

Templates (patrons) & STL

- Les *templates* (patrons) fournissent un **mécanisme de généralisation de code** supplémentaire
 - permet d'écrire des fonctions / algorithmes ou des classes conteneurs (piles, files, listes, arbres, *hash-table*...) pour n'importe quel type de données
 - cela permet de **réutiliser des algorithmes stables, performants et éprouvés** pour différents types sans devoir réécrire le code (STL)
 - le **compilateur fabrique le code exécutable adapté à chaque type à partir du code patron**

Templates

> Patrons de fonctions

- Exemple : écrire la fonction de calcul du minimum de deux valeurs pour plusieurs types (`int`, `double`, voire `Vector3`...)

```
int min(int a, int b) { return (a < b)? a : b; }  
double min(double a, double b) { return (a < b)? a : b; }
```

- En C++ on peut simplement définir le patron de fonction

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

on déclare un patron comportant un seul paramètre — type générique — que l'on appelle `T`, pour l'instant inconnu

le patron s'applique à une fonction de nom `min`, qui prend deux paramètres de type `T` et retourne une valeur de même type

Patrons de fonctions

> Instanciation du code

- Pour utiliser la fonction `min`, il suffit simplement de l'utiliser dans des conditions appropriées
 - deux arguments de même type
 - le type utilisé doit disposer d'un opérateur de comparaison `<`
- Par exemple, si on appelle `min(p,q)` avec `p` et `q` deux entiers `int`, le compilateurinstanciera automatiquement la fonction `int min(int a, int b)`
- Le code d'une fonction patron ressemble au code d'une fonction classique mais en pratique sa définition sera souvent placée dans le fichier `.h` (car instancié chaque fois que nécessaire)

Patrons de fonctions

> Instanciation du code

- On peut aussi séparer en-tête et corps de fonctions dans les fichiers `.h` et `.c`

```
// myTemplateMin.h
template <class T> //déclaration du patron
T min(T a, T b);
```

```
// myTemplateMin.cc
#include "myTemplateMin.h"
template <class T> //définition du patron
T min(T a, T b)    // corps de la fonction
{
    return (a < b)? a : b;
}
```

Patrons de fonctions

> Remarques

- Un patron de fonction peut comporter plusieurs paramètres

```
template <class T, class U>  
T somme(T a[], U op) { for (...) x=op.eval(x, a[i]); return x; }
```

- On peut surcharger des patrons de fonctions comme pour toute autre fonction

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

```
template <class T>  
T min(T a, T b, T c) { return min(min(a,b),c); }
```

- Un patron de fonction peut comporter également des paramètres non génériques ...

Templates

> Classes génériques

- On peut définir de la même façon des patrons de classe, la généricité s'applique alors à toute la classe :

```
template <class T>
class Vector3 {
    T _coeff[4];
public:
    Vector3(const T& s);

    T& operator[](int i);
    ...
}
```

```
template <class T>
T& Vector3<int>::operator[](int i) { return _coeff[i]; }
```

```
// déclaration de variables (fichier main.cc)
Vector3<int>    v1(10);
```

Templates

> Classes génériques

- On peut même utiliser une valeur de type simple (`int`, `double`...) comme paramètre, par exemple pour dimensionner la taille d'un type générique lors de son instantiation

```
template <class T, unsigned int size>
class Vector {
    T _coeff[size];
public:
    Vector(const T& s);

    T& operator[](int i) { return _coeff[i]; }
    ...
}
```

```
typedef Vector<double,8> VectorD8; // simplifier les déclarations de variable
typedef Vector<int,16> VectorI16;
```

Classes génériques

> Standard Template Library (STL)

- C++ inclut en standard (multi-plateforme) une bibliothèque de classes génériques très utilisées : la STL
 - listes, tableaux, tables de hachage, itérateurs et autres algorithmes génériques (recherche...)
- Permet d'unifier les définitions génériques les plus classiques comme les listes, les arbres *etc.*
- Permet de réduire fortement le temps de développement et de se concentrer sur la valeur ajoutée de son application
- Permet de bénéficier du travail testé et éprouvé de nombreux développeurs dans le domaine

Standard Template Library (STL)

