第九章 泛型编程

在C++里,**不考虑具体数据类型**的编程模式叫做泛型编程，**泛型也是一种数据类型，只不过它是一种用来代替所有类型的“通用类型”。**

泛型编程通过函数模板和类模板来实现泛型编程。

9.1 函数模板

9.1.1 函数模板的基本使用

当我们想写个Swap()交换函数时,通常这样写:

```
void Swap(int& a, int& b)
{
	int c = a;
	a = b;
	b = c;
}
```

但是这个函数仅仅只能支持int类型,如果我们想实现交换double,float,string等等时,就还需要从新去构造Swap()重载函数，这样不但重复劳动，容易出错，而且还带来很大的维护和调试工作量。更糟的是，还会增加可执行文件的大小。

我们可以通过函数模板解决这个问题：

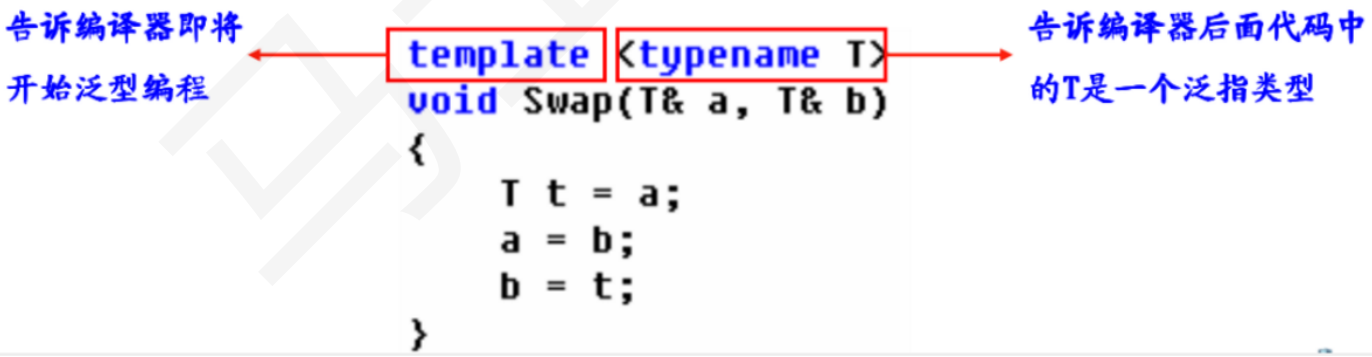
```
void Swap(T& a, T& b)
{
    T t = a;
    a = b;
    b = t;
}
```

