Lesson06---模板

【本节目标】

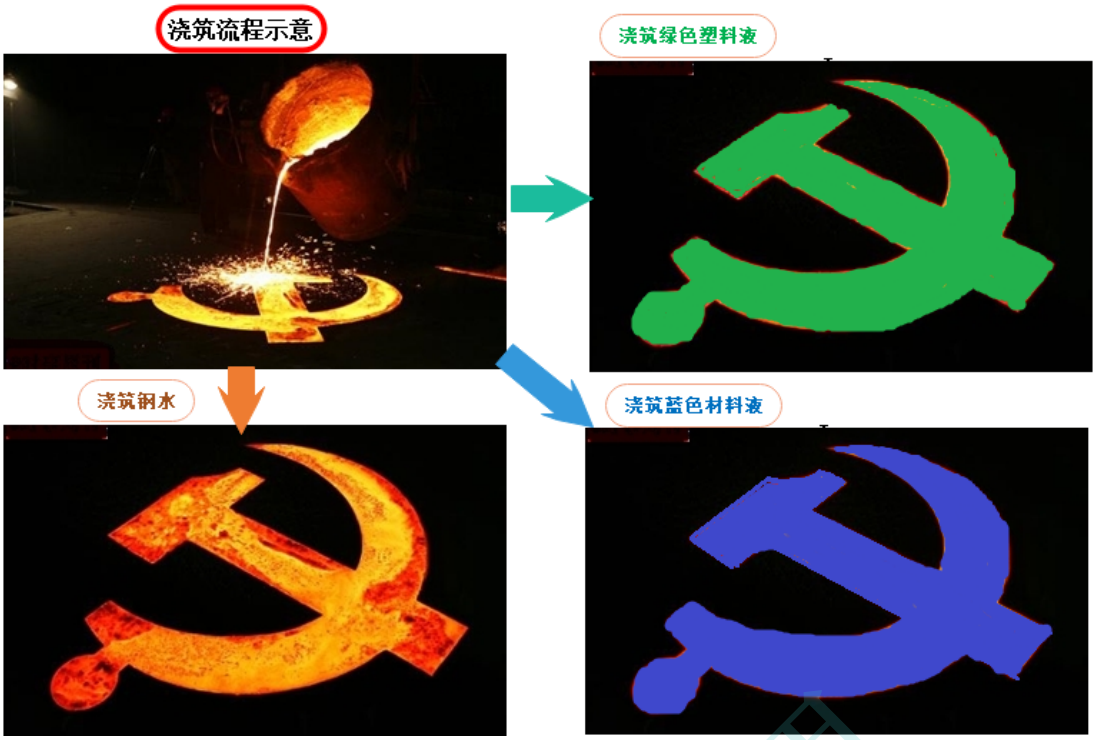
- 1. 泛型编程
- 2. 函数模板
- 3. 类模板
- 4. 非类型模板参数
- 5. 类模板的特化
- 6. 模板的分离编译

1. 泛型编程

> 如何实现一个通用的交换函数呢?

```
void Swap(int& left, int& right)
{
  int temp = left;
  left = right;
  right = temp;
}

void Swap(double& left, double& right)
{
  double temp = left;
  left = right;
  right = temp;
}

void Swap(char& left, char& right)
{
  char temp = left;
  left = right;
  right = temp;
}

......
```

> 使用函数重载虽然可以实现，但是有一下几个不好的地方：
>
> 1. 重载的函数仅仅是类型不同，代码复用率比较低，只要有新类型出现时，就需要用户自己增加对应的函数
> 2. 代码的可维护性比较低，一个出错可能所有的重载均出错
>
> 那能否**告诉编译器一个模子，让编译器根据不同的类型利用该模子来生成代码**呢?



如果在C++中，也能够存在这样一个**模具**，通过给这个模具中**填充不同材料(类型)**，来**获得不同材料的铸件(即生成具体类型的代码)**，那将会节省许多头发。巧的是前人早已将树栽好，我们只需在此乘凉。

泛型编程：编写与类型无关的通用代码，是代码复用的一种手段。模板是泛型编程的基础。

2. 函数模板

2.1 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。

2.1 函数模板格式

template<typename T1, typename T2,……,typename Tn>

返回值类型 函数名(参数列表){}

```
template<typename T>
void Swap( T& left,  T& right)
{
 T temp = left;
 left = right;
 right = temp;
}
```

