函数模板

1 定义

> 函数模板其实就是java中的泛型

函数模板不是一个实在的函数，编译器不能为其生成可执行代码。定义函数模板后只是一个对函数功能框架的描述，当它具体执行时，将根据传递的实际参数决定其功能。

2 用法:

面向对象的继承和多态机制有效提高了程序的可重用性和可扩充性。在程序的可重用性方面，程序员还希望得到更多支持。举一个最简单的例子，为了交换两个整型变量的值，需要写下面的 Swap 函数：

```
void Swap(int & x, int & y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

为了交换两个 double 型变量的值，还需要编写下面的 Swap 函数：

```
void Swap (double & xr double & y)
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

如果还要交换两个 char 型变量的值，交换两个 CStudent 类对象的值……都需要再编写 Swap 函数。而这些 Swap 函数除了处理的数据类型不同外，形式上都是一样的。能否只写一遍 Swap 函数，就能用来交换各种类型的变量的值呢？继承和多态显然无法解决这个问题。因此，“模板”的概念就应运而生了。

众所周知，有了“模子”后，用“模子”来批量制造陶瓷、塑料、金属制品等就变得容易了。程序设计语言中的模板就是用来批量生成功能和形式都几乎相同的代码的。有了模板，编译器就能在需要的时候，根据模板自动生成程序的代码。从同一个模板自动生成的代码，形式几乎是一样的。

3 函数模板的原理

C++ 语言支持模板。有了模板，可以只写一个 Swap 模板，编译器会根据 Swap 模板自动生成多个 Sawp 函数，用以交换不同类型变量的值。

在 C++ 中，模板分为函数模板和类模板两种。

- 函数模板是用于生成函数；
- 类模板则是用于生成类的。

函数模板的写法如下：

```
template <typename 类型参数1, typename 类型参数2, ...>
```

其中的 typename 关键字也可以用class 关键字替换，例如：

```
template <class 类型参数1, class类型参数2, ...>
返回值类型  模板名(形参表)
{
    函数体
}
```

函数模板看上去就像一个函数。前面提到的 Swap 模板的写法如下：

```
template <class T>
void Swap(T & x, T & y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
7
```

T 是类型参数，代表类型。编译器由模板自动生成函数时，会用具体的类型名对模板中所有的类型参数进行替换，其他部分则原封不动地保留。同一个类型参数只能替换为同一种类型。编译器在编译到调用函数模板的语句时，会根据实参的类型判断该如何替换模板中的类型参数。

例如下面的程序：

```
#include <iostream>
using namespace std;
template<class T>
void Swap(T & x, T & y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
int main()
{
    int n = 1, m = 2;
    Swap(n, m);  //编译器自动生成 void Swap (int &, int &)函数
    double f = 1.2, g = 2.3;
    Swap(f, g);  //编译器自动生成 void Swap (double &, double &)函数
    return 0;
}
```

编译器在编译到Swap(n, m);时找不到函数 Swap 的定义，但是发现实参 n、m 都是 int 类型的，用 int 类型替换 Swap 模板中的 T 能得到下面的函数：

```
void Swap (int & x, int & y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

该函数可以匹配Swap(n, m);这条语句。于是编译器就自动用 int 替换 Swap 模板中的 T，生成上面的 Swap 函数，将该 Swap 函数的源代码加入程序中一起编译，并且将Swap(n, m);编译成对自动生成的 Swap 函数的调用。

同理，编译器在编译到Swap(f, g);时会用 double 替换 Swap 模板中的 T，自动生成以下 Swap 函数：

```
void Swap(double & x, double & y)
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

然后再将Swap(f, g);编译成对该 Swap 函数的调用。

编译器由模板自动生成函数的过程叫模板的实例化。由模板实例化而得到的函数称为模板函数。在某些编译器中，模板只有在被实例化时，编译器才会检查其语法正确性。如果程序中写了一个模板却没有用到，那么编译器不会报告这个模板中的语法错误。

编译器对模板进行实例化时，并非只能通过模板调用语句的实参来实例化模板中的类型参数，模板调用语句可以明确指明要把类型参数实例化为哪种类型。可以用：

```
模板名<实际类型参数1, 实际类型参数2, ...>
```

的方式告诉编译器应该如何实例化模板函数。例如下面的程序：

```
#include <iostream>
using namespace std;
template <class T>
T Inc(int n)
{
    return 1 + n;
}
int main()
{
    cout << Inc<double>(4) / 2;
    return 0;
}
```

