vector, list, map, stack, ...
- Itérateurs
- Algorithmes : insertion, suppression, recherche, tri, ...
- Foncteur

STL : conteneurs

- Les conteneurs sont des classes permettant de représenter les structures de données les plus répandues.
- 2 catégories de conteneurs :
  - Conteneurs séquentiels: les éléments sont ordonnés. On peut parcourir le conteneur suivant cet ordre et insérer ou supprimer un élément à un endroit explicitement choisi : vector, list, stack, …
  - Conteneurs associatifs : les éléments sont identifiés par une clé et ordonnés selon celle-ci: set, multiset, map, multimap
- Fonctionnalités communes à ces conteneurs:
  - int size() const;
  - bool empty() const;
  - ...

STL : conteneur séquentiel list

Déclaration :

```
list<Type> l;
```

Quelques fonctions:

```
void  push_front(const Type &);
void  pop_front( );
void  remove(const Type &);
```

Exemple :

```
#include <list>
list<int> l;
for (int i=0; i<5; i++)
    l.push_back((10+2*i)% 5 +i);                    // {0,3,5,4,1}
l.sort();                                           // {0,1,3,4,5}
l.reverse();                                        // {5,4,3,1,0}
cout << l.front() << " " << l.back() << endl;
```

STL : conteneur associatif map

- Associer à un ensemble de clefs un ensemble correspondant de valeurs.

Déclaration:

```
map<Cle, Valeur> m;
```

Exemple :

```
#include <map>
map <string, string> repertoire;
repertoire["Jean Martin"] = "01.02.03.04.05";
repertoire["François Martin"] = "02.03.04.05.06";
repertoire["Louis Dupont "] = "03.04.05.06.07";

repertoire.insert(pair<string,string>("Louis Martin" ,  "04.05.06.07.08" ));
```

STL: itérateurs

Définition :

- Un itérateur est un objet qui peut pointer à un élément dans une collection, et qui a la capacité de parcourir les éléments de la collection en utilisant un ensemble des opérateurs (incrémentation ++, déréférence *, ...).
- Pas besoin de faire de distinction entre les conteneur vector, les map ou list même
- Chaque conteneur possède son propre itérateur
- Quelque soit le conteneur l'itérateur est déclaré de la même manière
- deque : double-ended queue. Une généralisation des piles et des files. Il est possible d'ajouter et retirer des éléments par les deux bouts

```
1 #include <vector>
2 using namespace std;
3
4 vector<int> tableau(5,4);      //Un tableau de 5 entiers valant 4
5 vector<int>::iterator it;      //Un itérateur sur un vector d'entiers
```

```
1 map<string, int>::iterator it1; //Un itérateur sur les tables associatives string-int
2
3 deque<char>::iterator it2; //Un itérateur sur une deque de caractères
4
5 list<double>::iterator it3; //Un itérateur sur une liste de nombres à virgule
```

STL: itérateurs

- Methode begin() renvoye l'itérateur sur le premier élément du conteneur.
- On avance alors dans le conteneur en utilisant l'opérateur ++
- Méthode end() renvoye l'itérateur sur la fin du conteneur

```
1  #include<deque>
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      deque<int> d(5,6);          //Une deque de 5 éléments valant 6
8      deque<int>::iterator it;    //Un itérateur sur une deque d'entiers
9
10     //Et on itère sur la deque
11     for(it = d.begin(); it!=d.end(); ++it)
12     {
13         cout << *it << endl;     //On accède à l'élément pointé via l'étoile
14     }
15     return 0;
16 }
```

STL: itérateurs

- Les random access iterators
- Ces itérateurs proposent des opérateurs + et - permettant d'avancer de plusieurs éléments à la fois
- Par exemple pour accéder au huitième élément d'un vector, on peut utiliser la syntaxe suivante :

```
1 vector<int> tab(100,2);  //Un tableau de 100 entiers valant 2
2
3 vector<int>::iterator it = tab.begin() + 7; //Un itérateur sur le 8ème élément
```

STL: foncteurs (function object)

Définition :

- Un foncteur est un objet qui se comporte comme une fonction
- Les foncteurs sont des objets possédant une surcharge de l'opérateur ().
  Ils sont souvent utilisés comme arguments de fonction, comme des prédicats ou des fonctions de comparaison dans les algorithmes.