> **注意：typename是用来定义模板参数关键字，也可以使用class(切记：不能使用struct代替class)**

2.3 函数模板的原理

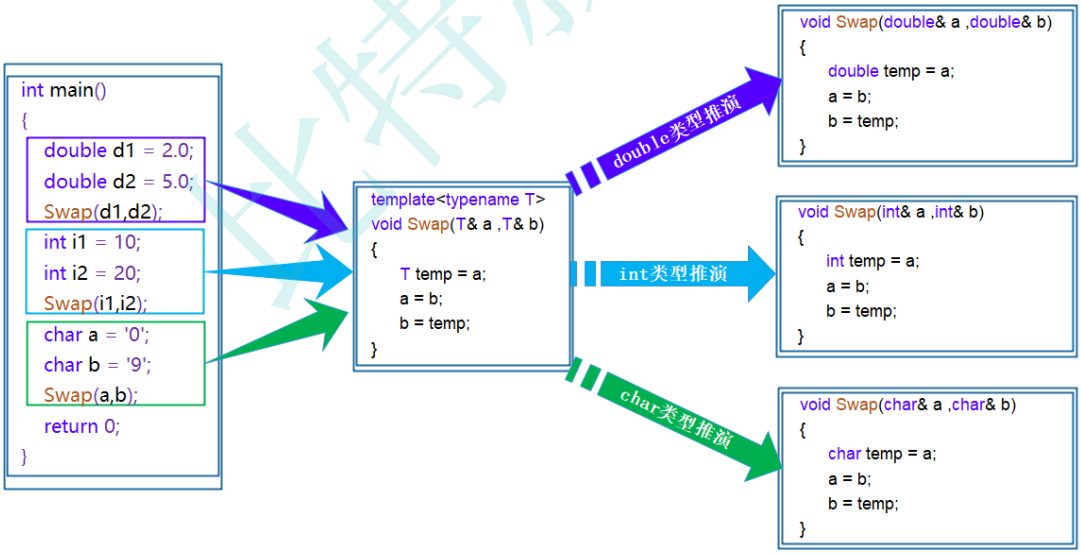
> 那么如何解决上面的问题呢？大家都知道，瓦特改良蒸汽机，人类开始了工业革命，解放了生产力。机器生产淘汰掉了很多手工产品。本质是什么，重复的工作交给了机器去完成。有人给出了论调：懒人创造世界。

马云：世界是懒人创造的

2015-11-04 16:26

懒不是傻懒，如果你想少干，就要想出懒的方法。要懒出风格，懒出境界。

函数模板是一个蓝图，它本身并不是函数，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应该我们做的重复的事情交给了编译器



在编译器编译阶段，对于模板函数的使用，**编译器需要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。比如：**当用double类型使用函数模板时，编译器通过对实参类型的推演，将T确定为double类型，然后产生一份专门处理double类型的代码**，对于字符类型也是如此。

2.4 函数模板的实例化

> **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。**模板参数实例化分为：**隐式实例化和显式实例化。**
>
> 1. **隐式实例化：让编译器根据实参推演模板参数的实际类型**

```
template<class T>
T Add(const T& left, const T& right)
{
    return left + right;
}

int main()
{
    int a1 = 10, a2 = 20;
    double d1 = 10.0, d2 = 20.0;
    Add(a1, a2);
    Add(d1, d2);

    /*
     该语句不能通过编译，因为在编译期间，当编译器看到该实例化时，需要推演其实参类型
     通过实参a1将T推演为int，通过实参d1将T推演为double类型，但模板参数列表中只有一个T，
     编译器无法确定此处到底该将T确定为int 或者 double类型而报错
    注意：在模板中，编译器一般不会进行类型转换操作，因为一旦转化出问题，编译器就需要背黑锅
     Add(a1, d1);
    */

    // 此时有两种处理方式：1. 用户自己来强制转化 2. 使用显式实例化
    Add(a, (int)d);
    return 0;
}
```

2. **显式实例化：在函数名后的<>中指定模板参数的实际类型**

```
int main(void)
{
    int a = 10;
    double b = 20.0;

    // 显式实例化
    Add<int>(a, b);
    return 0;
}
```

如果类型不匹配，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。

2.5 模板参数的匹配原则

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**

```
// 专门处理int的加法函数
int Add(int left, int right)
{
    return left + right;
}

// 通用加法函数
template<class T>
T Add(T left, T right)
{
```

```
    return left + right;
}

void Test()
{
    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，在调动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**

```
// 专门处理int的加法函数
int Add(int left, int right)
{
    return left + right;
}

// 通用加法函数
template<class T1, class T2>
T1 Add(T1 left, T2 right)
{
    return left + right;
}

void Test()
{
    Add(1, 2);      // 与非函数模板类型完全匹配，不需要函数模板实例化
    Add(1, 2.0);    // 模板函数可以生成更加匹配的版本，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**

