**模板的作用可以让一个函数适应多种类型的实参**

**基本格式**

template <typename T>int compare(const T &v1,const T &v2){

T song；  **//这里我们可以看到T类型可以用在函数内部和返回值**

return T;

}

**由上面的例子我们可以看出**

**我们的函数需要**

template <typename T>  **//这个东西叫做 模板类型参数**

template <unsigend m>  **//非模板类型参数**

**实例：**

template <unsigend m> int compare（const （&p1）【m】）{

return 0；

}  **//如果我们传入 compare（“ch”）m就会自动翻译成3，m不是一种类型，是一个值**

**我们还需要注意我们的模板在运行阶段才把模板实例化再带入到代码中去**

**类的模板**

#include<iostream>

#include<string>

using namespace std;

template <typename T>class Myclass

{

public:

Myclass(string b);

~ Myclass();

template<typename x> friend tring printfs1(x a);  **//这个是声明的友元**

template<typename T> friend string printfs2(Myclass<T>& my);

private:

static string a;

string printfs(T& da);

};

template<typename T> string a = "0";  **//静态变量的声明方式**

template <typename T>string Myclass<T>::printfs(T& da) {

cout << a;

}

template <typename T> Myclass<T>:: Myclass(string b):a(b) **//注意在外部时Myclass<T>而不是Myclass；**

{

}

template<typename T> Myclass<T>::~ Myclass()

{

}

template<typename x>string printfs1(x b) {

cout << b;

}

template<typename T>string printfs2(Myclass<T>& my){

cout << a << "000";

}

**我们一般认为模板参数定义的类型与普通的作用域相同**

**如果我们要在类外调用类的成员注意加上typename**

template <typename T> **typename** T::value\_type tpr(const T& a)

**类模板的成员函数**

**类模板不能是虚函数**

template <typename T> class song{

template<typename it> song(it abc,it cdn);

}

**注意我们在类的外部声明类模板的模板函数**

template <typename T>

template<typename it> song(it abc,it cdn){}；

**如果很多文件多次实例化同一个模板就会造成一种浪费所以我们可以显示实例化 P598**

extern template class song <int>;  **实例化声明**

template int compare<const int & >  **实例化定义**

**模板参数可以做到类型转化 p602**

**const转化 数组和函数向指针转换**

**如果一个模板参数T用两个不同类型实例化就会报错**

**我们可以显示化模板**

template<typename T1，typename T2,typename T3>T1 song (T2 \*a,T3);

song<long long,int ,long> a ; //显示化模板

**这样使用更加精确化不用编辑器推断出来；**

**注意这里有一个很重要的知识点**

尾置返回类型和类型转化

**有的时候我们不知道返回类型时候我们就可以使用**尾置返回类型

template <typename T>

auto fcn（T ig， T end）->delctype(\*beg){

return \*beg;

}

**我们还可以使用标准库来知道我们需要的类型**

**头文件 type\_traits**

**remove\_reference<int&>::type 这个type会被知道为int类型**

template <typename T>

auto fcn（T ig， T end）

**typename** remove\_reference<decltype(\*beg)>::type  **//remove\_reference依旧是一个类模板**{

；return \*beg;

}

**重载与模板**

**在调用一个函数时，如果有多个函数可以与之匹配那么我们优先选择非模板函数。**

song（const string \*& p）； song（const string \*）；

string \* a=“大师傅似的”；

song（a）**；两个函数都可以但是我们一般选择特例化的函数也就是**song（const string\*a）；

**重载模板和类型转换**

**我们还是依据以前学习的匹配机制**

**我们先注意形参的精确匹配**

**数组类型和函数类型向指针转换 ，，去掉顶层const**

可变参数模板

**可变参数被称为参数包**

**有两种参数包 模板参数包和函数参数包**

template<typename T> void print( T& st) {

cout << st;  **//此函数是用来结束下面函数的迭代的**

}

template<typename T, typename ...args >  **//这个函数介绍了如何把这个args展开**

void print( T t, args& ... rest) {

cout << t;

cout << sizeof...(rest)**;//计算参数的数目；**

print(rest...);  **//注意我们这里是将rest的第一个参数绑定到T上面然后不断的迭代循环**

}

**我们要理解参数包是如何展开的 包扩展**

**上面最近的例题** ...args**会扩展成一个类的列表**  ...rest**会扩展成一个参数的列表**

所以debug\_rep(rest)...**正确**  debug\_rep(rest...)**错误 是因为第二个扩展成了**debug\_rep(a,b,c,d)**但我们是想循环**debug\_rep(a),debug\_rep(b)**所以我们要理解**

**转发参数包**

**就是我们在该模板内使用别的函数 同时把模板参数当成函数的实参**

std：：forward<args>(rest)...（**还是原来的函数例题）**

**这个就被翻译成**std：：forward<int ，long，short>(rest)...  **所以很实用**

**模板特例化**

**为了更好的让函数功能化**

template<typename T> void song (T & a);

**如果我们直接实例化** song<int >(12);

但如果我们特例化就可以

template<> void song (const int \*const);

**所以我们可以看出我们使函数更加丰富化同时看出特例化就是函数实例化的特殊版本**

**类的特例化**

**如果该类是标准库的类就要加上**

namespace std{

template<>

class song<int>{

**如果想要部分实例化我们就正常写类模板然后加上我们想要写的函数**

template<> void song (const int \*const);

};

}