Inc(4)指明了此处实例化的模板函数原型应为：

```
double Inc(double);
```

编译器不会因为实参 4 是 int 类型，就生成原型为 int Inc(int) 的函数。因此，上面程序输出的结果是 2.5 而非 2。

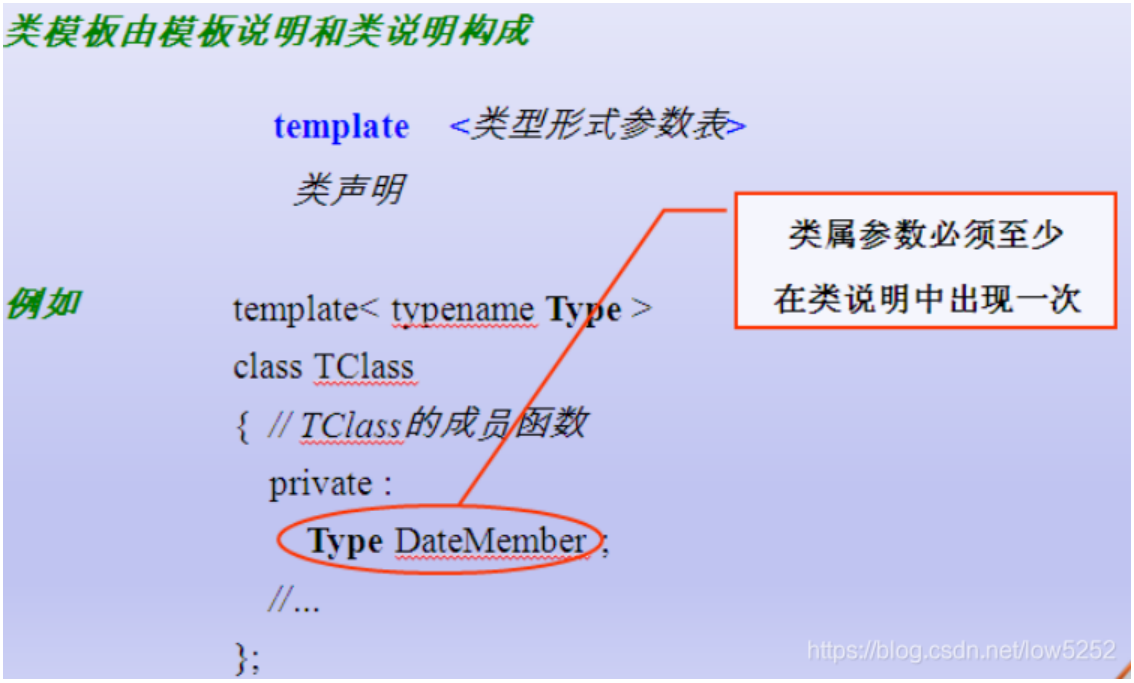
函数模板中可以有不止一个类型参数。例如，下面这个函数模板的写法是合法的：

```
template <class T1, class T2>
T2 print(T1 arg1, T2 arg2)
{
    cout << arg1 << " " << arg2 << endl;
    return arg2;
}
```

4 延申用法

4.1为什么需要类模板

- 类模板与函数模板的定义和使用类似，我们已经进行了介绍。 有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同，如下面语句声明了一个类：



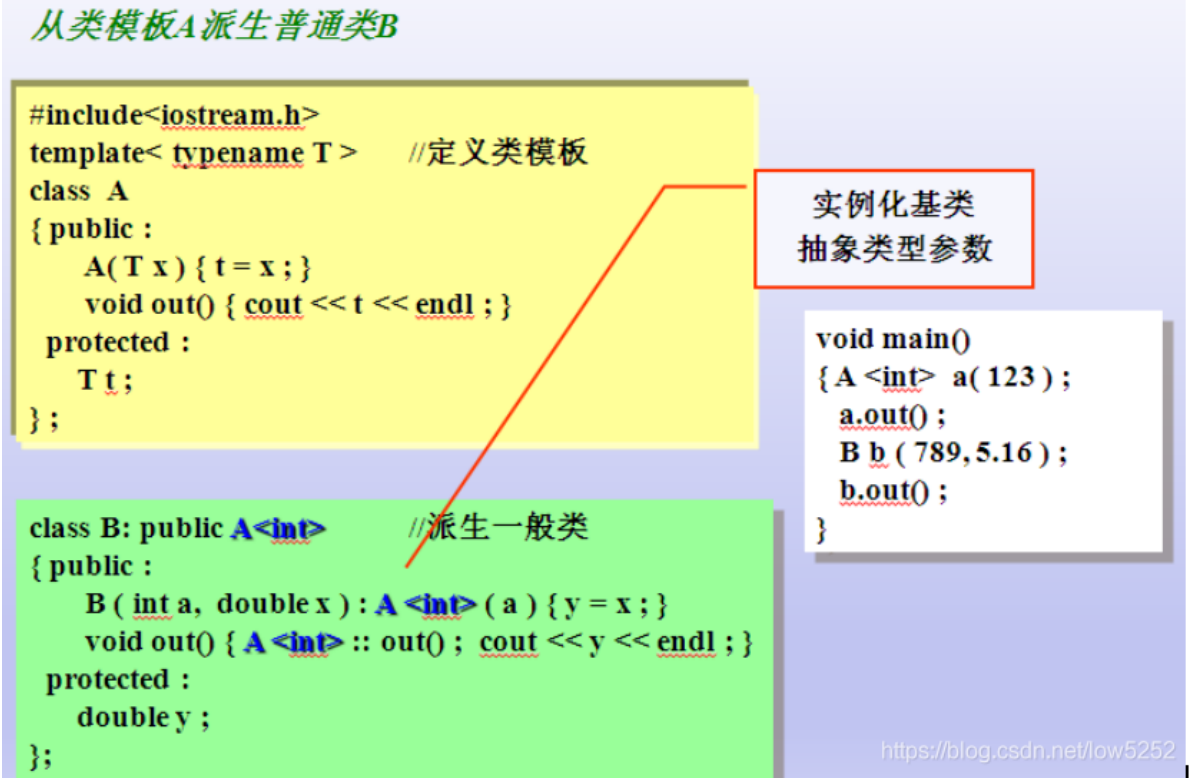
- 类模板用于实现类所需数据的类型参数化
- 类模板在表示如数组、表、图等数据结构显得特别重要，
- 这些数据结构的表示和算法不受所包含的元素类型的影响

4.2单个类模板语法

```
1 //类的类型参数化 抽象的类
2 //单个类模板
3 template<typename T>
4 class A
5 {
6 public:
7     A(T t)
8     {
9         this->t = t;
10    }
11
```

```
12      T &getT()
13      {
14          return t;
15      }
16  protected:
17  public:
18      T t;
19  };
20  void main()
21  {
22     //模板了中如果使用了构造函数,则遵守以前的类的构造函数的调用规则
23      A<int>  a(100);
24      a.getT();
25      printAA(a);
26      return ;
27  }
18192021222324252627
```

