

程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2024 年 3 月 27 日

大纲

- 1 函数模板
- 2 类模板
- 3 类模板与派生
- 4 类模板与友元
- 5 类模板与静态成员变量

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;  
}
```

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：

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  
};
```

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

```
1  int main() {  
2      int n = 1, m = 2;  
3      Swap(n, m); //编译器自动生成 void Swap(int &,int &) 函数  
4      double f = 1.2, g = 2.3;  
5      Swap(f, g); //编译器自动生成 void Swap(double &,double &) 函数  
6      return 0;  
7  }
```

函数模板

```
1 int main() {  
2     int n = 1, m = 2;  
3     Swap(n, m); //编译器自动生成 void Swap(int &,int &) 函数  
4     double f = 1.2, g = 2.3;  
5     Swap(f, g); //编译器自动生成 void Swap(double &,double &) 函数  
6     return 0;  
7 }
```

```
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。

```
1  template <class T1, class T2>
2  T2 print(T1 arg1, T2 arg2) {
3      cout<< arg1 << " " << arg2<<endl;
4      return arg2;
5  }
6
```

函数模板

函数模板中可以有不止一个类型参数。

```
1  template <class T1, class T2>
2  T2 print(T1 arg1, T2 arg2) {
3      cout<< arg1 << " "<< arg2<<endl;
4      return arg2;
5  }
6
```

求数组最大元素的 MaxElement 函数模板

```
1  template <class T>
2  T MaxElement(T a[], int size) { //size 是数组元素个数
3      T tmpMax = a[0];
4      for (int i = 1; i < size; ++i) {
5          if(tmpMax < a[i]) tmpMax = a[i];
6      }
7      return tmpMax;
8  }
9
```

不通过参数实例化函数模板

```
1  #include <iostream>
2  using namespace std;
3  template <class T>
4  T Inc(T n) {
5      return 1 + n;
6  }
7  int main() {
8      cout << Inc<double>(4)/2;  //输出 2.5
9      return 0;
10 }
11
```

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。

```
1  template<class T1, class T2>
2  void print(T1 arg1, T2 arg2) {
3      cout << arg1 << " " << arg2 << endl;
4  }
5  template<class T>
6  void print(T arg1, T arg2) {
7      cout << arg1 << " " << arg2 << endl;
8  }
9  template<class T, class T2>
10 void print(T arg1, T arg2) {
11     cout << arg1 << " " << arg2 << endl;
12 }
```

函数模板和函数的次序

在有多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。

函数模板的重载

```
1  template <class T>
2  T Max(T a, T b) {
3      cout << "TemplateMax" << endl;    return 0;
4  }
5  template <class T, class T2>
6  T Max(T a, T2 b) {
7      cout << "TemplateMax2" << endl;    return 0;
8  }
9  double Max(double a, double b){
10     cout << "MyMax" << endl;
11     return 0;
12 }
13 int main() {
14     int i = 4, j = 5;
15     Max(1.2, 3.4); // 输出 MyMax
16     Max(i, j);    // 输出 TemplateMax
17     Max(1.2, 3);  // 输出 TemplateMax2
18     return 0;
19 }
20
```


函数模板

匹配模板函数时，不进行类型自动转换

```
1  #include <iostream>
2  using namespace std;
3
4  template<class T>
5  T myFunction(T arg1, T arg2) {
6      cout << arg1 << "    " << arg2 << "\n";
7      return arg1;
8  }
9
10 int main() {
11     myFunction(5, 7); //ok: replace T with int
12     myFunction(5.8, 8.4); //ok: replace T with double
13     myFunction(5, 8.4); //error, no matching function for
14                          //call to 'myFunction(int, double)'
15     return 0;
16 }
17
```

函数模板

模板实参推导 zh.cppreference.com/w/cpp/language/template_argument_deduction

重载决议 zh.cppreference.com/w/cpp/language/overload_resolution

以下说法哪个不正确

- Ⓐ 函数模板中可以有不止一个类型参数
- Ⓑ 函数模板可以重载
- Ⓒ 函数模板中的类型参数也可以用来表示函数模板的返回值类型
- Ⓓ 函数模板中的类型参数不能用于定义局部变量

以下说法哪个不正确

- ☐ A 函数模板中可以有不止一个类型参数
- ☐ B 函数模板可以重载
- ☐ C 函数模板中的类型参数也可以用来表示函数模板的返回值类型
- ☐ D 函数模板中的类型参数不能用于定义局部变量

答案：D

函数模板示例：Map