Exemple : fichier addition.h

```
1 class Addition{
2 public:
3
4     int operator()(int a, int b)   //La surcharge de l'opérateur ()
5     {
6         return a+b;
7     }
8 };
```

Cette classe ne possède pas d'attribut et juste une méthode, la surcharge de l'opérateur().
Comme il n'y a pas d'attribut, le constructeur par défaut est largement suffisant.

STL: foncteurs (function object)

```
1.  #include <iostream>
2.  #include "addition.h"
3.  using namespace std;

4.  int main()
5.  {
6.      Addition add;
7.      int a(2), b(3);
8.      cout << a << " + " << b << " = " << add(a,b) << endl; //On utilise le foncteur comme s'il
    s'agissait d'une fonction
9.      return  0;
10. }
```

Le résultat

```
2 + 3 = 5
```

STL: foncteurs évolutifs

- Les foncteurs peuvent utiliser des attributs comme n'importe quelle autre classe.
- créer une fonction avec une mémoire.

```cpp
#include <iostream>
#include <vector>
using namespace std;
class Remplir {
public:
    Remplir(int i):val(i) {};
    int operator()() {
     ++val;
     return val;
    }
private:
    int val;
};

int main() {
    vector<int> tab(10,0);//un tableau de 50 entiers de valeur 0
    Remplir f(0);
    for(vector<int>::iterator it=tab.begin();it!=tab.end(); ++it){
      *it=f(); //on appelle le foncteur sur chaque element de tab
     cout << (*it)<<endl;
    }
 return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

STL: foncteurs génériques et prédicat

Définition :

- Générique : type inconnu à la création
- Prédicat : foncteurs prenant un seul argument et renvoyant un booléen.
- Ils servent à tester une propriété particulière de l'objet passé en argument

Exemple :

```
template <class T>
 class  greater {
  bool operator() (const T& x, const T& y) const   {
          return x>y;
    }
};
int numbers[]={20,40,50,10,30};
bool response;
response = std::sort (numbers, numbers+5, std::greater<int>());
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
    // vector<int> v = { 1, 5, 2, 4, 3 };
     vector<int> v;
     v.push_back(1);
     v.push_back(5);
     v.push_back(2);
     v.push_back(4);
     v.push_back(3);
    // v=[1, 5, 2, 4, 3 ];

     vector<int>::iterator it;

     sort(v.begin(), v.end(), greater<int>());

     for (auto value: v)
           cout << "auto : " << value << endl;

      sort(v.begin(), v.end());  // using default comparison (operator <)

     for (it=v.begin(); it!=v.end(); ++it)
        cout << "iterator : " <<*it << endl;

