1.函数模板

1.1 函数模板概念

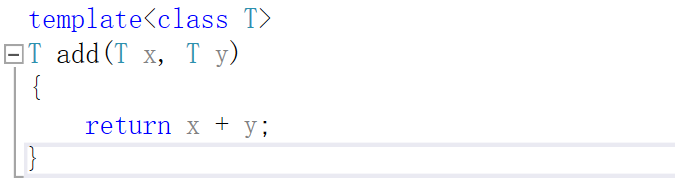
函数模板代表了一个函数家族，该函数模板与类型无关，在使用时被参数化，根据实参类型产生函数的特定类型版本。

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。

模板进阶

1. 非类型模板参数

1.1 模板参数分为类类型形参与非类型形参。

①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。

②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。

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。

2.模板的特化

2.1 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，比如

template<class T>

bool IsEqual(T& left, T& right)

{

return left == right;

}

注意：如果用该函数比较字符串，则比较的是字符串的的地址大小。

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。

2.2 函数模板特化

2.2.1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。

template<>

bool IsEqual<char\*>(char\*& left, char\*& right)

{

if(strcmp(left, right) > 0)

return true;

return false;

}

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。

bool IsEqual(char\* left, char\* right)

{

if(strcmp(left, right) > 0)

return true;

return false;

}

2.3 类模板特化

2.3.1 全特化

全特化即是将模板参数类表中所有的参数都确定化

2.3.2 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

3. 类模板特化应用之类型萃取

3.1 使用memcpy拷贝

template<class T>

void Copy(T\* dst, const T\* src, size\_t size)

{

memcpy(dst, src, sizeof(T)\*size);

}

int main()

{

// 试试下面的代码

std::string strarr1[3] = { "11", "22", "33" };

std::string strarr2[3];

Copy(strarr2, strarr1, 3);

return 0;

}

上述代码虽然对于任意类型的空间都可以进行拷贝，但是如果拷贝自定义类型对象就可能会出错，因为自定义类型对象有可能会涉及到深拷贝(比如string)，而memcpy属于浅拷贝（若拷贝的资源内有指针，则拷贝的是地址而不是指针所指向的资源）。如果对象中涉及到资源管理，就只能用赋值。

3.2 使用赋值方式拷贝

缺点：循环赋值的方式虽然可以，但是代码的效率比较低

3.3 增加bool类型区分自定义与内置类型

template<class T>

void Copy(T\* dst, const T\* src, size\_t size, bool IsPODType)

{

if(IsPODType)

memcpy(dst, src, sizeof(T)\*size);

else

{

for(size\_t i = 0; i < size; ++i)

dst[i] = src[i];

}

}

template<class T>

void Copy(T\* dst, const T\* src, size\_t size)

{

for(size\_t i = 0; i < size; ++i)

{

dst[i] = src[i];

}

}

缺点：用户需要根据所拷贝元素的类型去传递第三个参数，那出错的可能性就增加。