四、模板的其它特性

1.模板参数的缺省值

1)模板参数可以带有缺省值。如果某个模板参数带有缺省值，那么它后面的所有模板参数必须都带有缺省值。

2)与函数调用参数的缺省值不同，模板参数的缺省值可以来自其前面的参数。

注解：函数的缺省值是在程序运行时，调用函数生成压栈指令时使用的；模板的缺省值是编译时编译器自行处理的，没有压栈的过程。

----------------

3)C++98不允许为函数模板的模板参数指定缺省值，但是C++2011可以。

4)如果函数模板的模板参数的缺省值与隐式推断的类型不一致，以隐式推断的类型为准，忽略其缺省值。

-----------

GCC < 4.8 : -std=c++0x

GCC >= 4.8 : -std=c++11

参见：defarg.cpp

-------------------------

2.非类型模板参数(数值参数)

1)可以为类模板或函数模板指定数值型模板参数，即非类型参数，但是可传递给非类型形参的实参必须是常量、常量表达式，或者是带有常属性(const，C限定)的变量，但是不能同时拥有挥发性(volatile，V限定)。

参见：Kfarg.cpp

---------------------------

3.typename关键字作用

1)声明模板的类型形参

template<typename T> ...

typename意在说明其后的标示符T表示一个类型形参。这样的typename也可以被替换为class关键字。

2)解决类型嵌套

在一个类模板或者函数模板中，有时候需要引用某种类型的嵌套类型(内部类型)，但是包装该嵌套类型的外部类型又依赖于该模板的类型参数，这种现象被称为嵌套依赖。一旦出现嵌套依赖，编译器在第一次编译该模板时，因为无法确定依赖于类型参数的外部类型，就会将嵌套于该外部类型的内部类型假定为其静态成员变量，导致编译失败。这时，为这些被引用的嵌套类型加上typename关键字，意在告诉编译器，它们不是静态成员变量，而是某种类型，可以用于其它变量的声明。这样一来，编译器就会把有关嵌套类型的检查工作推迟到第二次编译时完成，避免编译错误。

参见：inner.cpp

------------------------

补充：3个关键字的作用(struct,class,typename)

声明类： struct,class

声明模板的类型参数： typename,class

解决嵌套依赖： typename(指明其后的标识符表示类型形参)

---------------------------------------------

4.template关键字作用

1)声明函数模板或类模板

template<...> ...

2)解决模板嵌套

在一个函数模板或类模板中，引用某个依赖于模板参数的模板类型的内部模板是，会因为编译器无法识别外部模板的类型而导致编译失败。通过template关键字可以向编译器显式指明所引用的内部标识是一个模板，避免编译器对模板参数表产生误解，解决编译错误。

参见：template.cpp

-----------------------------------

5.在子类模板中访问基类模板的成员

在子类模板中直接访问那些依赖于模板参数的基类模板的成员，编译器在第一次编译时，通常会因为基类类型不明确而只在子类和全局作用域中搜索所引用的符号。

解决方法：在它们前面加上(1)作用域限定符或(2)this->。

参见：inherit.cpp

-------------------------

6.内部模板的外部定义

参见：outer.cpp

-------------------------

7.模板型模板参数(与模板型成员变量的区别)

(1)类模板的模板参数本身又可以是个模板

(2)该形参不用typename关键字声明，必须使用template关键字结合模板实参的原型进行声明

(3)可以使用类模板的其它参数，实例化其模板型模板参数(或者使用缺省值)

template<typename T, template<typename> class C>

(4)某些编译器，函数模板不能使用模板型模板参数。

参见：stack.cpp

----------------------

8.零初始化

1)基本类型不存在缺省构造函数，未被显式初始化的局部变量都具有一个不确定的值。

2)包含(自定义的或系统提供)缺省构造函数的类，在未被显式初始化的情况下，都会有一个确定的缺省初始化状态。

3)在模板的实现中就会产生不一致的语法语义：

T var; // T是基本类型，var没初始化

// T是类类型，var缺省初始化

4)为了使模板对不同类型表现出的一致性，就需要采用零初始化语法：

T var = T( );

