程序设计与算法（三）

C++面向对象程序设计

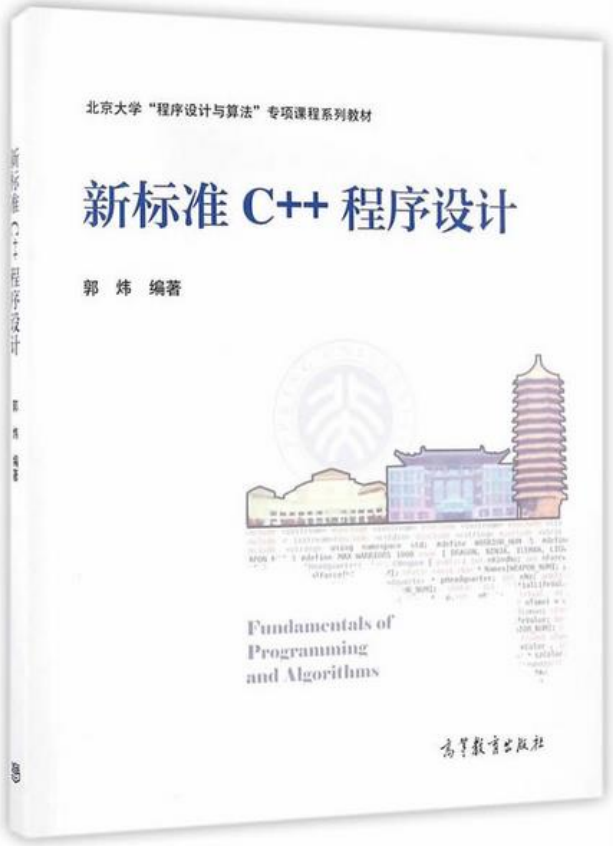
郭炜 **微博** http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著



函数模板和类模板

函数模板

九寨沟长海

函数模板

交换两个整型变量的值的Swap函数：

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

交换两个double型变量的值的Swap函数:

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

函数模板

交换两个整型变量的值的Swap函数：

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

交换两个double型变量的值的Swap函数:

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

能否只写一个Swap，就能交换各种类型的变量？

函数模板

用函数模板解决：

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

};

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

函数模板

```
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;
}
```

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

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

函数模板

函数模板中可以有不止一个类型参数。

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

函数模板

求数组最大元素的MaxElement函数模板

```
template <class T>
T MaxElement(T a[], int size) //size是数组元素个数
{
        T tmpMax = a[0];
        for( int i = 1;i < size;++i)
                if( tmpMax < a[i] )
                        tmpMax = a[i];
        return tmpMax;
}
```

函数模板

不通过参数实例化函数模板

```
#include <iostream>
using namespace std;
template <class T>
T Inc(T n)
{
        return 1 + n;
}
int main()
{
        cout << Inc<double>(4)/2;   //输出 2.5
        return 0;
}
```

函数模板的重载

函数模板可以重载，只要它们的形参表或类型参数表不同即可。

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

函数模板和函数的次序

在有多个函数和函数模板名字相同的情况下，编译器如下处理一条函数调用语句

1) 先找参数完全匹配的普通函数(非由模板实例化而得的函数)。
2) 再找参数完全匹配的模板函数。
3) 再找实参数经过自动类型转换后能够匹配的普通函数。
4) 上面的都找不到，则报错。

```
template <class T>
T Max( T a, T b)  {
        cout << "TemplateMax" <<endl;      return 0;
}
template <class T,class T2>
T Max( T a, T2 b) {
        cout << "TemplateMax2" <<endl;     return 0;
}
double Max(double a, double b){
        cout << "MyMax" << endl;
        return 0;
}
int main() {
    int i=4, j=5;
    Max( 1.2,3.4);  // 输出MyMax
    Max(i, j);  //输出TemplateMax
    Max( 1.2, 3); //输出TemplateMax2
    return 0;
}
```

函数模板

匹配模板函数时，不进行类型自动转换

```
template<class T>
T myFunction( T arg1, T arg2)
{ cout<<arg1<<"   "<<arg2<<"\n"; return arg1;}
……
myFunction( 5, 7); //ok：replace T with int
myFunction( 5.8, 8.4); //ok： replace T with double
myFunction( 5, 8.4); //error，no matching function for call
  to 'myFunction(int, double)'
```

函数模板示例：Map

```
#include <iostream>
using namespace std;
template<class T,class Pred>
void Map(T s, T e, T x, Pred op)
{
   for(; s != e; ++s,++x) {
        *x = op(*s);
   }
}

int Cube(int x) {    return x * x * x; }
double Square(double x) {  return x * x; }
```

```
int a[5] = {1,2,3,4,5}, b[5];
double d[5] = { 1.1,2.1,3.1,4.1,5.1} , c[5];
int main() {
   Map(a,a+5,b,Square);
   for(int i = 0;i < 5; ++i) cout << b[i] << ",";
   cout << endl;

   Map(a,a+5,b,Cube);
   for(int i = 0;i < 5; ++i) cout << b[i] << ",";
   cout << endl;

   Map(d,d+5,c,Square);
   for(int i = 0;i < 5; ++i) cout << c[i] << ",";
   cout << endl;
   return 0;    }
```