```
1  #include <iostream>
2  using namespace std;
3  template<class T, class Pred>
4  void Map(T s, T e, T x, Pred op) {
5      for(; s != e; ++s, ++x) {
6          *x = op(*s);
7      }
8  }
9  int Cube(int x) {
10     return x * x * x;
11 }
12 double Square(double x) {
13     return x * x;
14 }
15
```

函数模板示例：Map

```
15  int a[5] = {1, 2, 3, 4, 5}, b[5];
16  double d[5] = {1.1, 2.1, 3.1, 4.1, 5.1} , c[5];
17  int main() {
18      Map(a, a+5, b, Square);
19      for (int i = 0; i < 5; ++i) cout << b[i] << ", ";
20      cout << endl;
21
22      Map(a, a+5, b, Cube);
23      for (int i = 0; i < 5; ++i) cout << b[i] << ", ";
24      cout << endl;
25
26      Map(d, d+5, c, Square);
27      for (int i = 0; i < 5; ++i) cout << c[i] << ", ";
28      cout << endl;
29      return 0;
30  }
```

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

```
1  template<class T,class Pred>
2  void Map(T s, T e, T x, Pred op) {
3      for(; s != e; ++s,++x) {
4          *x = op(*s);
5      }
6  }
7  int a[5] = {1,2,3,4,5}, b[5];
8  Map(a, a+5, b, Square); //实例化出以下函数:
9  void Map(int * s, int * e, int * x, double (*op)(double)) {
10     for(; s != e; ++s,++x) {
11         *x = op(*s);
12     }
13 }
```

```
1  template <class T1, class T2, class T3>
2  T1 func(T1 * a, T2 b, T3 c) {    }
3  int a[10], b[10];
4  void f(int n) {    }
```

func(a, b, f); 将模板类型参数实例化的结果是：

- ☒ A T1: int, T2: int *, T3: void (*) (int)
- ☐ B T1: int *, T2: int *, T3: void (*) (int)
- ☐ C T1: int, T2: int, T3: void (int)
- ☐ D T1: int *, T2 int, T3: void (int)


```
1  template <class T1, class T2, class T3>
2  T1 func(T1 * a, T2 b, T3 c) {   }
3  int a[10], b[10];
4  void f(int n) {   }
```

func(a, b, f); 将模板类型参数实例化的结果是：

- ☒ A T1: int, T2: int *, T3: void (*) (int)
- ☐ B T1: int *, T2: int *, T3: void (*) (int)
- ☐ C T1: int, T2: int, T3: void (int)
- ☐ D T1: int *, T2 int, T3: void (int)

答案：A

类模板—问题的提出

为了多快好省地定义出一批相似的类型, 可以定义类模板, 然后由类模板生成不同的类

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
- 浮点数
- 字符串
- 其他自定义类

类模板一问题的提出

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
- 浮点数
- 字符串
- 其他自定义类

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
-

类模板一问题的提出

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
- 浮点数
- 字符串
- 其他自定义类

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
-

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

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

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

```
1  template <class 类型参数 1, class 类型参数 2, .....> //类型参数表
2  class 类模板名 {
3      成员函数和成员变量
4  };
```

类模板

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

```
1  template <class 类型参数 1, class 类型参数 2, .....> //类型参数表
2  class 类模板名 {
3      成员函数和成员变量
4  };
```

```
1  template <typename 类型参数 1, typename 类型参数 2, .....> //类型参数表
2  class 类模板名 {
3      成员函数和成员变量
4  };
5
```


类模板

类模板里成员函数的写法：

```
1  template <class 类型参数 1, class 类型参数 2, .....> //类型参数表  
2  返回值类型 类模板名 < 类型参数名列表 >:: 成员函数名 (参数表) {  
3      .....  
4  }  
5
```

类模板

类模板里成员函数的写法：

```
1  template <class 类型参数 1, class 类型参数 2, .....> //类型参数表  
2  返回值类型 类模板名 < 类型参数名列表 >:: 成员函数名 (参数表) {  
3      .....  
4  }  
5
```

用类模板定义对象的写法：

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

```
1  template <class T1,class T2>
2  class Pair {
3  public:
4      T1 key; //关键字
5      T2 value; //值
6      Pair(T1 k,T2 v):key(k),value(v) { };
7      bool operator < (const Pair<T1,T2> & p) const;
8  };
9  template<class T1,class T2>
10 bool Pair<T1,T2>::operator < ( const Pair<T1,T2> & p) const {
11 //Pair 的成员函数 operator <
12     return key < p.key;
13 }
14 int main() {
15     Pair<string,int> student("Tom", 19);
16     //实例化出一个类 Pair<string,int>
17     cout << student.key << " " << student.value;
18     return 0;
19 }
20
```

函数模版作为类模板成员

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) { //成员函数模板
        cout << t;
    }
};
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;
}
```

