

郭 婕 刻家臻

类模板



问题的提出

定义一批
相似的类

定义类
模板

生成不同
的类

数组

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
-

问题的提出



- 对于这些数组类
 - 除了元素的类型不同之外，其他的完全相同
- 类模板
 - 在定义类的时候给它一个/多个参数
 - 这个/些参数表示不同的数据类型
- 在调用类模板时，指定参数，由编译系统根据参数提供对应的数据类型自动产生相应的**模板类**

类模板的定义

- ▶ C++的类模板的写法如下：

```
template <类型参数表>
class 类模板名
{
    成员函数和成员变量
};
```
- ▶ 类型参数表的写法就是：

```
class 类型参数1, class 类型参数2, ...
```

类模板的定义

类模板里的成员函数，如在类模板外面定义时，

`template <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)`

{

...

}



类模板的定义

- 用类模板定义对象的写法如下：
类模板名 <真实类型参数表> 对象名(构造函数实际参数表);
- 如果类模板有无参构造函数，那么也可以只写：
类模板名 <真实类型参数表> 对象名；





类模板的定义

■ 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; }

类模板的定义

- ▶ Pair类模板的使用：

```
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;
}
```

输出结果：

Tom 19

使用类模板声明对象

- 编译器由类模板生成类的过程叫**类模板的实例化**
 - 编译器自动用**具体的数据类型**替换类模板中的**类型参数**, 生成模板类的代码
- 由类模板实例化得到的**类模板类**
 - 为类型参数指定的数据类型不同, 得到的模板类不同

使用类模板声明对象

- 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
```





函数模板反作用为类模板成员

```
#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
    return 0;
}
```

若函数模板改为

```
template <class T>
void Func(T t){cout<<t}
将报错“declaration of ‘class T’
shadows template parm ‘class T’”
```

程序输出：

K

类模板与非类型参数

类模板的参数声明中可以包括非类型参数

```
template <class T, int elementsNumber>
```

- 非类型参数：用来说明类模板中的属性
- 类型参数：用来说明类模板中的属性类型，成员操作的参数类型和返回值类型

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;
    }
};
```



类模板与非类型参数

```
CArray<double, 40> a2;  
CArray<int, 50> a3;
```

▶ 注意：

CArray<int,40>和CArray<int,50>完全是两个类
这两个类的对象之间不能互相赋值

类模板与继承



- 类模板派生出类模板
- 模板类(即类模板中类型/非类型参数实例化后的类)

模板类派生出类模板

普通类派生出类模板

模板类派生出普通类

类模板

普通类

模板类

派生

派生

派生

派生

类模板

类模板

类模板

普通类

普通类

普通类

(1) 类模板从人类模板派生

```
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;
};

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

class C<int> obj2;
B<int, double> obj1;
T v5;
};
int main(){
    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;
}
```

(2) 类模板从模板派生

```
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; return 0; }

    ▶ 自动生成两个模板类 : A<int, double>和B<char>
```

(3) 类模板从普通类派生

```
class A { int v1; };
```

```
template <class T>
class B:public A { T v; };
```

```
int main() {
    B<char> obj1;
    return 0;
}
```



(4) 普通类从模板类派生

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

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