输出：
1,4,9,16,25,
1,8,27,64,125,
1.21,4.41,9.61,16.81,26.01,

函数模板示例：Map

```
template<class T,class Pred>
void Map(T s, T e, T x, Pred op) {
   for(; s != e; ++s,++x) {
        *x = op(*s);
   }
}
int a[5] = {1,2,3,4,5}, b[5];
Map(a,a+5,b,Square);   //实例化出以下函数:
void Map(int * s, int * e, int * x, double ( *op)(double)) {
   for(; s != e; ++s,++x) {
        *x = op(*s);
   }
}
```

类模板

新加坡金沙酒店

类模板 – 问题的提出

- 为了多快好省地定义出一批相似的类,可以定义类模板,然后由类模板生成不同的类
- 数组是一种常见的数据类型，元素可以是：
  - 整数
  - 学生
  - 字符串
  - ……
- 考虑一个可变长数组类，需要提供的基本操作
  - len()：查看数组的长度
  - getElement(int index)：获取其中的一个元素
  - setElement(int index)：对其中的一个元素进行赋值
  - ……

类模板 – 问题的提出

- 这些数组类，除了元素的类型不同之外，其他的完全相同

- 类模板：在定义类的时候，加上一个/多个类型参数。在使用类模板时，指定类型参数应该如何替换成具体类型，编译器据此生成相应的模板类。

类模板的定义

```
template <class 类型参数1，class 类型参数2，……> //类型参数表
class 类模板名
{
        成员函数和成员变量
};
```

类模板的定义

```
template <typename 类型参数1，typename 类型参数2，……>
 //类型参数表
class 类模板名
{
        成员函数和成员变量
};
```

类模板的定义

类模板里成员函数的写法：

template <class 类型参数1，class 类型参数2，……> //类型参数表

返回值类型 类模板名<类型参数名列表>::成员函数名（参数表）

{

……

}

用类模板定义对象的写法：

类模板名 <真实类型参数表> 对象名(构造函数实参表);

类模板示例： Pair类模板

```
template <class T1,class T2>
class Pair
{
public:
        T1 key;         //关键字
        T2 value;       //值
        Pair(T1 k,T2 v):key(k),value(v) { };
        bool operator < ( const Pair<T1,T2> & p) const;
};
template<class T1,class T2>
bool Pair<T1,T2>::operator < ( const Pair<T1,T2> & p) const
//Pair的成员函数 operator <
{
        return key < p.key;
}
```

类模板示例：Pair类模板

```
int main()
{
        Pair<string,int> student("Tom",19);
        //实例化出一个类 Pair<string,int>
        cout << student.key << " " << student.value;
        return 0;
}
```

输出：

Tom 19

用类模板定义对象

编译器由类模板生成类的过程叫类模板的实例化。由类模板实例化得到的类，叫模板类。

- 同一个类模板的两个模板类是不兼容的

```
Pair<string,int> * p;
Pair<string,double> a;
p = & a; //wrong
```

函数模版作为类模板成员

```
#include <iostream>
using namespace std;
template <class T>
class A
{
public:
        template<class T2>
        void Func( T2 t) { cout << t; } //成员函数模板
};
int main()
{
        A<int> a;
        a.Func('K'); //成员函数模板 Func被实例化
        a.Func("hello"); //成员函数模板 Func再次被实例化
        return 0;
}
```

输出：*KHello*

类模板与非类型参数

类模板的“<类型参数表>”中可以出现非类型参数：

```
template <class T, int size>
class CArray{
        T  array[size];
    public:
        void Print( )
        {
                for( int i = 0;i < size; ++i)
                        cout << array[i] << endl;
        }
};
CArray<double,40> a2;
CArray<int,50> a3;        //a2和a3属于不同的类
```

类模板与派生

挪威峡湾

类模板与派生

- 类模板从类模板派生
- 类模板从模板类派生
- 类模板从普通类派生
- 普通类从模板类派生

类模板从类模板派生

```
template <class T1,class T2>
class A  {
   T1 v1; T2 v2;
};

template <class T1,class T2>
class B:public A<T2,T1>  {
   T1 v3; T2 v4;
};

template <class T>
class C:public B<T,T>  {
   T v5;
};
```

```
int main()    {
        B<int,double> obj1;
        C<int> obj2;
        return 0;
}
```