Swap 泛型写法中的 T 不是一个具体的数据类型,而是泛指任意的数据类型。

函数模板其实是一个具有相同行为的函数家族

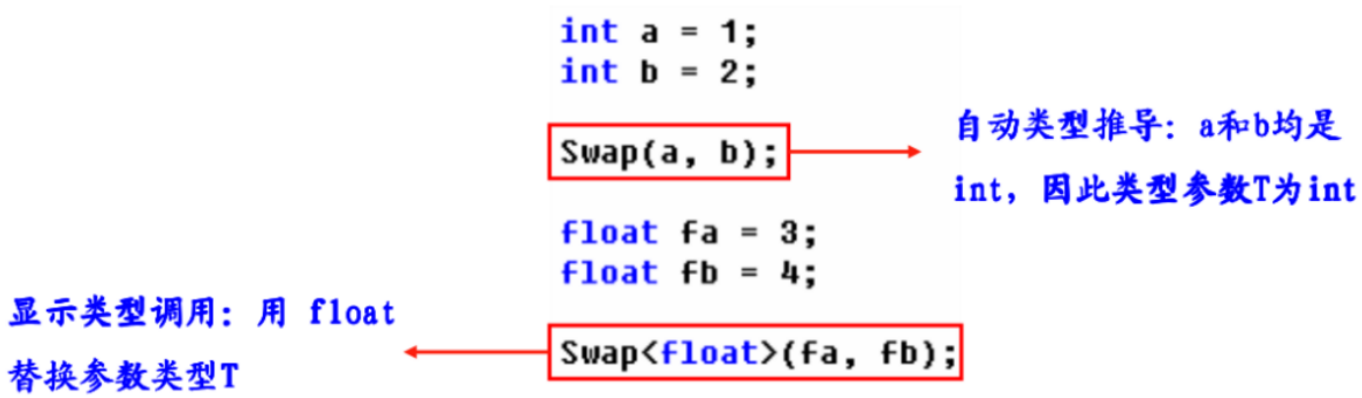
函数模板的语法规则如下

- template　　关键字用于声明开始进行泛型编程
- typename　　关键字用于声明泛指类型



函数模板的应用

- 自动类型推导调用
- 具体类型显示调用



9.1.2 深入理解函数模板

1、为什么函数模板能够执行不同的类型参数?

> 其实编译器对函数模板进行了两次编译
>
> 第一次编译时,首先去检查函数模板本身有没有语法错误
>
> 第二次编译时,会去找调用函数模板的代码,然后通过代码的真正参数,来生成真正的函数。
>
> 所以函数模板,其实只是一个模具,当我们调用它时,编译器就会给我们生成真正的函数.

2、如何验证呢？

编译程序生成.o文件，然后通过nm命令查看符号表

> g++ -c 函数模板.cpp
> nm 函数模板.o

```
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
000000000000017b t _GLOBAL__sub_I__Z5swap1RiS_
0000000000000066 T main
                 U __stack_chk_fail
0000000000000132 t _Z41__static_initialization_and_destruction_
0000000000000000 W _Z4SwapIfEvRT_S1_
0000000000000000 W _Z4SwapIiEvRT_S1_
000000000000002d T _Z5swap1RfS_
0000000000000000 T _Z5swap1RiS_
                 U _ZNSolsEi
                 U _ZNSolsEPFRSoS_E
                 U _ZNSt8ios_base4InitC1Ev
```

从符号表中可以非常直观的看到生成了两个不同的符号。

> **需要注意的是：函数模板是不允许隐式类型转换的，调用时类型必须严格匹配**

也就说如下代码是非法的：

```
int a;
float b;
Swap(a, b);
```

3、函数模板的特点

数模板还可以定义任意多个不同的类型参数，但是对于多参数函数模板：

- 编译器是无法自动推导返回值类型的
- 可以从左向右部分指定类型参数

```
#include <iostream>

using namespace std;

template <typename T1, typename T2, typename T3>
T1 add(T2 a, T3 b)
{
    T1 ret;
    ret = static_cast<T1>(a + b);
    return ret;
}
void main()
{
    int c = 12;
    float d = 23.4;
    //cout << add(c, d) << endl;        //error，无法自动推导函数返回值
    cout << add<float>(c, d) << endl;        //返回值在第一个类型参数中指定
    cout << add<int, int, float>(c, d) << endl;
}
```

4、函数模板的重载

函数模板跟普通函数一样，也可以被重载

- C++编译器优先考虑普通函数
- 如果函数模板可以产生一个更好的匹配，那么就选择函数模板

- 也可以通过空模板实参列表<>限定编译器只匹配函数模板

```
#include <iostream>

using namespace std;

template <typename T>`
void fun(T a)
{
    cout << "void fun(T1 a)" << endl;
}

template <typename T1, typename T2>
void fun(T1 a, T2 b)
{
    cout << "void fun(T1 a, T2 b)" << endl;
}

 //函数模板的重载
void fun(int a, float b)
{
    cout << "void fun(int a, float b)" << endl;
}

void main()
{
    int a = 0;
    float b = 0.0;
    fun(a);
    fun(a, b);        //普通函数void fun(int a, float b)已经能完美匹配，于是调用普通函数
    fun(b, a);        //这个调用，函数模板有更好的匹配，于是调用函数模板
    fun<>(a, b);        //限定只使用函数模板
}
```

5、总结

- 函数模板是泛型编程在C++中的应用方式之一
- 函数模板能够根据实参对参数类型进行推导
- 函数模板支持显示的指定参数类型
- 函数模板是C++中重要的代码复用方式