3. 类模板

3.1 类模板的定义格式

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

```
// 动态顺序表
// 注意：Vector不是具体的类，是编译器根据被实例化的类型生成具体类的模具
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();

 void PushBack(const T& data);
 void PopBack();
 // ...

 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;
}
```

3.2 类模板的实例化

类模板实例化与函数模板实例化不同，**类模板实例化需要在类模板名字后跟<>，然后将实例化的类型放在<>中即可，类模板名字不是真正的类，而实例化的结果才是真正的类。**

```
// Vector类名，Vector<int>才是类型
Vector<int> s1;
Vector<double> s2;
```

4. 非类型模板参数

模板参数分类类型形参与非类型形参。

类型形参即：出现在模板参数列表中，跟在class或者typename之类的参数类型名称。

非类型形参，就是用一个常量作为类(函数)模板的一个参数，在类(函数)模板中可将该参数当成常量来使用。

```
namespace bite
{
 // 定义一个模板类型的静态数组
```

```
    template<class T, size_t N = 10>
    class array
    {
    public:
    T& operator[](size_t index){return _array[index];}
    const T& operator[](size_t index)const{return _array[index];}

    size_t size()const{return _size;}
    bool empty()const{return 0 == _size;}

    private:
    T _array[N];
    size_t _size;
    };
}
```

注意：

1. **浮点数、类对象以及字符串是不允许作为非类型模板参数的。**
2. **非类型的模板参数必须在编译期就能确认结果。**

5. 模板的特化

5.1 概念

通常情况下，**使用模板可以实现一些与类型无关的代码，但对于一些特殊类型的可能会得到一些错误的结果**，需要特殊处理，比如：

```
struct Date
{
    int _year = 1;
    int _month = 1;
    int _day = 1;
};

template<class T>
bool IsEqual(T left, T right)
{
    return left == right;
}

// 函数模板的特化（针对某些类型的特殊化处理）
bool IsEqual(Date*  left, Date* right)
{
    return left->_year == right->_year
        && left->_month == right->_month
        && left->_day == right->_day;
}

int main()
{
    cout << IsEqual(1, 2) << endl;

    Date* p1 = new Date;
    Date* p2 = new Date;
```

```
    cout << IsEqual(p1, p2) << endl;;

    return 0;
}
```

此时，就**需要对模板进行特化。即：在原模板类的基础上，针对特殊类型所进行特殊化的实现方式。**模板特化中分为**函数模板特化**与**类模板特化。**

5.2 函数模板特化

函数模板的特化步骤：

1. 必须要先有一个基础的函数模板
2. 关键字template后面接一对空的尖括号<>
3. 函数名后跟一对尖括号，尖括号中指定需要特化的类型
4. 函数形参表: 必须要和模板函数的基础参数类型完全相同，如果不同编译器可能会报一些奇怪的错误。

```
struct Date
{
    int _year = 1;
    int _month = 1;
    int _day = 1;
};

template<class T>
bool IsEqual(T left, T right)
{
    return left == right;
}

template<>
bool IsEqual<Date*>(Date*  left, Date* right)
{
    return left->_year == right->_year
        && left->_month == right->_month
        && left->_day == right->_day;
}

int main()
{
    cout << IsEqual(1, 2) << endl;

    Date* p1 = new Date;
    Date* p2 = new Date;
    cout << IsEqual(p1, p2) << endl;;

    return 0;
}
```

注意：一般情况下如果函数模板遇到不能处理或者处理有误的类型，为了实现简单通常都是将该函数直接给出。

5.3 类模板特化

5.3.1 全特化

全特化即是将模板参数列表中所有的参数都确定化。

```
template<class T1, class T2>
class Data
{
public:
    Data() {cout<<"Data<T1, T2>" <<endl;}
private:
 T1 _d1;
 T2 _d2;
};

template<>
class Data<int, char>
{
public:
    Data() {cout<<"Data<int, char>" <<endl;}
private:
 int _d1;
 char _d2;
};

void TestVector()
{
 Data<int, int> d1;
 Data<int, char> d2;
}
```

5.3.2 偏特化

偏特化：任何针对模版参数进一步进行条件限制设计的特化版本。比如对于以下模板类：

```
template<class T1, class T2>
class Data
{
public:
    Data() {cout<<"Data<T1, T2>" <<endl;}
private:
 T1 _d1;
 T2 _d2;
};
```

