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.

```
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>  intObj;

MyClass<student>   studentObj;
```

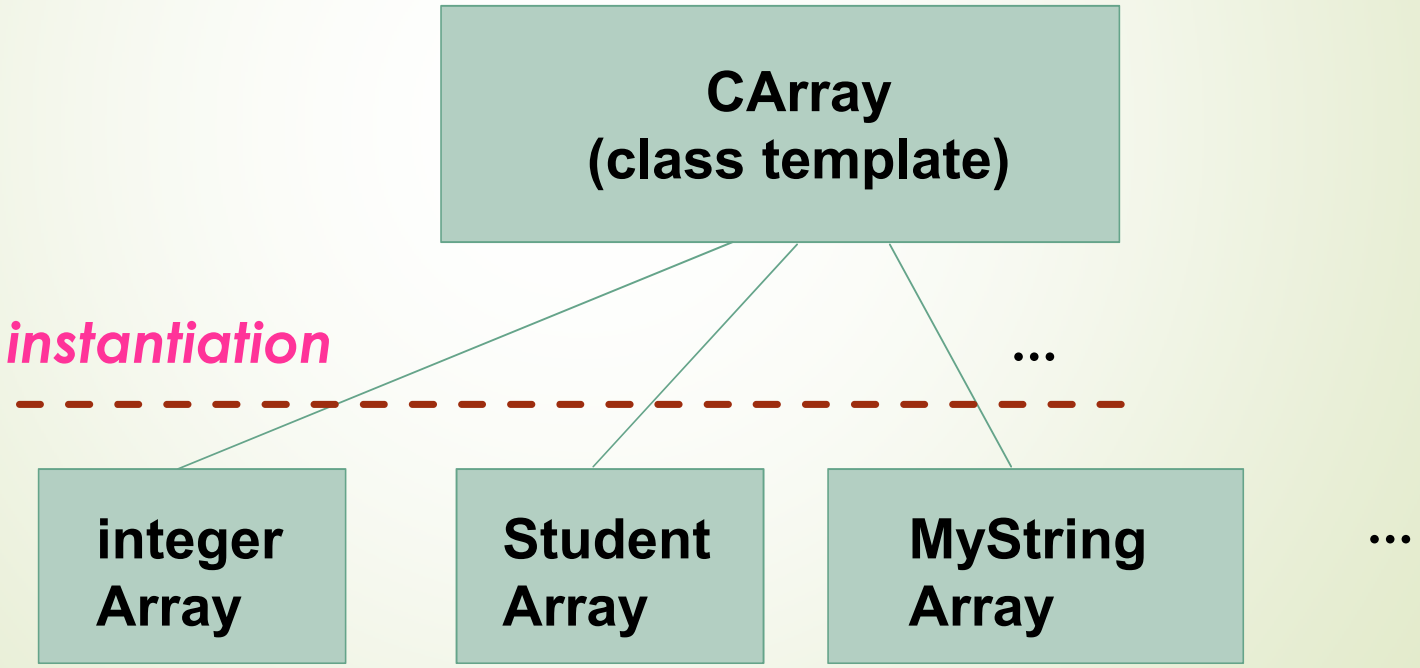
16.2.2 Class Templates

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 CArray {
public:
    CArray( unsigned sz );
    ~CArray();
    T& operator[ ](unsigned index);
private:
    T * value;
    unsigned size;
};
```

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

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

template<class T> CArray<T>::~CArray()
{    if (value)  delete[ ]  value;  }
```

Example: Array Class Template

```
int main()
{
    CArray<int>    dice(6);
    for (int i = 0; i < 6; i++)
        cin >> dice[i];
    for (i = 0; i < 6; i++)
        cout << dice[i] << endl;
    return 0;
}
```

```
int main()
{
    CArray<CMyString>    strArray(5);
    for (int i = 0; i < 5; i++)
        cin >> strArray [i];
    for (i = 0; i < 5; i++)
        cout << strArray [i] << endl;
    return 0;
}
```