4.3继承中的类模板语法



案例1：

```
1  //结论: 子类从模板类继承的时候,需要让编译器知道 父类的数据类型具体是什么(数据类型的本质:固定大小内存块的别名)A<int>
2  //
3  class B : public A<int>
4  {
5  public:
6      B(int i) : A<int>(i)
7      {
8
9      }
10     void printB()
11     {
```

```
        cout<<"A:"<<t<<endl;
    }
protected:
private:
};

//模板与上继承
//怎么样从基类继承
//若基类只有一个带参数的构造函数,子类是如何启动父类的构造函数
void pintBB(B &b)
{
    b.printB();
}
void printAA(A<int> &a)  //类模板做函数参数
{
     //
    a.getT();
}

void main()
{
    A<int>  a(100); //模板了中如果使用了构造函数,则遵守以前的类的构造函数的调用规则
    a.getT();
    printAA(a);

    B b(10);
    b.printB();


    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

案例2:

```
#include<iostream>
using namespace std;
//A编程模板类--类型参数化
/*
类模板的定义 类模板的使用 类模板做函数参数
*/
template <typename T>
class A
{
public:
    A(T a = 0)
    {
        this->a = a;
    }
public:
    void printA()
    {
        cout << "a:" << a << endl;
    }
protected:
    T a;
```

```
private:

};
//从模板类派生时，需要具体化模板类，C++编译器需要知道父类的数据类型是什么样子的
//要知道父类所占的内存多少
class B :public A<int>
{
public:
    B(int a =10, int b =20):A<int>(a)
    {
        this->b = b;
    }
    void printB()
    {
        cout << "a:" << a << "b:" << b << endl;
    }
protected:
private:
    int b;

};
//从模板类派生模板类
template <typename T>
class C :public A<T>
{

public:
    C(T c,T a) : A<T>(a)
    {
        this->c = c;
    }
    void printC()
    {
        cout << "c:" << c << endl;
    }
protected:
    T c;
private:

};

void main()
{
    //B b1(1, 2);
    //b1.printB();
    C<int> c1(1,2);
    c1.printC();
}
```

4.4类模板的基础语法

```
#include<iostream>
using namespace std;
//A编程模板类--类型参数化
/*
    类模板的定义 类模板的使用 类模板做函数参数
*/
```

```
template <typename T>
class A
{
public:
    A(T a = 0)
    {
        this->a = a;
    }
public:
    void printA()
    {
        cout << "a:" << a << endl;
    }
protected:
private:
    T a;
};
//参数 C++编译器具体的类
void UseA(A<int> &a)
{
    a.printA();
}
void main()
{
    //模板类本身就是抽象的，具体的类，具体的变量
    A<int> a1(11),a2(22),a3(33);//模板类是抽象的，  需要类型具体化
    //a1.printA();

    UseA(a1);
    UseA(a2);
    UseA(a3);
}
18192021222324252627282930313233343536373738
```

5类模板语法知识体系梳理

5.1.所有的类模板函数写在类的内部

代码:

复数类:

```
#include<iostream>
using namespace std;
template <typename T>
class Complex
{
public:
    friend Complex MySub(Complex &c1, Complex &c2)
    {
        Complex tmp(c1.a-c2.a, c1.b-c2.b);
        return tmp;
    }

    friend ostream & operator<< (ostream &out, Complex &c3)
    {
```

```
15          out << c3.a << "+" << c3.b <<"i"<< endl;
16          return out;
17      }
18      Complex(T a, T b)
19      {
20          this->a = a;
21          this->b = b;
22      }
23      Complex operator+(Complex &c2)
24      {
25          Complex tmp(a + c2.a, b + c2.b);
26          return tmp;
27      }
28      void printCom()
29      {
30          cout << "a:" << a << " b:" << b << endl;
31      }
32  protected:
33  private:
34      T a;
35      T b;
36  };
37
38  /*
39      重载运算符的正规写法：
40      重载左移<<  右移>> 只能用友元函数，其他的运算符重载都要用成员函数，不要滥用友元函数
41  */
42  //ostream & operator<< (ostream &out, Complex &c3)
43  //{
44  //    out<< "a:" << c3.a << " b:" << c3.b << endl;
45  //    return out;
46  //}
47  void main()
48  {
49      Complex<int>      c1(1,2);
50      Complex<int>      c2(3, 4);
51
52      Complex<int> c3 = c1 + c2;//重载加号运算符
53
54      c3.printCom();
55
56      //重载左移运算符
57      cout << c3 << endl;
58
59      {
60          Complex<int> c4 = MySub(c1 , c2);
61
62          cout << c4 << endl;
63      }
64      system("pause");
65  }
```

```
1819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364 65
```

5.2.所有的类模板函数写在类的外部，在一个cpp中

注意：

复制代码
//构造函数 没有问题

- 普通函数 没有问题
- 友元函数：用友元函数重载 << >>

  friend ostream& operator<< (ostream &out, Complex &c3) ;
- 友元函数：友元函数不是实现函数重载（非 << >>）

1 需要在类前增加 类的前置声明 函数的前置声明

```
template<typename T>

class Complex;

template<typename T>

Complex<T> mySub(Complex<T> &c1, Complex<T> &c2);
7
```

2 类的内部声明 必须写成:

```
friend Complex<T> mySub <T> (Complex<T> &c1, Complex<T> &c2);
1
```

3 友元函数实现 必须写成：

```
template<typename T>

  Complex<T> mySub(Complex<T> &c1, Complex<T> &c2)

{

Complex<T> tmp(c1.a - c2.a, c1.b-c2.b);

return tmp;

