C++模板

函数模板

模板是C++支持**参数化**多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

1.模板是一种对**类型**进行**参数化**的工具；

2.通常有两种形式：**函数模板**和**类模板**；

3.函数模板针对仅**参数类型**不同的**函数**；

4.类模板针对仅**数据成员**和**成员函数类型**不同的类。

使用模板的目的就是能够让程序员编写与类型无关的代码(代码重用)

01 变量交换函数模板

假设我们设计一个交换两个整型变量的值的函数，代码如下：

```
// 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;
}
```

果是浮点类型的变量的值交换，则替换 int 类型为 double 即可，代码如下：

```
// 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;
}
```

那如果是其他变量类型的值交换，那不是每次都要重新写一次 Swap函数？是不是很繁琐？且代码后面会越来越冗余。

能否只写一个 Swap函数，就能交换各种类型的变量？

答案是肯定有的，就是用「函数模板」来解决，「函数模板」的形式：

```
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
};
```

具体 Swap「函数模板」代码如下：

template就是模板定义的关键词，T代表的是任意变量的类型。

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

那么定义好「函数模板」后，在编译的时候，编译器会根据传入 Swap函数的参数变量类型，自动生成对应参数变量类型的 Swap函数：

```
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;
}
```

上面的实例化函数模板的例子，是让编译器自己来判断传入的变量类型，那么我们也可以自己指定函数模板的变量类型，具体代码如下：

```
int main(){
    int n = 1,m = 2;
    Swap<int>(n,m);      // 指定模板函数的变量类型为int
    double f = 1.2,g = 2.3;
    Swap<double>(f,g); // 指定模板函数的变量类型为double
    return 0;
}
```

例子：

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

void swap1(int &x, int &y){
    int nTemp = x;
    x = y;
    y = nTemp;
}

void swap1(float &x, float &y){
    float nTemp = x;
    x = y;
    y = nTemp;
}

// template在这里表示是一个模板,class T在这里是一个模板形参,这个模板形参要紧接着函数名来写
```

```
// typename来替换class,class在这里不是类
// template<typename T> void swap1(T &x, T &y){
template<class T> void swap1(T &x, T &y){
    T nTemp = x;
    x = y;
    y = nTemp;
}

int main(){
    int a = 10;
    int b = 20;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap1(a, b);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    float c = 20.4;
    float d = 30.5;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    swap1(c, d);
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
    return 0;
}
```

那么定义好「函数模板」后，在编译的时候，编译器会根据传入 Swap函数的参数变量类型，自动生成对应参数变量类型的 Swap函数：

```
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;
}
```

上面的实例化函数模板的例子，是让编译器自己来判断传入的变量类型，那么我们也可以自己指定函数模板的变量类型，具体代码如下：

```
int main(){
    int n = 1,m = 2;
    Swap<int>(n, m);      // 指定模板函数的变量类型为int
    double f = 1.2,g = 2.3;
    Swap<double>(f, g);  // 指定模板函数的变量类型为double
    return 0;
}
```

02 查询数组最大值函数模板

再举一个例子，下面的 MaxElement函数定义成了函数模板，这样不管是 int、double、char 等类型的数组，都可以使用该函数来查数组最大的值，代码如下：

```
template<class T> T maxElement(T a[], int size){
    T tmpMax = a[0];
    for(int i=1;i<size;++i){
        if(tmpMax < a[i]){
            tmpMax = a[i];
        }
    }
    return tmpMax;
}
```

03 多个类型参数模板函数

函数模板中，可以不止一个类型的参数：

```
// template<typename A, typename B> B MyFun(A arg1, B arg2){}
template<class T1, class T2> T2 MyFun(T1 arg1, T2 arg2){
    cout << arg1 << " " << arg2 << endl;
    return arg2;
}
```

T1 是传入的第一种任意变量类型，T2 是传入的第二种任意变量类型。

04 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> void print(T1 arg1){
    cout << arg1 << endl;
}
// 2
template<class T1> void print(T1 arg1, T1 arg2){
    cout << arg1 << endl;
}
// 3
template<class T1, class T2> void print(T1 arg1, T2 arg2){
    cout << arg1 << endl;
}

// 这个模板函数会和 3 函数冲突,被认为和3函数是一个函数redefinition
// 4
template<class T1, class T3> void print(T1 arg1, T3 arg2){
    cout << arg1 << endl;
}
```

上面都是 print(参数1, 参数2)模板函数的重载，因为「形参表」或「类型参数表」名字不同。

05 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，编译器如下规则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。
5. 匹配模板函数时，当模板函数只有一个参数类型时，传入了不同的参数类型，不进行类型自动转换

代码例子如下：

```
// 模板函数 - 1个参数类型
template<class T> T Max(T a, T b) {
    cout << "TemplateMax" << endl;
    return 0;
}

// 模板函数 - 2个参数类型
template <class T, class T2> T Max(T a, T2 b) {
    cout << "TemplateMax2" << endl;
    return 0;
}

// 普通函数
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 - 匹配普通函数
    Max(i, j);          // 输出TemplateMax - 匹配参数一样的模板函数
    Max(1.2, 3);        // 输出TemplateMax2 - 匹配参数类型不同的模板函数

    return 0;
}
```