> Conteneurs & Algorithmes

- Conteneurs : classes dont le rôle est de stocker d'autres objets

```
vector<int> v(3);           // déclare un vecteur de 3 éléments.  
  
v[0] = 7;  
v[1] = v[0] + 3;  
v[2] = v[0] + v[1];        // v[0] == 5, v[1] == -3, v[2] == 7
```

- On peut manipuler ces conteneurs via des algorithmes et outils classiques :

```
reverse(v.begin(), v.end()); // v[0] == 7, v[1] == -3, v[2] == 5  
  
int n = 12;  
v.resize(n); // redimensionne dynamiquement le vecteur
```

Classes génériques

> Standard Template Library (STL)

- `vector` : tableau redimensionnable dynamiquement
- `list`, `deque` : listes doublement chaînées
- `set`, `multiset` : ensemble d'éléments triés avec (`multiset`) ou sans doublon autorisé (`set`)
 - unions, intersections, insertions rapides...
- `map`, `multimap` : tableaux associatifs avec possibilité de définir un ordre particulier, strict (`map`) ou non strict (`multimap`)
- `hash_map`, `hash_multimap` : tableaux associatifs avec accès rapide (tables de hachage) mais sans ordre particulier

Standard Template Library (STL)

> Itérateurs

- Les itérateurs sont une **généralisation des pointeurs** pour des structures non alignées en mémoire :

```
list<double> l;           // liste vide

l.push_front(-2.7);       // insertion en tête
l.push_back(2.3);         // insertion en queue
l.push_front(5.9);        // insertion en tête

list<double>::iterator it = l.begin();

cout << *it << endl;     // 5.9
it++;
cout << *it << endl;     // -2.7

for (it = l.begin(); it != l.end(); it++)
    cout << *it << ", "; // 5.9, -2.7, 2.3
```

on parcourt la liste en commençant au début et on affiche chaque valeur



Standard Template Library (STL)

> Itérateurs

- On peut réutiliser le même algorithme pour parcourir un vecteur :

```
vector<double> v(3);           // vecteur de 3 éléments
```

```
v[0] = 5.9;  
v[1] = -2.7;  
v[2] = 2.3;
```

```
vector<double>::iterator it = v.begin();
```

```
cout << *it << endl;         // 5.9  
it++;  
cout << *it << endl;         // -2.7
```

on parcourt le vecteur en
commençant au début et on
affiche chaque valeur

```
for (it = v.begin(); it != v.end(); it++)  
    cout << *it << ", ";    // 5.9, -2.7, 2.3
```

Standard Template Library (STL)

> Itérateurs

- Mécanisme qui permet de découpler structures et algorithmes :

la fonction template `find()` cherche **value** dans un conteneur et retourne un itérateur (=last si value n'est pas présente)

on cherche la valeur -2.7 dans **la liste** avec la fonction `template find()`

on cherche la valeur -2.7 dans **le vecteur** avec la même fonction `template find()`

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T& value) {
    while (first != last && *first != value) ++first;
    return first;
}
```

...

```
list<double> l;           // liste vide
```

...

```
list<double>::iterator il = find(l.begin(), l.end(), -2.7);
cout << *il << endl;      // -2.7
```

```
vector<double> v(3);      // vecteur
```

...

```
vector<double>::iterator iv = find(l.begin(), l.end(), -2.7);
cout << *iv << endl;      // -2.7
```

Standard Template Library

> Exemple

- On veut utiliser **un tableau associatif** (`map`) pour stocker les valeurs de nos variables et les retrouver rapidement par leur nom.
- Un tableau associatif permet de stocker des valeurs par couple (*key*, *value*). Pour nous la clé sera le nom de la variable et la valeur la valeur de la variable : ex : ("**x**", 2.7).
- On utilise la classe générique `map<class K, class V>` définie dans la STL.
- En outre, on va utiliser une classe standard du C++ (`string`) qui permet de manipuler aisément les chaînes de caractères.