}
7891011
```

4 友元函数调用 必须写成

```
Complex<int> c4 = mySub<int>(c1, c2);

cout<<c4;
```

结论：友元函数只用来进行 左移 友移操作符重载。

复数类：

> 代码：

```cpp
#include<iostream>
using namespace std;

template<typename T>
class Complex;
template<typename T>
Complex<T> mySub(Complex<T> &c1, Complex<T> &c2);

template <typename T>
class Complex
{
public:
    friend Complex<T> mySub <T>(Complex<T> &c1, Complex<T> &c2);

    friend ostream & operator<< <T>(ostream &out, Complex &c3);
    Complex(T a, T b);
    void printCom();
    Complex operator+(Complex &c2);
    Complex operator-(Complex &c2);

protected:
private:
    T a;
    T b;
};

//构造函数的实现，写在了外部
template <typename T>
Complex<T>::Complex(T a, T b)
{
    this->a = a;
    this->b = b;
}

template <typename T>
void Complex<T>::printCom()
{
    cout << "a:" << a << " b:" << b << endl;
}
//成员函数实现加号运算符重载
template <typename T>
Complex<T> Complex<T>::operator+(Complex<T> &c2)
{
    Complex tmp(a + c2.a, b + c2.b);
    return tmp;
}
template <typename T>
Complex<T> Complex<T>::operator-(Complex<T> &c2)
{
    Complex(a-c2.a,a-c2.b);
    return tmp;
}
//友元函数实现<<左移运算符重载

/*
严重性    代码    说明    项目    文件    行    禁止显示状态
错误    C2768    “operator <<”: 非法使用显式模板参数    泛型编程课堂操练

```

```
错误的本质：两次编译的函数头，第一次编译的函数头，和第二次编译的函数有不一样
*/
template <typename T>
ostream & operator<< (ostream &out, Complex<T> &c3)//不加T
{
    out << c3.a << "+" << c3.b << "i" << endl;
    return out;
}


//
template <typename T>
Complex<T> mySub(Complex<T> &c1, Complex<T> &c2)
{
    Complex<T> tmp(c1.a - c2.a, c1.b - c2.b);
    return tmp;
}

void main()
{
    Complex<int>     c1(1, 2);
    Complex<int>     c2(3, 4);

    Complex<int> c3 = c1 + c2;//重载加号运算符

    c3.printCom();

    //重载左移运算符
    cout << c3 << endl;

    {
        Complex<int> c4 = mySub<int>(c1, c2);

        cout << c4 << endl;
    }
    system("pause");
}
```

所有的类模板函数写在类的外部，在不同的.h和.cpp中
也就是类模板函数说明和类模板实现分开

//类模板函数

构造函数

普通成员函数

友元函数

用友元函数重载<<>>;

用友元函数重载非<< >>

demo_complex.cpp

```
#include"demo_09complex.h"
```

```
#include<iostream>
using namespace std;

template <typename T>
Complex<T>::Complex(T a, T b)
{
    this->a = a;
    this->b = b;
}

template <typename T>
void Complex<T>::printCom()
{
    cout << "a:" << a << " b:" << b << endl;
}
//成员函数实现加号运算符重载
template <typename T>
Complex<T> Complex<T>::operator+(Complex<T> &c2)
{
    Complex tmp(a + c2.a, b + c2.b);
    return tmp;
}
//template <typename T>
//Complex<T> Complex<T>::operator-(Complex<T> &c2)
//{
//    Complex(a - c2.a, a - c2.b);
//    return tmp;
//}
template <typename T>
ostream & operator<< (ostream &out, Complex<T> &c3)//不加T
{
    out << c3.a << "+" << c3.b << "i" << endl;
    return out;
}


//
//template <typename T>
//Complex<T> mySub(Complex<T> &c1, Complex<T> &c2)
//{
//    Complex<T> tmp(c1.a - c2.a, c1.b - c2.b);
//    return tmp;
//}
18192021222324252627282930313233343536373839404142434
```

demo_09complex.h

```
#pragma once
#include<iostream>
using namespace std;
template <typename T>
class Complex
{
public:
    //friend Complex<T> mySub <T>(Complex<T> &c1, Complex<T> &c2);

    friend ostream & operator<< <T>(ostream &out, Complex &c3);
```

```
    Complex(T a, T b);
    void printCom();
    Complex operator+(Complex &c2);
    //Complex operator-(Complex &c2);

protected:
private:
    T a;
    T b;
};
```

demo_09complex_text.cpp

```
#include"demo_09complex.h"
#include"demo_09complex.cpp"

#include<iostream>
using namespace std;

void main()
{
    Complex<int>     c1(1, 2);
    Complex<int>     c2(3, 4);

    Complex<int> c3 = c1 + c2;//重载加号运算符

    c3.printCom();

    //重载左移运算符
    cout << c3 << endl;

    /*{
        Complex<int> c4 = mySub<int>(c1, c2);

        cout << c4 << endl;
    }*/
    system("pause");
}
```

5.3总结

归纳以上的介绍，可以这样声明和使用类模板：

1. 先写出一个实际的类。由于其语义明确，含义清楚，一般不会出错。
2. 将此类中准备改变的类型名(如int要改变为float或char)改用一个自己指定的虚拟类型名(如上例中的numtype)。
3. 在类声明前面加入一行，格式为：

```
template <class 虚拟类型参数>

 如：

      template <class numtype> //注意本行末尾无分号

      class Compare

         {…}; //类体
```

1. 用类模板定义对象时用以下形式：

```
类模板名<实际类型名> 对象名;

      类模板名<实际类型名> 对象名(实参表列);

  如：

      Compare<int> cmp;

      Compare<int> cmp(3,7);
789
```

1. 如果在类模板外定义成员函数，应写成类模板形式：

```
template <class 虚拟类型参数>

     函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {…}
123
```

5.4 关于类模板的几点说明：

1. 类模板的类型参数可以有一个或多个，每个类型前面都必须加class，如：

   ```
   template <class T1,class T2>

    class someclass

    {…};
   12345
   ```

   在定义对象时分别代入实际的类型名，如：

   someclass<int,double> obj;
2. 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。
3. 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

6 类模板中的static关键字

从类模板实例化的每个模板类有自己的类模板数据成员，该模板类的所有对象共享一个static数据成员和非模板类的static数据成员一样，模板类的static数据成员也应该在文件范围定义和初始化
每个模板类有自己的类模板的static数据成员副本