16.3 Function Templates

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

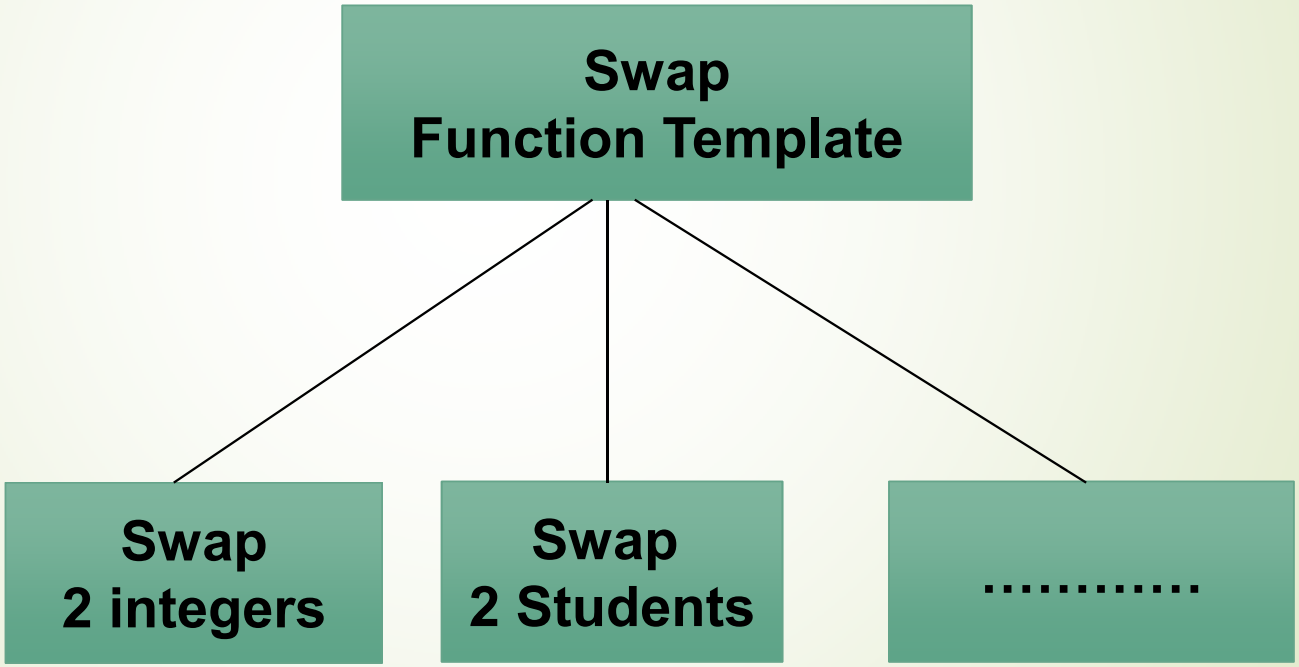
```
template <class T>
return-type function-name(T param)
```

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

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

16.3 Function Templates

Generic Programming



16.3 Example: Swap

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

```
class Student {
public:
    Student(const char* name) {
            Name = new char[strlen(name)+1];
            strcpy(Name, name);
     }
    ~Student()  {
            if (Name != nullptr)    {
                    delete[ ] Name;
                    Name = nullptr;
             }
     }
    const char* GetName()  {  return Name; }
private:     char* Name;
};
```

16.3 Example: Swap

```
int main() {
    int m = 10, n = 20;
    Student  S1("2022001"),  S2("2022002");

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

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

```
#include <iostream>
#include <vector>
using namespace std;
// routine use
int main()
{
    vector<int> vec = { 71, 23, 5, 68, 41 };
    for (int i = 0; i < vec.size(); i++)
        cout << vec[i] << ", ";
    cout << endl;

    return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;
// use with iterator
int main()
{
    vector<int> vec = { 71, 23, 5, 68, 41 };
    for (vector<int>::iterator it = vec.begin();
        it != vec.end();    ++it)
     {    cout << *it << ' ';      }

     return 0;
}
```

16.4 iterator in the template

```
class Iterator
{
public:    // Use this Iterator in the CMyString
        Iterator(CMyString& str, bool isEnd = false)
        {
                // 指向第一个元素，易于正向遍历
                p = str.m_pString;
                // 指向最后一个元素，易于逆向遍历
                if (isEnd)  p += str.m_iSize - 1;
        }
        Iterator operator++()       // overload prefix ++
        {
                p++;
                return *this;
        }
        Iterator operator++(int)    // overload postfix ++
        {
                Iterator temp(*this);
                p++;
                return temp;
        }
        Iterator operator--()       // overload prefix ++
        {
                p--;
                return *this;
        }
        Iterator operator--(int)    // overload postfix ++
        {
                Iterator temp(*this);
                p--;
                return temp;
        }
        bool operator< (const Iterator& it)
        {     return p < it.p + 1;      }
        char& operator*() { return *p; }  // If read only?
private:
        char* p;   // points to container's elements
};  // end of Iterator
```

16.4 iterator in the template

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