注意：匹配模板函数时，当模板函数只有一个参数类型时，传入了不同的参数类型，不进行类型自动转换，具体例子如下：

```
// 模板函数 - 1个参数类型
template<class T> T myFunction(T arg1, T arg2){
    cout << arg1 << " " << arg2 << endl;
    return arg1;
}

// OK : 替换 T 为 int 类型
myFunction(5, 7);
// OK : 替换 T 为 double 类型
myFunction(5.8, 8.4);
// error : 没有匹配到myFunction(int, double)函数
myFunction(5, 8.4);
```

类模板

01 类模板的定义

为了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。

类模板的定义形式如下：

```
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
};
```

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);
```

02 Pair类模板例子

接下来，用 Pair 类用类模板的方式的实现，Pair 是一对的意思，也就是实现一个键值对（key-value）的关系的类。

```
// 类模板
template <class T1, class T2> class Pair{
public:
    Pair(T1 k, T2 v):m_key(k),m_value(v) {};
    bool operator<(const Pair<T1, T2> &p) const;
private:
    T1 m_key;
    T2 m_value;
};

// 类模板里成员函数的写法
template <class T1, class T2> bool Pair<T1,T2>::operator<(const Pair<T1, T2> &p)
const{
```

```
    return m_value < p.m_value;
}

int main(){
    Pair<string, int> Astudent("Jay", 20);
    Pair<string, int> Bstudent("Tom", 21);
    cout << (Astudent < Bstudent) << endl;
    return 0;
}
```

输出结果：

```
1
```

需要注意的是，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; //错误！！
```

----------测试编译器能不能兼容

03 函数模板作为类模板成员

当函数模板作为类模板的成员函数时，是可以单独写成函数模板的形式，成员函数模板在使用的时候，编译器才会把函数模板根据传入的函数参数进行实例化，例子如下：

```
// 类模板
template<class T> class A{
public:
    // 成员函数模板
    template<class T2> void Func(T2 t) { cout << t << endl; }
};

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;
}
```

04 类模板与非类型参数

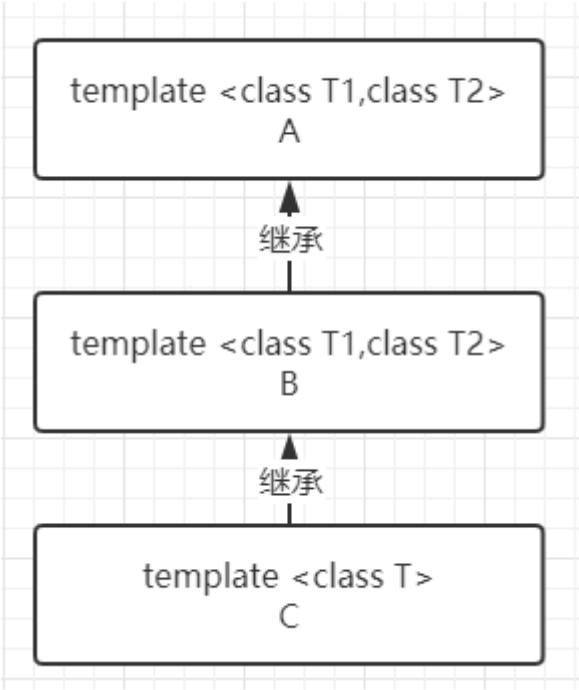
类模板的"<类型参数表>"中可以出现非类型参数：

```
template <class T, int size> class CArray{
public:
    void Print();
private:
    T array[size];
```

```
};

template <class T, int size> void CArray<T, int >::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

01 类模板从类模板派生

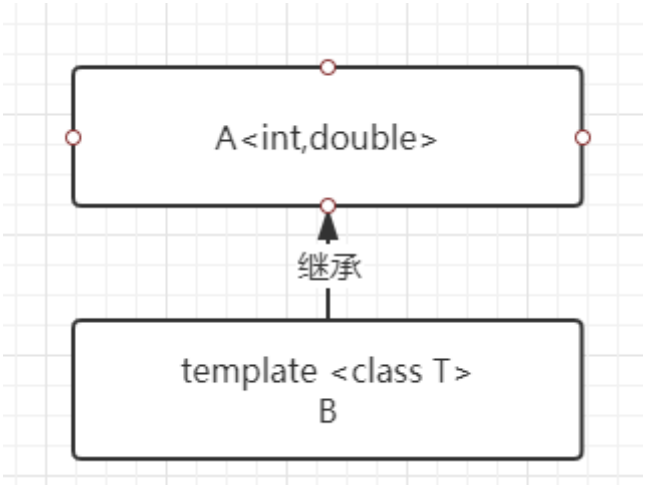


上图的代码例子如下：

```
// 基类 - 类模板
template <class T1,class T2>class A {
private:
    T1 v1;
    T2 v2;
};