```
#include<iostream>
using namespace std;
template <typename T>
class AA
{
public:
    static T m_a;
protected:
private:
};
template <typename T>
T AA<T>::m_a =0;
void main()
{
    AA<int> a1, a2, a3;
    a1.m_a = 10;
    a2.m_a++;
    a3.m_a++;
    cout << AA<int>::m_a << endl;

    AA<char> b1, b2, b3;
    b1.m_a = 'a';
    b2.m_a++;
    b3.m_a++;
    cout << AA<char>::m_a << endl;

    //m_a是每个类型的类，去使用，手工写两个类 int  char
    system("pause");
}
```

> **案例2：以下来自：C++类模板遇上static关键字**

```
#include <iostream>
using namespace std;

template<typename T>
class Obj{
public:
    static T m_t;
};

template<typename T>
T Obj<T>::m_t = 0;

int main04(){
    Obj<int> i1,i2,i3;
    i1.m_t = 10;
    i2.m_t++;
    i3.m_t++;
```

```
18      cout << Obj<int>::m_t<<endl;
19
20      Obj<float> f1,f2,f3;
21      f1.m_t = 10;
22      f2.m_t++;
23      f3.m_t++;
24      cout << Obj<float>::m_t<<endl;
25
26      Obj<char> c1,c2,c3;
27      c1.m_t = 'a';
28      c2.m_t++;
29      c3.m_t++;
30      cout << Obj<char>::m_t<<endl;
31  }
```

当类模板中出现static修饰的静态类成员的时候，我们只要按照正常理解就可以了。static的作用是将类的成员修饰成静态的，所谓的静态类成员就是指类的成员为类级别的，不需要实例化对象就可以使用，而且类的所有对象都共享同一个静态类成员，因为类静态成员是属于类而不是对象。那么，类模板的实现机制是通过二次编译原理实现的。c++编译器并不是在第一个编译类模板的时候就把所有可能出现的类型都分别编译出对应的类（太多组合了），而是在第一个编译的时候编译一部分，遇到泛型不会替换成具体的类型（这个时候编译器还不知道具体的类型），而是在第二次编译的时候再将泛型替换成具体的类型（这个时候编译器知道了具体的类型了）。由于类模板的二次编译原理再加上static关键字修饰的成员，当它们在一起的时候实际上一个类模板会被编译成多个具体类型的类，所以，不同类型的类模板对应的static成员也是不同的（不同的类），但相同类型的类模板的static成员是共享的（同一个类）。

7 类模板在项目开发中的应用

小结

> 1. 模板是C++类型参数化的多态工具。C++提供函数模板和类模板。
> 2. 模板定义以模板说明开始。类属参数必须在模板定义中至少出现一次。
> 3. 同一个类属参数可以用于多个模板。
> 4. 类属参数可用于函数的参数类型、返回类型和声明函数中的变量。
> 5. 模板由编译器根据实际数据类型实例化，生成可执行代码。实例化的函数。
> 6. 模板称为模板函数；实例化的类模板称为模板类。
> 7. 函数模板可以用多种方式重载。
> 8. 类模板可以在类层次中使用 。

8 训练题

8.1 请设计一个数组模板类（MyVector），完成对int、char、Teacher类型元素的管理。

设计:

- 类模板 构造函数 拷贝构造函数 << [] 重载=操作符
- a2=a1
- 实现

8.2 请仔细思考：

a) 如果数组模板类中的元素是Teacher元素时，需要Teacher类做什么工作

b) 如果数组模板类中的元素是Teacher元素时，Teacher类含有指针属性哪?

```
class Teacher
{
    friend ostream & operator<<(ostream &out, const Teacher &obj);
public:
    Teacher(char *name, int age)
    {
        this->age = age;
        strcpy(this->name, name);
    }

    Teacher()
    {
        this->age = 0;
        strcpy(this->name, "");
    }

private:
    int age;
    char name[32];
};


class Teacher
{
    friend ostream & operator<<(ostream &out, const Teacher &obj);
public:
    Teacher(char *name, int age)
    {
        this->age = age;
        strcpy(this->name, name);
    }

    Teacher()
    {
        this->age = 0;
        strcpy(this->name, "");
    }

private:
    int age;
    char *pname;
};
```

结论1： 如果把Teacher放入到MyVector数组中，并且Teacher类的属性含有指针，就是出现深拷贝和浅拷贝的问题。

结论2：需要Teacher封装的函数有：

1. 重写拷贝构造函数
2. 重载等号操作符

3. 重载左移操作符。

理论提高：

所有容器提供的都是值（value）语意，而非引用（reference）语意。容器执行插入元素的操作时，内部实施拷贝动作。所以STL容器内存储的元素必须能够被拷贝（必须提供拷贝构造函数）。

8.3 请从数组模板中进行派生

```
//演示从模板类 派生 一般类
#include "MyVector.cpp"

class MyArray01 : public MyVector<double>
{
public:
    MyArray01(int len) : MyVector<double>(len)
    {
        ;
    }
protected:
private:
};


//演示从模板类 派生 模板类 //BoundArray
template <typename T>
class MyArray02 : public MyVector<T>
{
public:
    MyArray02(int len) : MyVector<double>(len)
    {
        ;
    }
protected:
private:
};
测试案例：

//演示 从模板类 继承 模板类
void main()
{
    MyArray02<double> dArray2(10);
    dArray2[1] = 3.15;