- 函数模板通过具体类型产生不同的函数
- 函数模板可以定义任意多个不同的类型参数
- 函数模板中的返回值类型必须显示指定
- 函数模板可以像普通函数一样重载
- 在任何一个函数模板前都必须 使用template <typename 泛型,...>声明开始泛型编程

9.2 类模板

9.2.1 类模板的定义

还记得我们上次实现的Array类吗？在Array类中我们只能操作int类型的数据，如果需要操作char，float类型的数据我们该如何处理呢？难道我们再重新实现一个类吗？当然不用，我们可以使用类模板来实现。

在实际工作中，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，我们可以使用类模板来实现。

```
template<class 模板参数表>
class 类名 {
// 类定义．．．．．．
} ;
```

— 提供一种特殊的类以相同的行为处理不同的类型

— 在类声明前使用template进行标识

— <typename T>用于说明类中使用的泛指类型 T

```
template<typename T>
class Operator
{
public:
    T add(T a, T b)
    {
        return a + b;
    }

    T minus(T a, T b)
    {
        return a - b;
    }
};
```

对象op1对象op1用于处理int类型的加减法

```
Operator<int> op1;
Operator<double> op2;
cout<<op1.add(5, 4)<<endl;
cout<<op2.minus(1.5, 0.01)<<endl;
```

对象op2对象op2用于处理double类型的加减法

这样我们就定义了一个简单的类模板，其中的T代表任意的类型，可以出现在类模板中的任意地方，与函数模板不同的是，使用类模板构造对象时必须显示的指定数据类型，编译器无法自动推导，例如test<int>t；需要显示的指定数据类型。

编译器对类模板的处理方式和函数模板相同

- **编译器从类模板通过具体类型产生不同的类**
- **编译器在声明的地方对类模板代码本身进行编译**
- **编译器在使用的地方对参数替换后的代码进行编译**

由于类模板的编译机制不同，所以不能像普通类一样分开实现后在使用时只包含头文件，在工程实践上，一般会把类模板的定义直接放到头文件中!!

只有被调用的类模板成员函数才会被编译器生成可执行代码!!!

一在模板类外部定义成员函数的实现时，需要加上template<typename T>的声明

```
template<typename T>
class Operator
{
public:
    T add(T a, T b);
    T minus(T a, T b);
};

template<typename T>
T Operator<T>::add(T a, T b)
{
    return a + b;
}

template<typename T>
T Operator<T>::minus(T a, T b)
{
    return a - b;
}
```

9.2.2 类模板的特化

上面的类模板好像已经实现了add加法运算.但是却不能支持指针类型。其实,类模板也可以像函数重载一样, 类模板通过特化的方式可以实现特殊情况.

类模板特化：

- 表示可以存在多个相同的类名,但是模板类型都不一致(和函数重载的参数类似)
- 特化类型有**完全特化**和**部分特化**两种类型
- 完全特化表示显示指定类型参数，模板声明只需写成template<>,并在类名右侧指定参数,比如：

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

template < typename T1,typename T2 >  //声明的模板参数个数为2个
class Operator                        //正常的类模板
{
```

```
        public:
                Operator()
                {
                        cout << "Operator" << endl;
                }
          void add(T1 a, T2 b)
          {
                        cout<<a+b<<endl;
          }
};

template <>                 //不需要指定模板类型,因为是完全特化的类模板
class Operator<int, int>     //指定类型参数,必须为2个参数,和正常类模板参数个数一致
{
        public:
                Operator()
                {
                        cout << "Operator<int , int>" << endl;
                }
                void add(int a, int b)
                {
                        cout<<a+b<<endl;
                }
};

int main()
{
        //匹配完全特化类模板:class Operator< int,int>
   Operator<int,int> Op1;