// 派生类 - 类模板
template <class T1, class T2>class B : public A<T2, T1> {
    T1 v3;
    T2 v4;
};

// 派生类 - 类模板
template <class T>class C : public B<T1, T2> {
```

```
    T v5;
};

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

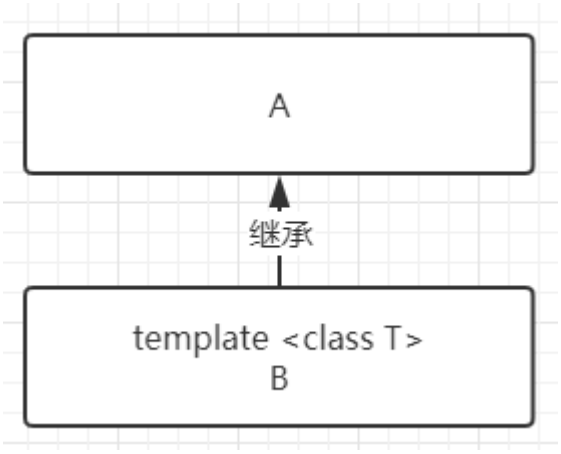
02 类模板从模板类派生



上图的代码例子如下：

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

// A<int,double> 模板类
template <class T>class B : public A<int, double>  {
    T v;
};

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

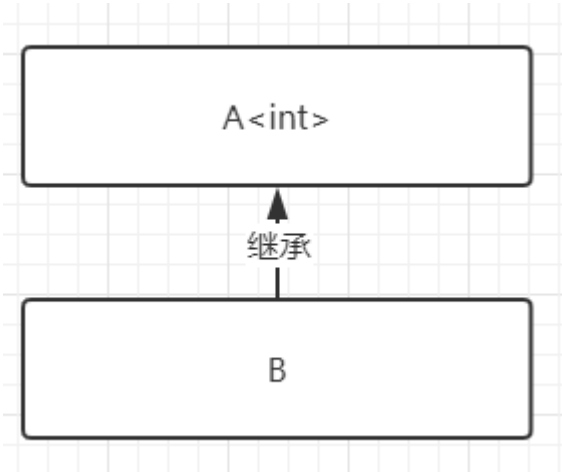
03 类模板从普通类派生



上图的代码例子如下：

```
// 基类 - 普通类
class A {
    int v1;
};

// 派生类 - 类模板
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;
}
```

04 普通类从模板类派生



上图的代码例子如下：

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

class B : public A<int> {
    double v;
};

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

类模板与友元

01 函数、类、类的成员函数作为类模板的友元

代码例子如下：

```
// 普通函数
void Func1(){}

// 普通类
class A{};

// 普通类
class B{
public:
    void Func(){} // 成员函数
};

// 类模板
template <class T> class Tmp{
    friend void Func1();    // 友元函数
    friend class A;         // 友元类
    friend void B::Func();  // 友元类的成员函数
}; // 任何从 Tmp 实例化来的类 ，都有以上三个友元
```

02 函数模板作为类模板的友元

```
// 类模板
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) { };

    // 友元函数模板
    template <class T3, class T4>
    friend ostream & operator<<(ostream & o, const Pair<T3, T4> & p);
```

```
};

// 函数模板
template <class T3,class T4>
ostream & operator<<(ostream & o, const Pair<T3, T4> & p){
    o << "(" << p.key << "---" << p.value << ")" ;
    return o;
}

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

输出结果：

```
(Tom,29) (12,3.14)
```

03 函数模板作为类的友元

```
// 普通类
class A{
private:
    int v;
public:
    explicit A(int n):v(n) { }
    template <class T>friend void Print(const T & p); // 函数模板
};

// 函数模板
template <class T>void Print(const T & p){
    cout << p.v;
}

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

输出结果：

```
4
```

04 类模板作为类模板的友元

```
// 类模板
template <class T>class B {
private:
    T v;
public:
    B(T n):v(n) { }
    template <class T2>friend class A; // 友元类模板
};

// 类模板
template <class T> class A {
public:
    void Func()
    {
        B<int> o(10); // 实例化B模板类
        cout << o.v << endl;
    }
};

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

输出结果：

```
10
```

类模板与静态成员变量

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。

```
template <class T>class A{
private:
    static int count; // 静态成员
public:
    A() { count ++; }
    ~A() { count -- ; };
    A(A &) { count ++ ; }

    static void PrintCount() { cout << count << endl; } // 静态函数
};

template<> int A<int>::count = 0;     // 初始化
template<> int A<double>::count = 0; // 初始化

int main(){
    A<int> ia;
    A<double> da;     // da和ia不是相同模板类
    ia.PrintCount();
```

```
    da.PrintCount();
    return 0;
}
```

输出：

```
1
1
```

上面的代码需要注意的点：

类模板里的静态成员初始化的时候，最前面要加template<>。

- ia 和 da 对象是不同的模板类，因为类型参数是不一致，所以也就是不同的模板类。