16. Templates

Hu Sikang
skhu@163.com

Contents

- Introduction templates
- Class temples
- Function templates

16.1 Introduction: templates

Why use Templates? **They are different interfaces only, but the algorithm are the same exactly. In fact we have to define many functions used function overloading.**

```
int abs(int x)
{
    return x >= 0 ? x : -x;
}

float fabs(float x)
{
   return x >= 0 ? x : -x;
}
```

```
double dabs(double x)
{
     return x >= 0 ? x : -x;
}
```

16.1 Introduction: templates

- Templates give us the means of *defining a family of functions or classes* that share the same functionality but which may differ with respect to the data type used internally.
- A class template is a *framework* for generating the source code for any number of related classes.
- A function template is a *framework* for generating related functions.

16.2 Class Templates

```
template <class T>
class class-name { …… }
```

```
template <typename T>
class class-name( …… }
```

- *T* is a template parameter.

16.2.1 Class Templates

One parameter in a template:

- **Declare and define an object:**

```
template <class T>
class MyClass {
    T  val;
    //……
}
```

```
class Student;

MyClass<int>  iObj;

MyClass<student>   studentObj;
```

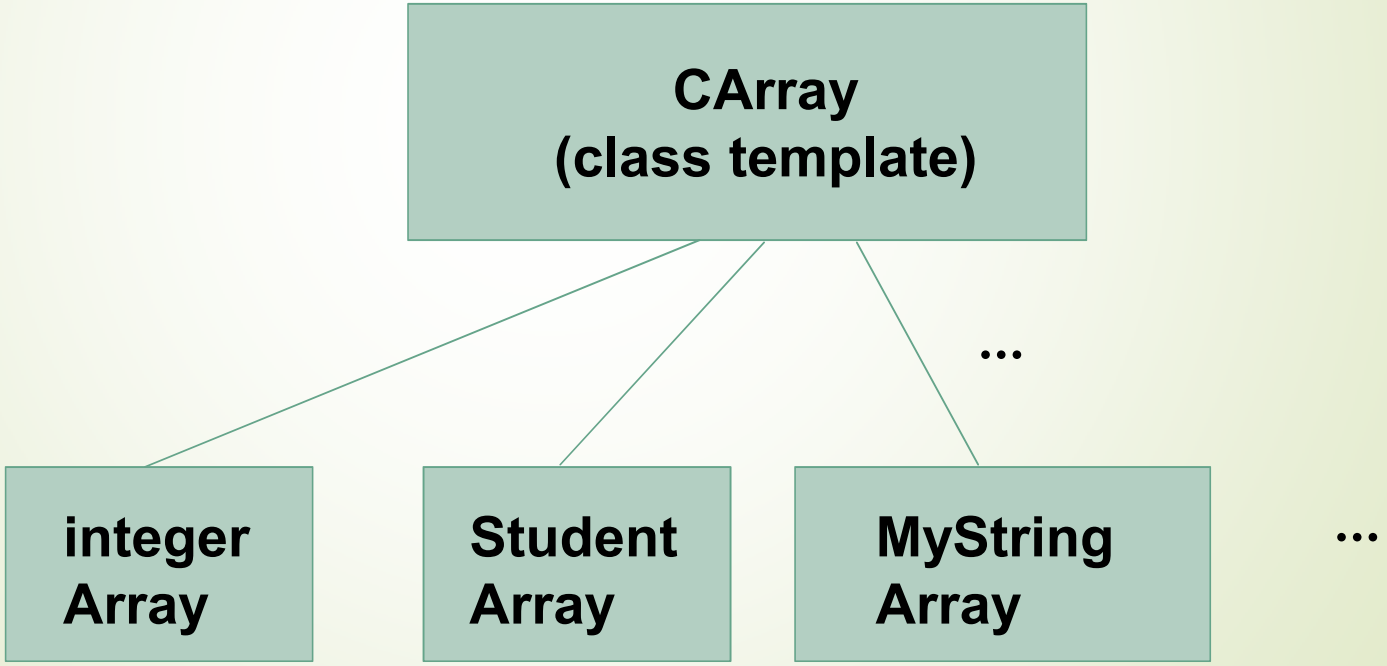
16.2.2 Class Templates

Multi parameters in a template:

```
template <class T1, class T2>
class Circle {
//...                                    // To distinguish different arguments
private:                                  //      between template and constructor
     T1  x, y;                           Circle <int, double>   circle(10, 20, 12.3);
     T2  radius;
public:
     T2 GetArea()    {  3.14 * radius * radius;  }
};
```

16.2.3 Class Templates

Generic Programming



Example: Array Class Template

```
#include <iostream>
using namespace std;
template <class T>
class Array {
public:
     Array( unsigned sz );
     ~Array();
     T& operator[ ](unsigned index);
private:
     T * values;
     unsigned size;
};
```

```
template<class T>
Array<T>::Array( unsigned sz )  {
    values = new T [sz];
    size = sz;
}

template<class T>
T & Array<T>::operator[ ] ( unsigned i )
{    return values[i];   }

template<class T> Array<T>::~Array()
{     if (values != NULL)  delete[ ]  values;   }
```

Example: Array Class Template

```
void main()
{
    Array<int>    dive(3);
    for (int i = 0; i < 3; i++)
        cin >> dive[i];
    for (i = 0; i < 3; i++)
        cout << dive[i] << endl;
}
```

How about Student?