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：

```
1  template <class T, int size>
2  class CArray{
3      T array[size];
4  public:
5      void Print() {
6          for(int i = 0; i < size; ++i)
7              cout << array[i] << endl;
8      }
9  };
10 CArray<double,40> a2;
11 CArray<int,50> a3;
```

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：

```
1  template <class T, int size>
2  class CArray{
3      T array[size];
4  public:
5      void Print() {
6          for(int i = 0; i < size; ++i)
7              cout << array[i] << endl;
8      }
9  };
10 CArray<double,40> a2;
11 CArray<int,50> a3;
```

a2 和 a3 属于不同的类

以下说法不正确的是：

- Ⓐ 类模板可以用函数模板作为其成员
- Ⓑ 类模板的类型参数可以用来定义成员变量
- Ⓒ 不可以有多个类模板名字相同
- Ⓓ 同一类模板实例化出来的类，是互相类型兼容的

以下说法不正确的是：

- Ⓐ 类模板可以用函数模板作为其成员
- Ⓑ 类模板的类型参数可以用来定义成员变量
- Ⓒ 不可以有多个类模板名字相同
- Ⓓ 同一类模板实例化出来的类，是互相类型兼容的

答案：D

- 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

```
1  template <class T1, class T2>
2  class A {
3      T1 v1;
4      T2 v2;
5  };
6
7  template <class T1, class T2>
8  class B:public A<T2, T1> {
9      T1 v3;
10     T2 v4;
11 };
12
13 template <class T>
14 class C:public B<T, T> {
15     T v5;
16 };
17 int main() {
18     B<int, double> obj1;
19     C<int> obj2;
20     return 0;
21 }
```

类模板从模板类派生

```
1  template <class T1, class T2>
2  class A {
3      T1 v1;
4      T2 v2;
5  };
6
7  template <class T>
8  class B:public A<int, double> {
9      T v;
10 };
11 int main() {
12     B<char> obj1; //自动生成两个模板类: A<int,double> 和 B<char>
13     return 0;
14 }
15
```

类模板从普通类派生

```
1  class A {  
2      int v1;  
3  };  
4  template <class T>  
5  class B:public A {    //所有从 B 实例化得到的类，都以 A 为基类  
6      T v;  
7  };  
8  int main() {  
9      B<char> obj1;  
10     return 0;  
11 }
```

普通类从模板类派生

```
1  template <class T>
2  class A {
3      T v1;
4      int n;
5  };
6
7  class B:public A<int> {
8      double v;
9  };
10 int main() {
11     B obj1;
12     return 0;
13 }
14
```

类模板与友元函数

- 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

```
1  void Func1() {  
2  }  
3  class A {  
4  };  
5  class B{  
6  public:  
7      void Func() {  
8      }  
9  };  
10 template <class T>  
11 class Tmpl {  
12     friend void Func1();  
13     friend class A;  
14     friend void B::Func();  
15 }; //任何从 Tmpl 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

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  template <class T1, class T2>
5  class Pair {
6  private:
7      T1 key;    //关键字
8      T2 value;  //值
9  public:
10     Pair(T1 k,T2 v):key(k),value(v) { };
11     bool operator < (const Pair<T1, T2> & p) const;
12     template <class T3, class T4>
13     friend ostream & operator<< (ostream & o, const Pair<T3, T4> & p);
14 };
15 template<class T1, class T2>
16 bool Pair<T1,T2>::operator < (const Pair<T1, T2> & p) const {
17     return key < p.key; // "小" 的意思就是关键字小
18 }
19 template <class T1,class T2>
20 ostream & operator<< (ostream & o, const Pair<T1,T2> & p) {
21     o << "(" << p.key << "," << p.value << ")" ;
22     return o;
```

函数模板作为类模板的友元

```
25  int main() {  
26      Pair<string,int> student("Tom",29);  
27      Pair<int,double> obj(12,3.14);  
28      cout << student << " " << obj;  
29      return 0;  
30  }
```

函数模板作为类模板的友元

```
25  int main() {  
26      Pair<string,int> student("Tom",29);  
27      Pair<int,double> obj(12,3.14);  
28      cout << student << " " << obj;  
29      return 0;  
30  }
```

