模板

引入

在之前的C++程序设计课程中我们已经学过了函数重载方法，可以实现代码复用。但存在着两种问题

- 重载的函数仅仅只是类型不同，代码的复用率比较低，只要有新类型出现时，就要增加相应的函数。
- 代码的可维护性比较低，一个出错可能所以的重载均会出错。

所以我们引入了模板方法来实现用一个通用的模具来实现不同类型的代码。

函数模板

概念：

函数模板代表了一个函数家族，该函数模板与类型无关，在使用时被参数化，根据实参类型产生函数的特定类型版本。

格式：

```
template<typename T1,typename T2,.....,typename Tn> //可以有许多不同类型的不同变量
//下为样例：
template<typename T>
void swap(T &a,T &b){
    T tmp = a;
    a = b;
    b = tmp;
}
```

原理：

函数模板本身并不是函数，是编译器用使用方式产生特定具体类型函数的模具，所以其实模板就是将本来我们该做的重复的事情交给了编译器。

在编译阶段，编译器会根据传入的实参类型来推演生成对应类型的函数以供调用。当用int类型使用模板函数时，编译器通过对实参类型的推演，将T确定为int类型，然后生成一份专门处理int类型的代码，对double类型、字符类型也是如此。

实例化：

用不同类型的参数使用函数模板时，称为函数模板的实例化。

模板参数的实例化分为：**隐式实例化**和**显式实例化**。

1. 隐式实例化：

```
template<class T>
T add(const T& left,const T& right){
    return left + right;
}
int main(){
    int a1=10;
    double b1=10.0;
    //add(a1,b1); 不可以通过编译
    add(a1,(int)b1);  //我们可以通过强制类型转换来解决这个问题，或者通过下面要讲到的显式实例化。
    return 0;
}
```

2. 显式实例化：

```
int main(){
    int a=10;
    double b=10.0;
    add<int>(a,b);
    return 0;
}
```

注：类型若不匹配，编译器会尝试进行隐式类型转换，如果无法转换成功编译器将会报错。

匹配原则：

1. 一个非模板函数可以和一个同名的函数模板同时存在，而且该函数模板还可以被实例化为这个非模板函数。

```
int add(int a, int b){  //专门处理int的加法函数
    return a + b;
}
template<class T>
T add(T a, T b){  //通用加法函数
    return a + b;
}
void main(){
    add(1, 2);  //与非模板函数匹配，编译器不需要特化
    add<int>(1, 2);  //调用编译器特化的add版本
}
```

2. 对于非模板函数和同名函数模板，如果其他条件都相同，在调用时会优先调用非模板函数而不会从该模板产生出一个实例。如果模板可以产生一个具有更好匹配的函数，那么将选择模板。

```
int add(int a, int b){  //专门处理int的加法函数
    return a + b;
}
template<class T1, class T2>
T1 add(T1 a, T2 b){  //通用加法函数
    return a + b;
}
void main(){
    add(1, 2);  //与非模板函数匹配，不需要函数模板实例化
    add(1, 2.0);  //模板函数可以产生更加匹配的版本，编译器根据实参生成更加匹配的add函数
}
```

3. 模板函数不允许自动类型转换，但普通函数可以进行自动类型转换。

类模板

格式：

类声明

注意此处 class 和 typename 是完全等价的。

```
template<class T1,class T2,...,class Tn> //类型参数表
class 类模板名{
    ... //类内成员定义
};
```

类成员函数

```
template <class T1,class T2,...,class Tn> //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)
{
    ... //类内成员定义
}
```

例子：

```
template<class T>
class Vector{
public:
    Vector(size_t capacity=10):_pData(new
T[capacity]),_size(0),_capacity(capacity){}
    //使用析构函数演示：在类中声明，在类外定义。
    ~Vector();
    size_t Size(){
        return _size;
    }
    T& operator[](size_t pos){
        assert(pos<_size);
        return _pData[pos];
```

```
    }
private:
    T* _pData;
    size_t _size;
    size_t _capacity;
};
//注意：类模板中函数放在类外进行定义时，需要加模板参数列表
template<class T>
Vector<T>::~Vector(){
    if(_pData) delete[] _pData;
    _size = _capacity = 0;
}
```

