模版初阶复习

1. 泛型编程

> 对于功能相同但是参数类型不同的函数，如果对其进行函数重载，代码复用率较低，只要有新类型出现就要在进行函数重载；代码可维护性太低，一个储出错可能所有重载都出错。

所以C++中提供了模版这样一个**模具，**通过给这个不同的材料，来制作成不同材料的同样形状的物品。

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. 函数模版

2.1 函数模版概念

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。

2.2 函数模版格式

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

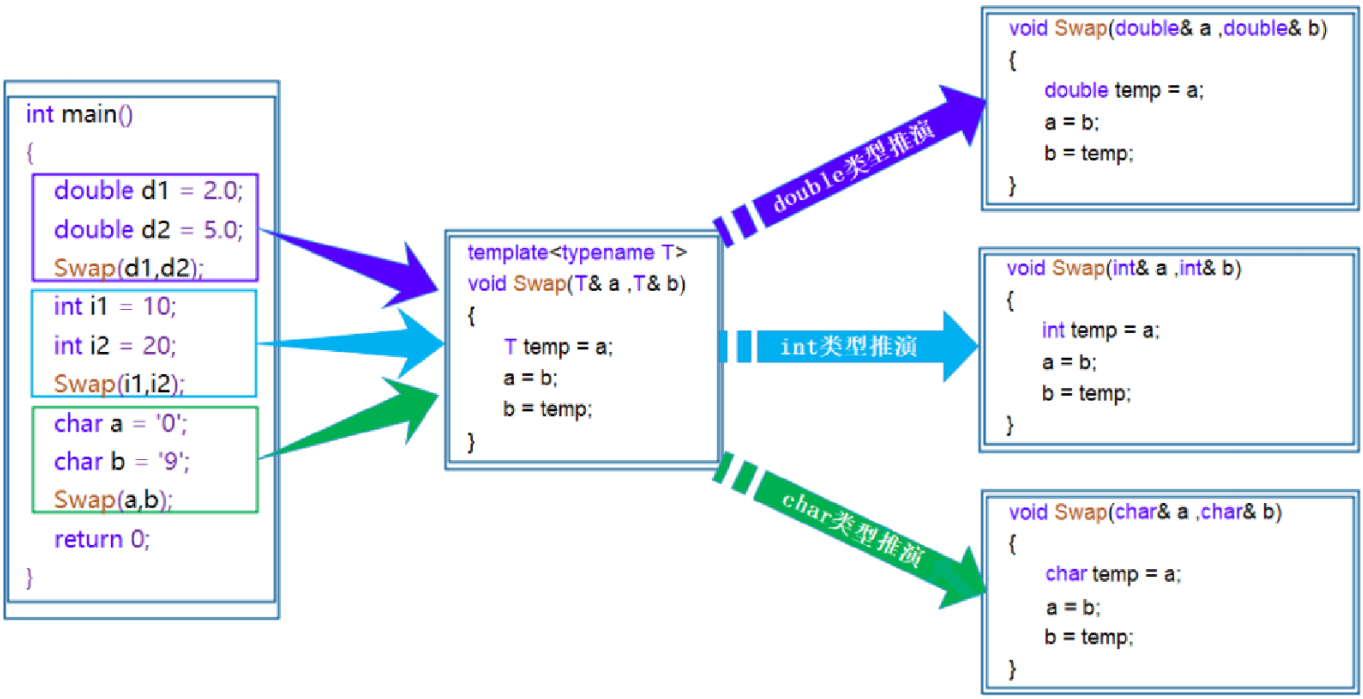
typename可以用class代替，但是不能用struct

返回值类型 函数名(参数列表){}

代码块

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

2.3 函数模版的原理



在编译器编译阶段，对于函数模版的使用，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。

2.4 函数模版的实例化

> **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版参数的实际类型

代码块

```
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
    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. 显示实例化：在函数名后的<>中指定模版参数的实际类型

代码块

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

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

2.5 模版参数的匹配原则

- **一个非模版函数可以和一个同名的函数模版同时存在，而且该函数模版还可以被实例化为这个非模版函数。**

  代码块

  ```
  // 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)
  {
  ```

```
11      return left + right;
12  }
13
14  void Test()
15  {
16      Add(1, 2);      // 与非模板函数匹配，编译器不需要特化
17      Add<int>(1, 2); // 调用编译器特化的Add版本
18  }
19
```

- **对于非模版函数和同名函数模版，如果其他条件都相同(区别仅在于模版函数的参数为输入实参)，在调用时会优先调用非模版函数而不会从该末班产生出一个实例。如果模版可以产生一个具有更好匹配的函数，那么将优先选择模版。**

代码块

```
1   // 专门处理int的加法函数
2   int Add(int left, int right)
3   {
4       return left + right;
5   }
6
7   // 通用加法函数
8   template<class T1, class T2>
9   T1 Add(T1 left, T2 right)
10  {
11      return left + right;
12  }
13
14  void Test()
15  {
16      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
17      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
18  }
19
```

- **模版函数不允许自动类型转换，但普通函数可以。**

3. 类模版

3.1 定义格式

代码块

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

代码块

```
#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
```

```
37        return 0;
38    }
39
```

3.2 类模版实例化

> 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**。

代码块

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