}


//演示 从模板类 继承 一般类
void main11()
{
    MyArray01 d_array(10);

    for (int i=0; i<d_array.getLen(); i++)
    {
        d_array[i] = 3.15;
    }

    for (int i=0; i<d_array.getLen(); i++)
```

```
    {
        cout << d_array[i] << " ";
    }

    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

```
18192021222324252627282930313233343536373839404142434445464748495051525354555657
```

8.4 作业:

封装你自己的数组类；设计被存储的元素为类对象；

思考：类对象的类，应该实现的功能。

1. 优化Teacher类, 属性变成 char *panme, 构造函数里面 分配内存
2. 优化Teacher类,析构函数 释放panme指向的内存空间
3. 优化Teacher类,避免浅拷贝 重载= 重写拷贝构造函数
4. 优化Teacher类,在Teacher增加 <<
5. 在模板数组类中,存int char Teacher Teacher*(指针类型)

zuoye.h

```
#pragma once

template <typename T>
class MyVector
{

    //friend ostream & operator<< <T>(ostream &out, const MyVector &obj);
public:
    MyVector(int size = 0);//构造函数
    MyVector(const MyVector &obj);//copy构造函数
    ~MyVector();
public:
    T& operator [](int index);
    MyVector &operator=(const MyVector &obj);


    int getLen()
    {
        return m_len;
    }
protected:
private:
    T *m_space;
    int m_len;

};
```

zuoye_test12.cpp

```
#include"zuoye.h"
#include"zuoye12.cpp"
#include<iostream>
```

```
using namespace std;
class Teacher
{
public:
    Teacher()
    {
        age = 33;
        m_p = new char[1];
        strcpy(m_p, " ");
    }

    Teacher(char *name, int age)
    {
        this->age = age;
        m_p = new char[strlen(name)+1];
        strcpy(this->m_p, name);

    }
    Teacher(const Teacher &obj)
    {
        m_p = new char[strlen(obj.m_p) + 1];
        strcpy(this->m_p, obj.m_p);
        age = obj.age;
    }
    ~Teacher()
    {
        if (m_p!=NULL)
        {
            delete[] m_p;
            m_p = NULL;
        }
    }
    void printT()
    {
        cout << m_p << ", " << age;
    }
public:
    //重载<< ==
    friend ostream & operator<<(ostream &out,Teacher &t);
    Teacher & operator=(const Teacher &obj)
    {
        //1
        if (m_p!=NULL)
        {
            delete[] m_p;
            m_p = NULL;
            age = 33;
        }
        //2
        m_p = new char[strlen(obj.m_p) + 1];
        age = obj.age;
        //3
        strcpy(this->m_p, obj.m_p);
        return *this;
    }
protected:
private:
    int age;
```

```
62      //char name[32];
63      char *m_p;
64  };
65  ostream & operator<<(ostream &out, Teacher &t)
66  {
67      out << t.m_p << ", " << t.age << endl;
68      return out;
69  }
70
71  void main()
72  {
73      Teacher t1("t1", 31), t2("t2", 32);
74
75      MyVector<Teacher *> Tarray(2);
76
77      Tarray[0] = &t1;
78      Tarray[1] = &t2;
79      for (int i = 0; i < 2; i++)
80      {
81          Teacher *tmp = Tarray[i];
82          tmp->printT();
83      }
84      system("pause");
85  }
86  void main123()
87  {
88      Teacher t1("t1", 31), t2("t2", 32);
89      MyVector<Teacher> Tarray(2);
90      Tarray[0] = t1;
91      Tarray[1] = t2;
92      for (int i = 0; i < 2; i++)
93      {
94          Teacher tmp = Tarray[i];
95          tmp.printT();
96      }
97      system("pause");
98  }
99  void main112()
100 {
101     MyVector<int>  myv1(10);
102     myv1[0] = 'a';
103     myv1[1] = 'b';
104     myv1[2] = 'c';
105     myv1[3] = 'd';
106     myv1[4] = 'e';
107     //cout << myv1;
108     MyVector<int>  myv2 = myv1;
109 }
110
111
112 void main111()
113 {
114     MyVector<int>  myv1(10);
115     for (int i = 0; i < myv1.getLen(); i++)
116     {
117         myv1[i] = i + 1;
118         cout << myv1[i] << " ";
119     }
```

```
120
121     MyVector<int>  myv2 = myv1;
122     for (int i = 0; i < myv2.getLen(); i++)
123     {
124         myv2[i] = i + 1;
125         cout << myv2[i] << " ";
126     }
127
128     //cout << myv2 << endl;//重载左移运算符
129
130     system("pause");
131 }
```

zuoye12.cpp

```
1  #include"zuoye.h"
2  #include<iostream>
3  using namespace std;
4
5  template <typename T>
6  ostream & operator<<(ostream &out, const MyVector<T> &obj)
7  {
8      for (int i = 0; i<obj.m_len; i++)
9      {
10         out << obj.m_space[i] << " ";
11     }
12     out << endl;
13     return out;
14 }
15 //构造函数
16 template <typename T>
17 MyVector<T>::MyVector(int size = 0)
18 {
19     m_space = new T[size];
20     m_len = size;
21 }
22 //MyVector<int>  myv2 = myv1;
23 template <typename T>
24 MyVector<T>::MyVector(const MyVector &obj)
25 {
26     //根据大小分配内存
27     m_len = obj.m_len;
28     m_space = new T[m_len];
29     //copy数据
30     for (int i = 0; i<m_len; i++)
31     {
32         m_space[i] = obj.m_space[i];
33     }
34 }
35 template <typename T>
36 MyVector<T>::~MyVector()
37 {
38     if (m_space != NULL)
39     {
40         delete[] m_space;
41         m_space = NULL;
```

```
        m_len = 0;

    }
}
template <typename T>
T& MyVector<T>::operator [](int index)
{
    return m_space[index];
}
template <typename T>
MyVector<T> & MyVector<T>::operator=(const MyVector<T> &obj)
{
    //先把a2的内存释放掉
    if (m_space != NULL)
    {
        delete[] m_space;
        m_space = NULL;
        m_len = 0;

    }

    //根据a1分配内存
    m_len = obj.m_len;
    m_space = new T[m_len];

    //copy数据
    for (i = 0; i<m_len; i += )
    {
        m_space[i] = obj.m_space[i];
    }
    return *this;//a2= a1 返回a2的自身
}
```