传递参数：

在使用模板的时候我们可以看到，`template <typename T>`类似一个函数签名，而`T`就是这个函数签名里面的形参，`typename`就是参数类型。`typename`表示该位置接收一个类型名作为参数。

除了`typename`接收类型名作为参数之外，模板还可以接收数据作为参数，例如`int`类型的数据。**例子：**

```
template <typename T, int n>     //接收一个整型数据作为模板的参数
class Stack{
    private:
        static int num = 0;
        int cap;
        T st[n];                 //用接收到的数据初始化数组长度
};
```

注意模板参数只能接收`整型`，`枚举`，`引用`和`指针`类型的数据，并且在使用时传入数据的时候，只能使用常量表达式传入数据。

```
//正确
Stack<int, 4> s;

//错误，传入数据的时候只能使用常量表达式
int x = 4;
Stack<double, x> s;
```

实例化与具体化：

跟函数模板一样，类模板的具体化也有不同的方式。

隐式实例化

这是最普通的具体化，不需要用户自己指出，编译器会在使用用户用模板声明一个类的时候自动生成类。例如:

```
Stack<int, 4> s;
```

显式实例化

使用一个语句来向编译器发出指令，让其立即生成一个类，即使现在不需要这个类。例如：

```
template class Stack<int, 5>;
```

与编译器编译模式有关，详情可以访问模板显式实例化用途详解。(https://blog.csdn.net/Jxianxu/article/details/124359007)

显式具体化

显式实例化跟显式具体化的区别是，一个不需要修改模板，另一个需要修改模板。显式具体化的作用是，为特定的模板参数实现对应的类。例如：

```
template <>
class stack<double , 5>{
    private:
        static int num;
        int cap = 5;
        double * st;
};
```

上面代码的含义是当模板参数为`<double, 5>`时，使用另外给出的类实现方式，而放弃原来的方式。

部分具体化

也可以在显式具体化时，之规定部分参数的值，例如：

```
template <int n>
class stack<double, n>{
    private:
        static int num;
        int cap = 5;
        double * st;
};
```

上面代码表示的是，如果模板参数的第一个参数为`double`，则使用另外给定的类型。这里只规定了第一个参数的值，而没有规定第二个参数，所以是部分显式具体化。

指针部分具体化

如果传入的参数是一个指针，则使用给定的实现：

```
template <typename T>
class stack{
    private:
        static int num;
        int cap;
        T st[4];
    public:
        stack(){
            cout << "normal class" << endl;
        }
};

template <typename T>
class stack<T*>{        //表明参数为指针时，使用该实现
    private:
        static int num;
```

```
        int cap = 5;
        T* st;
    public:
        stack(){
            cout << "pointer class" << endl;
        }
};

stack<int> s;   //使用第一个实现，普通实现
stack<int*> s1; //使用第二个实现，指针专用实现
```

内部模板

定义一个模板的时候也可以在一个类模板的内部定义另一个类模板。例如：

```
template <typename T>
class Outer{
    private:
        template<typename U>
        class Inner{
            private:
                U u;
            public:
                U getU();
        };
        Inner<int> intInner;
        Inner<T> argInner;
    public:
        T getU(){
            return argInner.getU();
        }
};

//如果要在类外定义函数，使用这种嵌套模板的格式
template <typename T>
template <typename U>
U Outer<T>::Inner<U>::getU(){
    return u;
}
```

使用模板作为参数

模板也可以接收另外一个模板作为参数，例如下面实现的`Store`类，接收模板作为参数，以使得在使用的时候可以改变数据的存取方式：

```
template <typename T> class Stack{};
template <typename T> class Queue{};

template <template<typename T> class Ag, typename u>
class Store{
    private:
        Ag<u> st;
};

Store<Stack, int> s1;
Store<Queue, double> s2;
```

这样就能在运行的时候决定`Store`类运用哪种数据结构组织数据。

友元相关

模板参数表中的类型参数同样可以声明为该模板类的友元类

例子

```
template <typename T>
class DemoClass{
    friend T; //将参数类型T声明为DemoClass的友元类
    ...
}
```

设定别名

我们也可以通过typedef和using来为类型设定别。

例子

```
typedef DemoClass<int> intDemoClass;
using intDemoClass = DemoClass<int>;
```