Cours de C++

Fonctions génériques

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 in which uses a parameter of unknown types 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.
⇒ Constraints the operation that the type support

Syntax

```
template <class type-param [, class type-param] ...>
ret-type function-name(param-list)
```

Template parameters

- Works like variable but for a type
- Let us write programs in term of common behavior

First template function in C++

Exercice

Write a template function for the median function

```
template <class T>
T median(vector<T>)
{
  typedef typename vector<T>::sizetype vec_size;
  vec_size size = v.size();

  if(size==0)
   throw domain_error("median of an empty vector");
  sort(v.begin(),v.end());

  vec_sz mid = size/2;

  return size%2 == 0? (v[mid]+v[mid-1])/2 : v[mid];
}
```

First template function in C++

Exercice

Write a template function for the median function

```
template <class T>
T median(vector<T>)
{
  typedef typename vector<T>::sizetype vec_size;
  vec_size size = v.size();

  if(size==0)
    throw domain_error("median of an empty vector");
  sort(v.begin(),v.end());

  vec_sz mid = size/2;

  return size%2 == 0? (v[mid]+v[mid-1])/2 : v[mid];
}
```

Instantiation

```
vector<int> v;
...
int a = median(v);
```

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 run time
  - All types don't match for a given template
  - Be careful with the automatic conversion of types (*cast*)

Template functions for sequential containers

Algorithm standard library

Goal : Write function that deals with any values stored in any kind of containers.

Using iterators

```
find(c.begin(),c.end(),val)
```

- We can write a single function for any contiguous part of any containers.
- We can look in part of containers only.
- We can access element in different order.

Algorithms can be data-structure independent by using iterator.

Iterators and algorithms

Iterators particularities

- Containers don't support all the same operations
- Different Iterators offer different kinds of operations
- The library defines five iterator categories that corresponds to a specific collection of iterator operations.

Specification

- Correspond to a specific collection of iterator operations
- Classify the kind of iterator each containers provides
- Used by standard algorithm to specify which kind of iterator it expects
- Determine a strategy for accessing container elements

Iterators categories

Categories

1. Input iterator : Sequential access in one direction, input only
2. Output iterator : Sequential access in one direction, output only
3. Forward iterator : Sequential access in one direction, input and output
4. Bidirectional iterator : Sequential access in both direction, input and output
5. Random-access iterator : Efficient access to any element input and output

Example

```
template<class InputIterator, class T>
  InputIterator find ( InputIterator first,
      InputIterator last, const T& value )
```

Resume operations

category				characteristic	valid expressions
all categories				Can be copied and copy-constructed	T b(a) b = a
				Can be incremented	++a a++ *a++
RandomAccess	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	a == b a != b
				Can be dereferenced as an rvalue	*a a->m
			Output	Can be dereferenced to be the left side of an assignment operation	*a = t *a++ = t
				Can be default-constructed	T a T()
				Can be decremented	--a a-- *a--
				Supports arithmetic operators + and -	a + n n + a n - a a - b
				Supports inequality comparisons (< and >) between iterators	a < b a > b
				Supports compound assignment operations +=, -=, <= and >=	a += n a -= n a <= b a >= b
				Supports offset dereference operator ([])	a[n]