9 字符串

string 类的成员函数有很多，同一个名字的函数也常会有五六个重载的版本。篇幅所限，不能将这些原型一一列出并加以解释。这里仅对常用成员函数按功能进行分类，并直接给出应用的例子，通过例子，读者可以基本掌握这些成员函数的用法。

要想更深入地了解 string 类，还要阅读 C++ 的参考手册或编译器自带的联机资料。对于前面介绍过的字符串处理的内容，这里不再重复说明。

1. 构造函数

string 类有多个构造函数，用法示例如下：

```
string s1();  // si = ""
string s2("Hello");  // s2 = "Hello"
string s3(4, 'K');  // s3 = "KKKK"
string s4("12345", 1, 3);  //s4 = "234"，即 "12345" 的从下标 1 开始，长度为 3 的子串
```

为称呼方便，本教程后文将从字符串下标 n 开始、长度为 m 的字符串称为“子串(n, m)”。

string 类没有接收一个整型参数或一个字符型参数的构造函数。下面的两种写法是错误的：

```
string s1('K');
string s2(123);
```

2. 对 string 对象赋值

可以用 char* 类型的变量、常量，以及 char 类型的变量、常量对 string 对象进行赋值。例如：

```
string s1;
s1 = "Hello";  // s1 = "Hello"
s2 = 'K';  // s2 = "K”
```

string 类还有 assign 成员函数，可以用来对 string 对象赋值。assign 成员函数返回对象自身的引用。例如：

```
string s1("12345"), s2;
s3.assign(s1);  // s3 = s1
s2.assign(s1, 1, 2);  // s2 = "23"，即 s1 的子串(1, 2)
s2.assign(4, 'K');  // s2 = "KKKK"
s2.assign("abcde", 2, 3);  // s2 = "cde"，即 "abcde" 的子串(2, 3)
```

3. 求字符串的长度

length 成员函数返回字符串的长度。size 成员函数可以实现同样的功能。

4. string对象中字符串的连接

除了可以使用`+`和`+=`运算符对 string 对象执行字符串的连接操作外，string 类还有 append 成员函数，可以用来向字符串后面添加内容。append 成员函数返回对象自身的引用。例如：

```
string s1("123"), s2("abc");
s1.append(s2);  // s1 = "123abc"
s1.append(s2, 1, 2);  // s1 = "123abcbc"
s1.append(3, 'K');  // s1 = "123abcbcKKK"
s1.append("ABCDE", 2, 3);  // s1 = "123abcbcKKKCDE"，添加 "ABCDE" 的子串(2, 3)
```

5. string对象的比较

除了可以用 <、<=、==、!=、>=、> 运算符比较 string 对象外，string 类还有 compare 成员函数，可用于比较字符串。compare 成员函数有以下返回值：

- 小于 0 表示当前的字符串小；
- 等于 0 表示两个字符串相等；
- 大于 0 表示另一个字符串小。

例如：

```
string s1("hello"), s2("hello, world");
int n = s1.compare(s2);
n = s1.compare(1, 2, s2, 0, 3);  //比较s1的子串 (1,2) 和s2的子串 (0,3)
n = s1.compare(0, 2, s2);  // 比较s1的子串 (0,2) 和 s2
n = s1.compare("Hello");
n = s1.compare(1, 2, "Hello");  //比较 s1 的子串(1,2)和"Hello”
n = s1.compare(1, 2, "Hello", 1, 2);  //比较 s1 的子串(1,2)和 "Hello" 的子串(1,2)
```

6. 求 string 对象的子串

substr 成员函数可以用于求子串 (n, m)，原型如下：

string substr(int n = 0, int m = string::npos) const;

调用时，如果省略 m 或 m 超过了字符串的长度，则求出来的子串就是从下标 n 开始一直到字符串结束的部分。例如：

```
string s1 = "this is ok";
string s2 = s1.substr(2, 4);  // s2 = "is i"
s2 = s1.substr(2);  // s2 = "is is ok"
```

7. 交换两个string对象的内容

swap 成员函数可以交换两个 string 对象的内容。例如：

```
string s1("West"), s2("East");
s1.swap(s2);  // s1 = "East"，s2 = "West"
```

8. 查找子串和字符

string 类有一些查找子串和字符的成员函数，它们的返回值都是子串或字符在 string 对象字符串中的位置（即下标）。如果查不到，则返回 string::npos。string: :npos 是在 string 类中定义的一个静态常量。这些函数如下：

- find：从前往后查找子串或字符出现的位置。
- rfind：从后往前查找子串或字符出现的位置。
- find_first_of：从前往后查找何处出现另一个字符串中包含的字符。例如：
- s1.find_first_of("abc"); //查找s1中第一次出现"abc"中任一字符的位置
- find_last_of：从后往前查找何处出现另一个字符串中包含的字符。
- find_first_not_of：从前往后查找何处出现另一个字符串中没有包含的字符。
- find_last_not_of：从后往前查找何处出现另一个字符串中没有包含的字符。