偏特化有以下两种表现方式：

- 部分特化

  将模板参数类表中的一部分参数特化。

```
// 将第二个参数特化为int
template <class T1>
class Data<T1, int>
{
public:
    Data() {cout<<"Data<T1, int>" <<endl;}
private:
    T1 _d1;
    int _d2;
};
```

- 参数更进一步的限制

  偏特化并不仅仅是指特化部分参数，而是针对模板参数更进一步的条件限制所设计出来的一个特化版本。

```
//两个参数偏特化为指针类型
template <typename T1, typename T2>
class Data <T1*, T2*>
{
public:
    Data() {cout<<"Data<T1*, T2*>" <<endl;}

private:
    T1 _d1;
    T2 _d2;
};

//两个参数偏特化为引用类型
template <typename T1, typename T2>
class Data <T1&, T2&>
{
public:
    Data(const T1& d1, const T2& d2)
        : _d1(d1)
        , _d2(d2)
    {
            cout<<"Data<T1&, T2&>" <<endl;
    }

private:
    const T1 & _d1;
    const T2 & _d2;
 };

void test2 ()
{
     Data<double , int> d1;       // 调用特化的int版本
     Data<int , double> d2;       // 调用基础的模板
     Data<int *, int*> d3;        // 调用特化的指针版本
     Data<int&, int&> d4(1, 2);   // 调用特化的指针版本
}
```

6 模板分离编译

6.1 什么是分离编译

> 一个程序（项目）由若干个源文件共同实现，而每个源文件单独编译生成目标文件，最后将所有目标文件链接起来形成单一的可执行文件的过程称为分离编译模式。

6.2 模板的分离编译

> 假如有以下场景，模板的声明与定义分离开，在头文件中进行声明，源文件中完成定义：

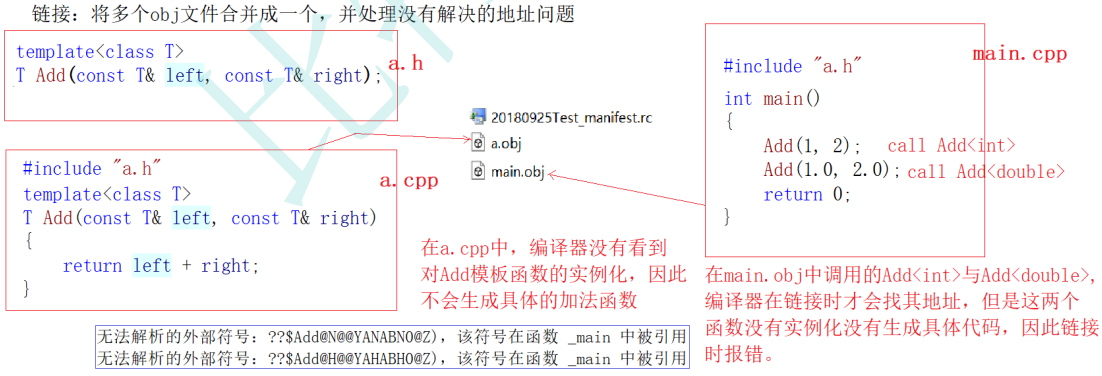
```
// a.h
template<class T>
T Add(const T& left, const T& right);

// a.cpp
template<class T>
T Add(const T& left, const T& right)
{
 return left + right;
}

// main.cpp
#include"a.h"
int main()
{
 Add(1, 2);
 Add(1.0, 2.0);

 return 0;
}
```

分析：

C/C++程序要运行，一般要经历一下步骤：
预处理 ---> 编译 ---> 汇编 ---> 链接

编译：对程序按照语言特性进行词法、语法、语义分析，错误检查无误后生成汇编代码
注意头文件不参与编译 编译器对工程中的多个源文件是分离开单独编译的。

链接：将多个obj文件合并成一个，并处理没有解决的地址问题



6.3 解决方法

> 1. **将声明和定义放到一个文件 "xxx.hpp" 里面或者xxx.h其实也是可以的。**推荐使用这种。
> 2. **模板定义的位置显式实例化。**这种方法不实用，不推荐使用。
>
> **【分离编译扩展阅读】** http://blog.csdn.net/pongba/article/details/19130

7. 模板总结

> **【优点】**
>
> 1. **模板复用了代码，节省资源，更快的迭代开发，C++的标准模板库(STL)因此而产生**
> 2. **增强了代码的灵活性**

【缺陷】

1. **模板会导致代码膨胀问题，也会导致编译时间变长**
2. **出现模板编译错误时，错误信息非常凌乱，不易定位错误**