输出:
(Tom,29) (12,3.14)

函数模板作为类模板的友元

```
25 int main() {  
26     Pair<string,int> student("Tom",29);  
27     Pair<int,double> obj(12,3.14);  
28     cout << student << " " << obj;  
29     return 0;  
30 }
```

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 模板类的友元

函数模板作为类的友元

```
1  class A {  
2      int v;  
3  public:  
4      A(int n):v(n) { }  
5      template <class T>  
6      friend void Print(const T & p);  
7  };  
8  template <class T>  
9  void Print(const T & p) {  
10     cout << p.v;  
11 }  
12  
13 int main() {  
14     A a(4);  
15     Print(a);  
16     return 0;  
17 }  
18
```

函数模板作为类的友元

```
1  class A {  
2      int v;  
3  public:  
4      A(int n):v(n) { }  
5      template <class T>  
6      friend void Print(const T & p);  
7  };  
8  template <class T>  
9  void Print(const T & p) {  
10     cout << p.v;  
11 }  
12  
13 int main() {  
14     A a(4);  
15     Print(a);  
16     return 0;  
17 }  
18
```

输出：4

函数模板作为类的友元

```
1  class A {  
2      int v;  
3  public:  
4      A(int n):v(n) { }  
5      template <class T>  
6      friend void Print(const T & p);  
7  };  
8  template <class T>  
9  void Print(const T & p) {  
10     cout << p.v;  
11 }  
12  
13 int main() {  
14     A a(4);  
15     Print(a);  
16     return 0;  
17 }  
18
```

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。

函数模板作为类的友元

```
1  class A {  
2      int v;  
3  public:  
4      A(int n):v(n) { }  
5      template <class T>  
6      friend void Print(const T & p);  
7  };  
8  template <class T>  
9  void Print(const T & p) {  
10     cout << p.v;  
11 }  
12  
13 int main() {  
14     A a(4);  
15     Print(a);  
16     return 0;  
17 }  
18
```

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

```
1  template <class T>
2  class B {
3      T v;
4  public:
5      B(T n):v(n) { }
6      template <class T2>
7      friend class A;
8  };
9  template <class T>
10 class A {
11 public:
12     void Func() {
13         B<int> o(10);
14         cout << o.v << endl;
15     }
16 };
17 int main() {
18     A<double> a;
19     a.Func();
20     return 0;
21 }
```

类模板作为类模板的友元

```
1  template <class T>
2  class B {
3      T v;
4  public:
5      B(T n):v(n) { }
6      template <class T2>
7      friend class A;
8  };
9  template <class T>
10 class A {
11 public:
12     void Func() {
13         B<int> o(10);
14         cout << o.v << endl;
15     }
16 };
17 int main() {
18     A<double> a;
19     a.Func();
20     return 0;
21 }
```

输出: 10

类模板作为类模板的友元

```
1  template <class T>
2  class B {
3      T v;
4  public:
5      B(T n):v(n) { }
6      template <class T2>
7      friend class A;
8  };
9  template <class T>
10 class A {
11 public:
12     void Func() {
13         B<int> o(10);
14         cout << o.v << endl;
15     }
16 };
17 int main() {
18     A<double> a;
19     a.Func();
20     return 0;
21 }
```

输出: 10

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 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每个模板类，都有自己的类模板静态数据成员，该模板类的所有对象，共享一个静态数据成员

```
1  template <class T>
2  class A {
3  private:
4      static int count;
5  public:
6      A() { count ++; }
7      ~A() { count -- ; };
8      A(A & ) { count ++ ; }
9      static void PrintCount() { cout << count << endl; }
10 };
11 template<> int A<int>::count = 0;
12 template<> int A<double>::count = 0;
13 int main() {
14     A<int> ia;
15     A<double> da;
16     ia.PrintCount();
17     da.PrintCount();
18     return 0;
19 }
```

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每个模板类，都有自己的类模板静态数据成员，该模板类的所有对象，共享一个静态数据成员

```
1  template <class T>
2  class A {
3  private:
4      static int count;
5  public:
6      A() { count ++; }
7      ~A() { count -- ; };
8      A(A & ) { count ++ ; }
9      static void PrintCount() { cout << count << endl; }
10 };
11 template<> int A<int>::count = 0;
12 template<> int A<double>::count = 0;
13 int main() {
14     A<int> ia;
15     A<double> da;
16     ia.PrintCount();
17     da.PrintCount();
18     return 0;
19 }
```

输出：

1

1