Cours de C++

Template

Cécile Braunstein

cecile.braunstein@lip6.fr

Template functions

Objects of different types may nevertheless share common behavior

Generic functions

- Have one definition for a function family
- Parameters types and/or return type can be unknown
- Type is determined when the function is called

Example

```
#include <algorithm>

iterator find(iterator, iterator, val);
```

Works for any appropriate types in any kind of containers

How does it work ?

Language responsibilities

The ways parameter of unknown types are used constrain the parameter's type.

```
f(x,y)= x + y
```

- Requires that $+$ is defined for `x` and `y`
- When the function is called, the implementation check for the compatibility

STL responsibilities

When a generic function is defined with iterator.
$\Rightarrow$ Constraints the operation that the type support

First example

int

```
void Swap(int& a, int& b){
  int tmp = a;
  a = b;
  b = tmp;
}
```

class C

```
void Swap(C& a, C& b){
  C tmp = a;
  a = b;
  b = tmp;
}
```

double

```
void Swap(double& a, double& b){
  double tmp = a;
  a = b;
  b = tmp;
}
```

std::string

```
void Swap(std::string& a, std::
     string& b){
  std::string tmp = a;
  a = b;
  b = tmp;
}
```

Factoring

It's always the same code ! Only the type changes

➛ Use template

Factorization example

```
template<typename T>
void Swap(T& a, T& b){
  T tmp = a;
  a = b;
  b = tmp;
}
```

Syntax

```
template <class|typename type-param[=type], class|typename type-param
    [=type] ...>
ret-type namefuncition(param-list);
```

Template parameters

- Works like variable but for a type.
- Let us write programs in term of common behavior.
- In this context `class` and `typename` are equivalent.

More examples 1

Uses template functions

```
template<typename T>
T min (T tab[], int n) {
T min = tab[0]
for(int i = 0; i<n; i++)
  if(tab[i]<min) min = tab[i];
return min;
}
```

```
template<typename T1, typename T2>
int min (T1 ta[], T2 tb[], int n) {
int m = 0
for(int i = 0; i<n; i++)
  if(ta[i]<ta[m] || ta[i]==tb[m] && tb[i]<tb[m]) m = i ;
return m;
}
```

Syntax : examples

```
template<typename T, typename T2, typename T3>
void function(T1 arg, const T2& arg2, T2* t3)
{
  // Generic Algorithm
}
class A{};
int main()
{
  float x, y;
  function(1, x, &y); // automatic deduction/creation
  int m, n, p;
  A a;
  function(m, a, &a); // automatic deduction/creation
}
```

```
template<typename T, typename T1 = int, typename T3 = float>
void fonction(T1 arg, const T2& arg2, T2* t3)
{
  // generic algorithm
}
```

Syntaxe : examples (2)

```
template<typename T = float, typename X = float>
T function(int x)
{
  T t(x);
  return t;
}

int main()
{
  int x = function<int, int>(42);
  int y = function<int>(22);
  float v = function<>(24);
  return 0;
}
```

Instantiation

Instantiates a template

The implementation will effectively create and compile an instance of the function that replaces every use of `T` by `int`.

- Templates don't slow down the application speed.
- The more template instances there are, the bigger the application's code gets.
- The template code is not completely compiled before its use.
  - Errors may occur at link time
  - All types don't match for a given template
  - Be careful with the automatic conversion of types (*cast*)

Specialization - Constant parameters

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

template<int N, typename X>
void fill(X& x) {
  for (size_t i = 0; i < x.size(); ++i)
    x[i] = N;
}
int main()
{
  std::vector<float> v(10);
  fill<42, std::vector<float> >(v);
  fill<42>(v);// deduction du 2nd parametre
  std::cout << v[0] << std::endl;
}
```

Specialization - Complete instanciation

```
#include <iostream>
template<typename T> void print(const T& t) {
  std::cout << t << std::endl;
}
// specialization pour les caracteres
template<> void print(const char& t) {
  std::cout << "[" << t << "] code:" << (int) t
      << std::endl;
}
int main()
{
  float x = 42;
  char a = 42;
  print(x);
  print(a);
}
// sortie
// 42
// [*] code:42
```

Template Class

Definition

A template class is a type parametrized by others type and values known as compile time.