16.3 Function Templates

A function can be defined in terms of an *unspecified type*.

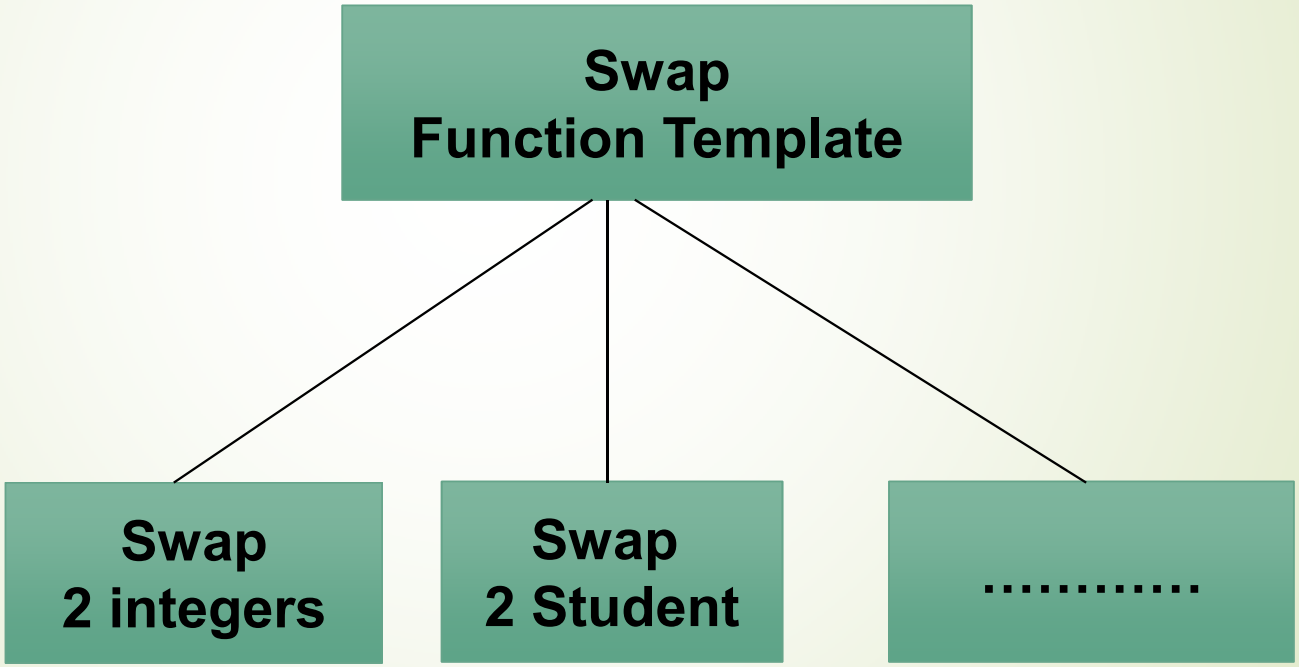
template <class T>
return-type function-name(T param)

template <typename T>
return-type function-name(T param)

- *one parameter function.*
- *T is called template parameter.*

16.3 Function Templates

generic



16.3 Example: Swap

```
#include <iostream.h>
template <class TParam>
void Swap( TParam & x,
           TParam & y )
{
  TParam temp;
  temp = x;
  x = y;
  y = temp;
}
```

```
class Student {
public:
  Student( unsigned id = 0 )
  {
        idnum = id;
  }
  int getID() {return idnum;};
private:
  unsigned idnum;
};
```

16.3 Example: Swap

```
int main() {
    int m = 10, n = 20;
    Student  S1(1234),  S2(5678);

    cout << m << " " << n << " " << endl;
    Swap( m, n );          // call with integers
    cout << m << " " << n << " " << endl;

    cout << S1.getID() << " " << S2.getID() << " " << endl;
    Swap( S1, S2 );        // call with Students
    cout << S1.getID() << " " << S2.getID() << " " << endl;

    return 0;
}
```

16.4 iterator in the template

What's the iterator?

An *iterator* is *an object* that moves through a container of other objects and selects them one at a time, without providing direct access to the implementation of that container.

***Iterators* provide a *standard way* to access elements, whether or not a container provides a way to access the elements directly.**

16.4 iterator in the template

Why not to overload copy constructor in the iterator which *p is defined?