 //匹配正常的类模板
   Operator<int,float> Op2;

    return 0;
}
```

- 部分特化表示通过特定规则约束类型参数,模板声明和类相似,并在类名右侧指定参数,比如:

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

template < typename T1,typename T2 >          //声明的模板参数个数为2个
class Operator                               //正常的类模板
{
public:
       void add(T1 a, T2 b)
      {
```

```
        cout<<a+b<<endl;
    }
};

template < typename T >          //有指定模板类型以及指定参数,所以是部分特化的类模板
class Operator< T* ,T*>          //指定类型参数,必须为2个参数,和正常类模板参数个数一致
{
public:
    void add(T* a, T* b)
    {
    cout<<*a+*b<<endl;
    }
};

int main()
{
    Operator<int*,int*> Op1;          //匹配部分特化: class Operator< T* ,T*>
    Operator<int,float> Op2;          //匹配正常的类模板: class Operator
return 0;
}
```

- 编译时,会根据对象定义的类模板类型,首先去匹配完全特化,再来匹配部分特化,最后匹配正常的类模板

```
#include <iostream>

using namespace std;

template < typename T1,typename T2 >
class Operator              //正常的类模板
{
public:
    void add(T1 a, T2 b)
    {
        cout<<"add(T1 a, T2 b)"<<endl;
        cout<<a+b<<endl;
    }
};

template < typename T >
class Operator<T,T>       //特化的类模板,当两个参数都一样,调用这个
{
public:
    void add(T a, T b)
    {
        cout<<"add(T a, T b)"<<endl;
```

```
        cout<<a+b<<endl;
    }
};

template < typename T1,typename T2 >
class Operator<T1*,T2*>                    //部分特化的类模板,当两个参数都是指针,调用这个
{
public:
    void add(T1* a, T2* b)
    {
        cout<<"add(T1* a, T2* b)"<<endl;
        cout<<*a+*b<<endl;
    }
};

template < >
class Operator<void*,void*>        //完全特化的类模板,当两个参数都是void*,调用这个
{
public:
    void add(void* a, void* b)
    {
        cout<<"add(void* a, void* b)"<<endl;
        cout<<"add void* Error"<<endl;    //void*无法进行加法
    }
};

int main()
{
    int *p1 = new int(1);
    float *p2 = new float(1.25);

    Operator<int,float>  Op1;        //匹配正常的类模板:class Operator
    Op1.add(1,1.5);

    Operator<int,int>  Op2;          //匹配部分特化的类模板:class Operator<T,T>
    Op2.add(1,4);

    Operator<int*,float*>  Op3;      //匹配部分特化的类模板:class Operator<T1*,T2*>
    Op3.add(p1,p2);

    Operator<void*,void*>  Op4;      //匹配完全特化的类模板:class Operator<void*,void*>
    Op4.add(NULL,NULL);

    delete p1;
    delete p2;
```

```
    return 0;
}
```

9.2.3 继承中类模板的使用

9.2.3.1 父类是一般类，子类是模板类

```
class A {
public:
    A(int temp = 0) {
        this->temp = temp;
    }
    ~A(){}
private:
    int temp;
};

template <typename T>
class B :public A{
public:
    B(T t = 0) :A(666) {
        this->t = t;
    }
    ~B(){}
private:
    T t;
} ;
```

9.2.3.2 子类是一般类，父类是模板类

```
template <typename T>
class A {
public:
    A(T t = 0) {
        this->t = t;
    }
    ~A(){}
private:
    T t;
};


class B:public A<int> {
```

```
public:
   //也可以不显示指定，直接A(666)
      B(int temp = 0):A<int>(666) {
            this->temp = temp;
      }
      ~B() {}
private:
      int temp;
};
```

9.2.3.3 子类和父类都是模板类

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

//1、类模板继承类模板
template <typename T1, typename T2>
class A
{
   T1 x;
   T2 y;
};

template <typename T1, typename T2>
class B : public A<T2, T1>
{
   T1 x1;
   T2 y2;
};

template <typename T>
class C : public B<T, T>
{
   T x3;
};

//2、类模板继承模板类
template <typename T>
class D : public A<int, double> //具体化的模板类
{
   T x4;
};
//3、类模板继承普通类
class E
{
```

```
    int x4;
};
template <typename T>
class F : public E
{
    T X5;
};
```