     cout << endl;
}
```

```
auto : 5
auto : 4
auto : 3
auto : 2
auto : 1
iterator : 1
iterator : 2
iterator : 3
iterator : 4
iterator : 5
```

STL: algorithmes

Définition :

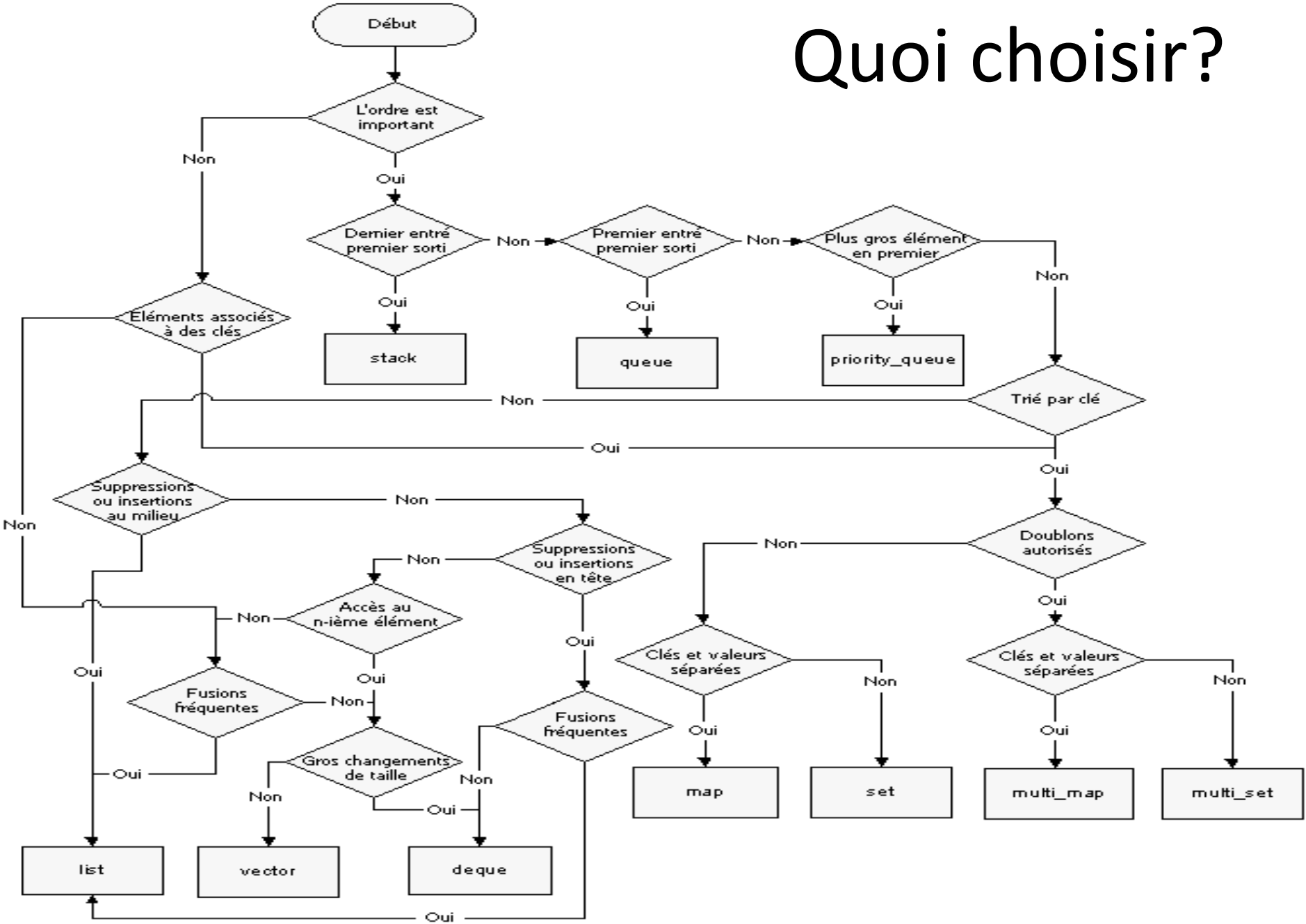
Un algorithme est une fonction template qui peut être utilisée dans une collection des éléments, en utilisant des itérateurs.

Exemple :

Compter le nombre d'occurrences d'une valeur dans un intervalle d'itérateurs:

```
#include<algorithm>
…
vector <int> v;
int n = count(v.begin(), v.end(),1);  //nombre d'elements egaux à 1
list<double> l;
int m = count(l.begin(),l.end(),3.5);  // nombre d'elements egaux à 3.5
```

Quoi choisir?



Lien

- lien vers la STL