```
template<class Telem>
class Table {
  Telem *tab;
  int nb;
public:
Table(int n){tab = new Telem[nbr=n];}
~Table(){delete [] tab;}
}
```

Syntax : template class

```
template <class|typename type-param[=type], class|typename type-param
     [=type] ...>
class classname{
...
};
```

Methodology

- Declaration of template parameters
- Class Definition/Declaration

Instantiation

```
Table<int> myTab(10);
Table<Point> myPoints(20);
```

Member functions for template class

Definition

- They are template functions.
- They use the same parameters as the template class.
- If they are define outside the template class, parameters has to be re-called.

```
template<typename Telem>
class Table {
...
Telem min() const;
};

template<typename Telem>
Telem Table<Telem>::min() const{
  Telem m;
  ...
}
```

Typedef

The most usefull keywords for manipulating templates !

```
// Classical use
typedef int* iptr_t;
typedef unsigned int uint_t;

// When we can't do without
typedef std::vector<std::vector<int> > mat_t;
mat_t m;
mat_t::iterator it = m.begin();
// comparer avec :
std::vector<std::vector<int> >::iterator it = m.begin();
```

C++11 **auto**

```
// Classical use
auto a1 = 25; // The type of a1 is int
auto a2 = 'a'; // The type of a2 is char
std::vector<std::vector<int> > m;
auto it = m.begin(); //the type of it is an iterator on std::vector<
    std::vector<int> >
```

Typedef et typename

```cpp
template<typename T>
class Vecteur
{
public:
  typedef T type;
  Vecteur() :
    _data(new T[42]), _size(42) { }
  const T& operator[](size_t i) const
  { return _data[i];}
  size_t size() const { return _size;}
protected:
  T* _data;
  size_t _size;
};

template<typename V>
typename V::type maximum(const V& v) {
  typename V::type x = v[0];
  for (size_t i = 0; i < v.size(); ++i)
    x = std::max(x, v[i]);
  return x;
}
```

```cpp
// ..
int main() {
  Vecteur<float> v;
  std::cout
    << maximum(v)
    << std::endl;
  return 0;
}
```

Conclusion

- A powerful mechanism
- Another abstract form
- Fast execution time
- Slow Compile time
- Used a lot in modern c++

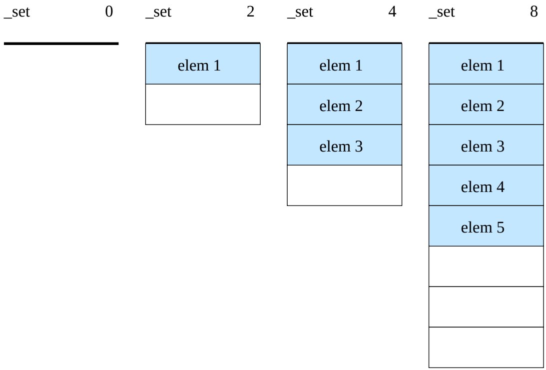
- ... but quickly complicated !
- ... quickly really complicated !

- note : Complete code must be only be in .hpp

Our own template class Set

- An element exists only one time in the set (unicity).
- The set manages the memory allocation according to the number of elements it contains.

The allocation policy is the following : every time the table is full the size is doubled. When elements are deleted, we need to manage the memory too.



Attributes - Constructor/destructors

Resize(int size) method

```
template<typename Element>
  void Set<Element>::_Resize ( int newsize )
```

HasElement() method

```
template<typename Element>
  bool Set<Element>::HasElement ( Element element ) const
```

AddElement(Telem element) method

```
template<typename Element>
 bool Set<Element>::AddElement ( Element element )
```

Delelement(Telem element) method

```
template<typename Element>
 bool Set<Element>::DelElement ( Element element )
```