类模板从类模板派生

```
template <class T1,class T2>
class A  {
   T1 v1; T2 v2;
};

template <class T1,class T2>
class B:public A<T2,T1>  {
   T1 v3; T2 v4;
};

template <class T>
class C:public B<T,T>  {
   T v5;
};
```

```
int main()    {
        B<int,double> obj1;
        C<int> obj2;
        return 0;
}
```

```
class B<int,double>:
        public A<double,int>
{
        int v3; double v4;
};
```

```
class A<double, int>
{
        double v1; int v2;
};
```

类模板从模板类派生

```
template <class T1,class T2>
class A {
   T1 v1; T2 v2;
};

template <class T>
class B:public A<int,double> {
   T v;
};
int main()  {
   B<char> obj1;  //自动生成两个模板类：A<int,double> 和 B<char>
   return 0;
}
```

类模板从普通类派生

```
class A {
   int v1;
};
template <class T>
class B:public A {     //所有从B实例化得到的类，都以A为基类
   T v;
};
int main() {
   B<char> obj1;
   return 0;
}
```

普通类从模板类派生

```
template <class T>
class A {
   T v1;
   int n;
};

class B:public A<int> {
   double v;
};
int main()  {
   B obj1;
   return 0;
}
```

类模板与友元

挪威Nigardsbreen 冰川

类模板与友员函数

- 函数、类、类的成员函数作为类模板的友元
- 函数模板作为类模板的友元
- 函数模板作为类的友元
- 类模板作为类模板的友元

函数、类、类的成员函数作为类模板的友元

```
void Func1() {  }
class A {  };
class B
{
public:
   void Func() { }
};
template <class T>
class Tmpl
{
   friend void Func1();
   friend class A;
   friend void B::Func();
}; //任何从Tmp1实例化来的类，都有以上三个友元
```

函数模板作为类模板的友元

```
#include <iostream>
#include <string>
using namespace std;
template <class T1,class T2>
class Pair
{
private:
        T1 key;     //关键字
        T2 value;   //值
public:
        Pair(T1 k,T2 v):key(k),value(v) { };
        bool operator < ( const Pair<T1,T2> & p) const;
        template <class T3,class T4>
        friend ostream & operator<< ( ostream & o,
                                      const Pair<T3,T4> & p);
};
```

函数模板作为类模板的友元

```
template<class T1,class T2>
bool Pair<T1,T2>::operator < ( const Pair<T1,T2> & p) const
{ //"小"的意思就是关键字小
        return key < p.key;
}
template <class T1,class T2>
ostream & operator<< (ostream & o,const Pair<T1,T2> & p)
{
        o << "(" << p.key << "," << p.value << ")" ;
        return o;
}
```

函数模板作为类模板的友元

```
int main()
{
        Pair<string,int> student("Tom",29);
        Pair<int,double> obj(12,3.14);
        cout << student << " " << obj;
        return 0;
}
```

输出：

(Tom,29) (12,3.14)

任意从 `template <class T1,class T2>`

`ostream & operator<< (ostream & o,const Pair<T1,T2> & p)`

生成的函数，都是任意Pair摸板类的友元

函数模板作为类的友元

```
#include <iostream>
using namespace std;
class A
{
    int v;
public:
    A(int n):v(n) { }
    template <class T>
    friend void Print(const T & p);
};
template <class T>
void Print(const T & p)
{
        cout << p.v;
}
```

```
int main()  {
    A a(4);
    Print(a);
    return 0;
}
```

输出：

4

所有从 `template <class T> void Print(const T & p)` **生成的函数，都成为** `A` **的友元**

但是自己写的函数

`void Print(int a) { }`

不会成为A的友元

类模板作为类模板的友元

```
#include <iostream>
using namespace std;
template <class T>
class B  {
    T v;
public:
    B(T n):v(n) { }
    template <class T2>
    friend class A;
};
template <class T>
class A  {
  public:
    void Func( )   {
       B<int> o(10);
       cout << o.v << endl;
    }
};
```

```
int main()
{
  A< double > a;
  a.Func ();
  return 0;
}
```

输出：
10

A< double>类，成了B<int>类的友元。任何从A模版实例化出来的类，都是任何B实例化出来的类的友元

类模板与静态成员变量

冰岛杰古沙龙冰河湖

类模板与static成员

- **类模板中可以定义静态成员，那么从该类模板实例化得到的所有类，都包含同样的静态成员。**

```
#include <iostream>
using namespace std;
template <class T>
class A
{
 private:
   static int count;
public:
   A() { count ++;  }
   ~A() { count -- ; };
   A( A & ) { count ++ ; }
   static void PrintCount() { cout << count << endl; }
};
```

类模板与static成员

```
template<> int A<int>::count = 0;
template<> int A<double>::count = 0;
int main()
{
   A<int> ia;
   A<double> da;
   ia.PrintCount();
   da.PrintCount();
   return 0;
}
```

输出：

```
1
1
```