T=int -> int var = int( ); // 和类型对应的零初始化

T=string -> string var = string( ); // 缺省构造

参见：init.cpp

------------------------------

9.类模板和虚函数

1)在类模板中可以声明虚函数，而且也可以表现出多态性，但是在该模板的实例化过程中不能违背虚函数有效覆盖的条件，否则将丧失多态性，甚至可以引发编译错误(相同类型不同返回值时的覆盖失败报错)。

2)模板函数不能被声明为虚函数。基于虚函数的多态机制，需要一个名为虚函数表的函数指针数组。该数组在类被编译或类模板被实例化的过程中产生，而此时那些模板形式的成员函数尚未被实例化，其入口地址和重载版本的个数，要等到编译器处理完对该函数的所有调用以后才能确定。成员函数模板的延迟编译阻碍了虚函数表的静态构建。

3)广义多态：通过一种类型的一种方法表现出多种不同的行为。

A.动态多态，基于虚函数和动态绑定的多态。(动态，因为自动转变)

B.静态多态，基于类模板和类型参数的多态。(静态，因为需要给出具体类型参数)

比较：

动态多态：运行期处理，代码复杂，运行速度慢，生成二进制文件小

静态多态：编译期完成，代码简单，运行速度快，生成二进制文件大

参见：dpoly.cpp、spoly.cpp

================================

10.编译模型(因为模板的延迟编译，才会有那么多事)

1)单一模型：将模板的声明、实现和实例化放在同一个源文件中。

优点：编译简单。

缺点：源文件规模较大，不易于维护，不易于协作开发。

2)分离模型：将模板的声明、实现和实例化分别放在不同的源文件中。

优点：源文件规模可控，易于维护和协作开发。

缺点：链接障碍，分离后只会分别做第一次编译。

参见：分离模型/

\*\*\*\*\*\*\*\*\*\*\*\*

3)包含模型：在模板的声明文件尾部包含该模板的实现文件，促使预编译器将模板的声明、实现和实例化代码放到同一个编译单元中，避免分离模型因为模板代码得不到二次编译而遇到的链接障碍(处理后，效果等同与单一模型)。

优点：链接成功。

缺点：延长编译时间，模板的实现源码必须公开(目标文件(.o)或库文件(.a/.so)无效)。

--------------

注解：一般头文件只包含数据的声明，这些头文件中包含了模板的定义，每个包含该头文件的文件都会单独编译，需要对整个模板库的实现做语法检查，如果多个文件都要使用到该模板库，就都要包含该头文件，并进行单独编译，这样编译器就会对整个模板库做重复的语法检查(一次编译)。

参见：包含模型/

4)实例模型：在模板实现文件中以显式实例化的方式，强制编译器提前对所实现的模板做二次编译，解决分离模型的链接问题。

优点：不会延长编译时间，模板的实现源码可以不公开。

缺点：类型于受限，不适于构建通用模板库。

参见：实例模型/

\*\*\*\*\*\*\*\*\*\*\*\*

5)导出模型：通过export关键字将模板声明为导出，迫使编译器将被导出模板的内部表示保存在目标文件中，待到链接阶段再补充二次编译的过程，顺利通过链接。

优点：不会延长编译时间，模板的实现源码可以不公开，类型不受限制。

缺点：绝大多数的C++编译器都不支持导出模型。

参见：导出模型/

------------------------------

不支持的编译器： GNU/Microsoft/IBM/Oracle/HP

支持的编译器： Edison Design Ltd.

C++2011将导出模型删除了！

----------------------------

11.预编译头

(1)将一些公共头文件和含有模板实现的文件包含在一个独立的头文件中，比如：common.h

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include "cmp.h"（包含模板实现）

(2)编译该头文件：

g++ -c common.h -> common.h.gch

1. 每个使用到common.h中内容的文件的第一行包含该头文件
2. 编译器会以加载预编译头文件common.h.gch中内容的方式代替对头文件的扩展和编译，从而缩短编译时间。

注意：任何对预编译头文件的修改都必须反映到编译结果中，否则修改无效。

微软：

afxstd.h

afxstd.cpp -> <项目>.pch