下面是 string 类的查找成员函数的示例程序。

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1("Source Code");
    int n;
    if ((n = s1.find('u')) != string::npos) //查找 u 出现的位置
        cout << "1) " << n << "," << s1.substr(n) << endl;
    //输出 l)2,urce Code
    if ((n = s1.find("Source", 3)) == string::npos)
        //从下标3开始查找"Source"，找不到
        cout << "2) " << "Not Found" << endl;  //输出 2) Not Found
    if ((n = s1.find("Co")) != string::npos)
        //查找子串"Co"。能找到，返回"Co"的位置
        cout << "3) " << n << ", " << s1.substr(n) << endl;
    //输出 3) 7, Code
    if ((n = s1.find_first_of("ceo")) != string::npos)
        //查找第一次出现或 'c'、'e'或'o'的位置
        cout << "4) " << n << ", " << s1.substr(n) << endl;
```

```
    //输出 4) l, ource Code
    if ((n = s1.find_last_of('e')) != string::npos)
        //查找最后一个 'e' 的位置
        cout << "5) " << n << ", " << s1.substr(n) << endl;  //输出 5) 10, e
    if ((n = s1.find_first_not_of("eou", 1)) != string::npos)
        //从下标1开始查找第一次出现非 'e'、'o' 或 'u' 字符的位置
        cout << "6) " << n << ", " << s1.substr(n) << endl;
    //输出 6) 3, rce Code
    return 0;
}
```

9. 替换子串

replace 成员函数可以对 string 对象中的子串进行替换，返回值为对象自身的引用。例如：

```
string s1("Real Steel");
s1.replace(1, 3, "123456", 2, 4);  //用 "123456" 的子串(2,4) 替换 s1 的子串(1,3)
cout << s1 << endl;  //输出 R3456 Steel
string s2("Harry Potter");
s2.replace(2, 3, 5, 'O');  //用 5 个 'O' 替换子串(2,3)
cout << s2 << endl;  //输出 HaOOOOO Potter
int n = s2.find("OOOOO");  //查找子串 "OOOOO" 的位置，n=2
s2.replace(n, 5, "XXX");  //将子串(n,5)替换为"XXX"
cout << s2 < < endl;  //输出 HaXXX Potter
```

10. 删除子串

erase 成员函数可以删除 string 对象中的子串，返回值为对象自身的引用。例如：

```
string s1("Real Steel");
s1.erase(1, 3);  //删除子串(1, 3)，此后 s1 = "R Steel"
s1.erase(5);  //删除下标5及其后面的所有字符，此后 s1 = "R Ste"
```

11. 插入字符串

insert 成员函数可以在 string 对象中插入另一个字符串，返回值为对象自身的引用。例如：

```
string s1("Limitless"), s2("OO");
s1.insert(2, "123");  //在下标 2 处插入字符串"123"，s1 = "Li123mitless"
s1.insert(3, s2);  //在下标 2 处插入 s2 , s1 = "Li1OO23mitless"
s1.insert(3, 5, 'X');  //在下标 3 处插入 5 个 'X'，s1 = "Li1XXXXXOO23mitless"
```

12. 将 string 对象作为流处理

使用流对象 istringstream 和 ostringstream，可以将 string 对象当作一个流进行输入输出。使用这两个类需要包含头文件 sstream。

示例程序如下：

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
```

```
    string src("Avatar 123 5.2 Titanic K");
    istringstream istrStream(src); //建立src到istrStream的联系
    string s1, s2;
    int n;  double d;  char c;
    istrStream >> s1 >> n >> d >> s2 >> c; //把src的内容当做输入流进行读取
    ostringstream ostrStream;
    ostrStream << s1 << endl << s2 << endl << n << endl << d << endl << c
<<endl;
    cout << ostrStream.str();
    return 0;
}
```

程序的输出结果是

> Avatar
> Titanic
> 123
> 5.2
> K

第 11 行，从输入流 istrStream 进行读取，过程和从 cin 读取一样，只不过输入的来源由键盘变成了 string 对象 src。因此，"Avatar" 被读取到 s1，123 被读取到 n，5.2 被读取到 d，"Titanic" 被读取到 s2，'K' 被读取到 c。

第 12 行，将变量的值输出到流 ostrStream。输出结果不会出现在屏幕上，而是被保存在 ostrStream 对象管理的某处。用 ostringstream 类的 str 成员函数能将输出到 ostringstream 对象中的内容提取出来。

13. 用 STL 算法操作 string 对象

string 对象也可以看作一个顺序容器，它支持随机访问迭代器，也有 begin 和 end 等成员函数。STL 中的许多算法也适用于 string 对象。下面是用 STL 算法操作 string 对象的程序示例。

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main()
{
    string s("afgcbed");
    string::iterator p = find(s.begin(), s.end(), 'c');
    if (p!= s.end())
        cout << p - s.begin() << endl;  //输出 3
    sort(s.begin(), s.end());
    cout << s << endl;  //输出 abcdefg
    next_permutation(s.begin(), s.end());
    cout << s << endl; //输出 abcdegf
    return 0;
}
```

14 c_str函数使用

c_str是string类中的一个函数，

它返回当前字符串的首字符地址。c_str函数 返回**const char*类型**(可读不可改)的指向字符数组的指针。

函数声明：

```
const char* c_str();
```

c_str()函数返回一个指向正规C字符串的指针, 内容与本string串相同.

这是为了与c语言兼容，在c语言中没有string类型，故必须通过string类对象的成员函数c_str()把string 对象转换成c中的字符串样式。

> 注意：一定要使用strcpy()函数 等来操作方法c_str()返回的指针

比如：最好不要这样:

```
char *c;
string s=“1234”;
c = s.c_str();
```

> c最后指向的内容是垃圾，因为s对象被析构，其内容被处理（纠正：s对象的析构是在s的生命周期结束时，例如函数的返回）

应该这样用：

```
char c[20];
string s=“1234”;
strcpy(c,s.c_str());
```

> //这样才不会出错，c_str()返回的是一个临时指针，不能对其进行操作

c_str在打开文件时的用处：

当需要打开一个由用户自己输入文件名的文件时，可以这样写：

```
ifstream in(st.c_str());
```

其中st是string类型，存放的即为用户输入的文件名。