6.模板初阶

1. 泛型编程

> 如何实现一个通用的交换函数呢?

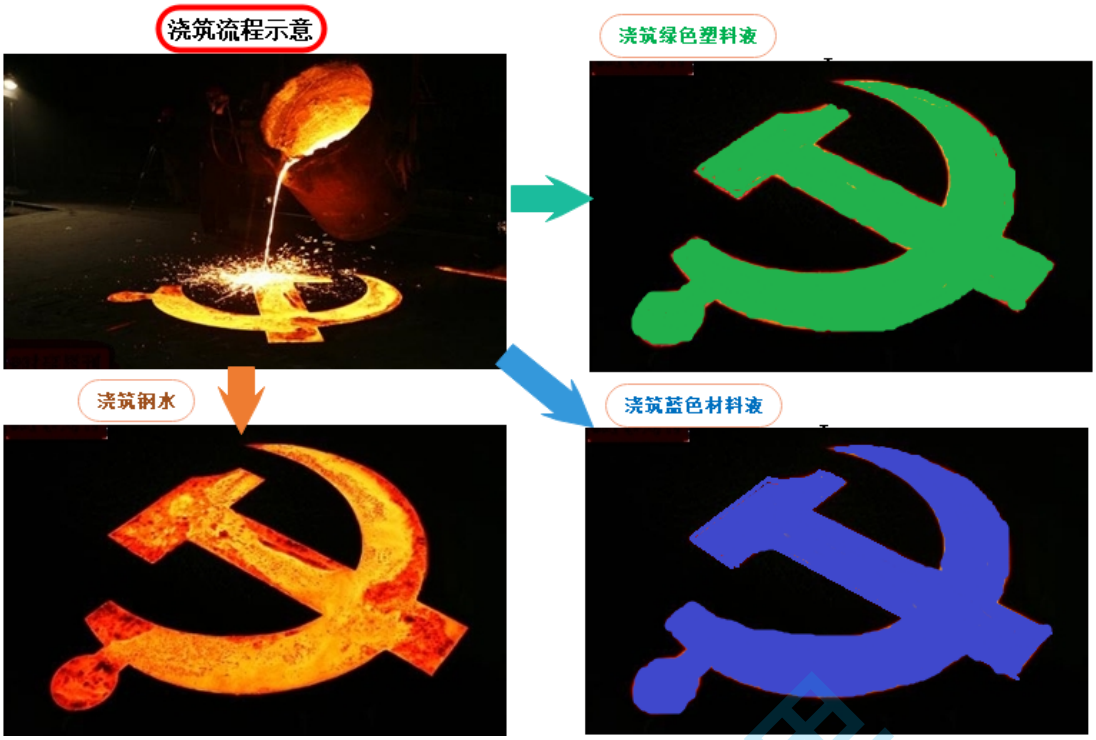
```
void Swap(int& left, int& right)
{
  int temp = left;
  left = right;
  right = temp;
}

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

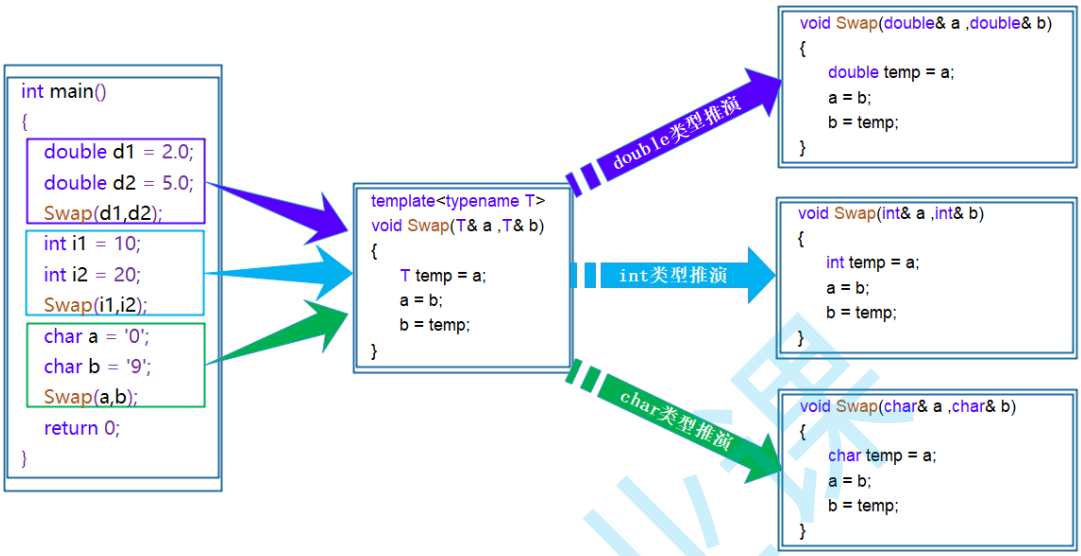
> **template<typename T1, typename T2,……,typename Tn>**
>
> **返回值类型　函数名(参数列表){}**

```
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

> **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
};
```

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

// 类模版
template<typename T>
class Stack
{
public:
    Stack(size_t capacity = 4)
    {
        _array = new T[capacity];
        _capacity = capacity;
        _size = 0;
    }

    void Push(const T& data);
private:
    T* _array;
    size_t _capacity;
    size_t _size;
};

// 模版不建议声明和定义分离到两个文件.h 和.cpp会出现链接错误，具体原因后面会讲
template<class T>
void Stack<T>::Push(const T& data)
```

```
{
    // 扩容
    _array[_size] = data;
    ++_size;
}

int main()
{
    Stack<int> st1;    // int
    Stack<double> st2; // double

    return 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。**

```
// Stack是类名，Stack<int>才是类型
Stack<int> st1;    // int
Stack<double> st2; // double
```