```
map<string, double> var_tab;
```

```
var_tab["x"] = 2.6;  
var_tab["y"] = 6.5e-2;  
var_tab["z"] = -56.0;
```

```
cout << "x = " << var_tab["x"] << endl;
```

```
map<string, double>::iterator it;
```

```
it = var_tab.find("y");  
if (it != var_tab.end()) {  
    cout << it->first << " = " << it->second << endl;  
} else {  
    cout << "y does not exist" << endl;  
}
```

```
it = var_tab.find("Y");  
if (it != var_tab.end()) {  
    cout << it->first << " = " << it->second << endl;  
} else {  
    cout << "Y does not exist" << endl;  
}
```

en sortie

```
> x = 2.6  
> y = 0.065  
> Y does not exist
```

Standard
Template
Library

la méthode `find` retourne un itérateur sur la paire («y», valeur) si elle existe, sinon elle retourne la valeur `end()`

l'itérateur `it` pointe sur une paire de type `pair<Key,Value>`, c.-à-d. une structure contenant deux champs : `first` (qui stocke la clé) et `second`, qui stocke la valeur

183

// fichier variable.h

```
class Variable : public Expression {
```

```
protected:
```

```
    // tableau associatif des valeurs de toutes les variables
```

```
    static map<string,double> _memory;
```

```
public:
```

```
    Variable(const string& name) : Expression(name) {  
        if (_memory.find(name) == _memory.end())  
            _memory[name] = 0.0;
```

si la variable n'existe pas encore dans la mémoire on l'ajoute avec la valeur 0.0

```
    }
```

```
    Variable(const string& name, double val_init) : Expression(name) {  
        _memory[name] = val_init;
```

```
    }
```

```
    double eval() const { return _memory[this->getDescription()]; }
```

```
// méthodes de classe pour lire/écrire les valeurs de variables
```

```
static double getValue(const string& name) {  
    return _memory[name];  
}
```

```
static void setValue(const string& name, double val) {  
    _memory[name] = val;  
}
```

```
};
```

// fichier variable.cc

```
map<string,double> Variable::_memory;
```

ne pas oublier de définir
l'attribut de classe dans le
fichier source variable.cc

Standard
Template
Library

184

```

int main (int argc, char * const argv[]) {
    Mult    *m = new Mult(new Const(2.0), new Cos(new Variable("x")));

    Plus    exp(new Const(2.0), m);

    cout << "exp : " << exp.getDescription() << endl;

    for (unsigned i = 0; i <= 6; i++) {

        Variable::setValue("x", 2*i*M_PI/6.0);

        cout << "x = " << Variable::getValue("x")
              << " ; exp = " << exp.eval() << endl;

    }

    return 0;
}
  
```

en sortie

```

> exp : (2 + (2 * cos(x)))
> x = 0 ; exp = 4
> x = 1.0472 ; exp = 3
> x = 2.0944 ; exp = 1
> x = 3.14159 ; exp = 0
> x = 4.18879 ; exp = 1
> x = 5.23599 ; exp = 3
> x = 6.28319 ; exp = 4
  
```

```
// fichier variable.h
class Variable : public Expression {
protected:
    // tableau associatif des valeurs de toutes les variables
    static map<string,double> _memory;

public:
    ...
    // méthodes de classe pour lire/écrire les valeurs de variables
    static double getValue(const string& name);
    static void setValue(const string& name, double val);
    static void dumpMemory(); // affiche le contenu de la mémoire
};
```

```
// fichier variable.cc
map<string,double> Variable::_memory;

void Variable::dumpMemory() {
    map<string,double>::iterator it;

    cout << "Memory dump :" << endl;
    for (it = _memory.begin(); it != _memory.end(); it++)
        cout << "\t" << it->first << " = " << it->second << endl;
}
```

```
int main (int argc, char * const argv[])
{
    Mult      exp(new Const(2.0), new Cos(new Variable("x")));

    Variable::setValue("x", M_PI/3.0);
    cout << "exp = " << exp.eval() << endl;

    Variable   var_y("z", -11.7), var_z("y");

    Variable::dumpMemory();

    return 0;
}
```

en sortie

```
> exp = 1
> Memory dump :
>   x = 1.0472
>   y = 0
>   z = -11.7
```

Exercices :

- 1/ Ajouter une méthode de classe pour vider la mémoire à la fin du programme.
- 2/ Modifier le programme de façon à ce qu'une variable disparaisse de la mémoire s'il n'y a plus aucune référence vers elle...