1.函数模板

1.1 函数模板概念

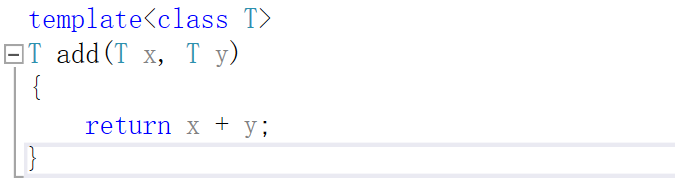
函数模板代表了一个函数家族，该函数模板与类型无关，在使用时被参数化，根据实参类型产生函数的特定类型版本。

1.2 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){}

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)



1.3 函数模板的原理

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。

1.4 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。

1.4.1 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(T x, T y)

{

return x + y;

}

int main()

{

int a1 = 2;

int b1 = 6;

int c1 = add(a1, b1);

double a2 = 3.4;

/\*

add(a1, a2);

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a2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

\*/

return 0;

}

解决办法：强制类型转换和显示实例化

1.4.2 显式实例化：在函数名后的<>中指定模板参数的实际类型

double c2 = add<int>(a1, a2);

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。如一个Time类，用Time类创建一个对象t

Add<int>(t,a1)报错

1.5 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。

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。

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

2.类模板

2.1 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

};

2.2 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。