类模板的编译

首先，C++标准中提到，一个编译单元[translation unit]是指一个.cpp文件以及它所include的所有.h文件，.h文件里的代码将会被扩展到包含它的.cpp文件里，然后编译器编译该.cpp文件为一个.obj文件，后者拥有PE[Portable Executable,即windows可执行文件]文件格式，并且本身包含的就已经是二进制码，但是，不一定能够执行，因为并不保证其中一定有main函数。当编译器将一个工程里的所有.cpp文件以分离的方式编译完毕后，再由连接器(linker)进行连接成为一个.exe文件。

举个例子：

//---------------test.h-------------------//

void f();//这里声明一个函数f

//---------------test.cpp--------------//

#include”test.h”

void f()

{

…//do something

} //这里实现出test.h中声明的f函数

//---------------main.cpp--------------//

#include”test.h”

int main()

{

f(); //调用f，f具有外部连接类型

}

在这个例子中，test. cpp和main.cpp各被编译成为不同的.obj文件[姑且命名为test.obj和main.obj]，在main.cpp中，调用了f函数，然而当编译器编译main.cpp时，它所仅仅知道的只是main.cpp中所包含的test.h文件中的一个关于void f();的声明，所以，编译器将这里的f看作外部连接类型，即认为它的函数实现代码在另一个.obj文件中，本例也就是test.obj，也就是说，main.obj中实际没有关于f函数的哪怕一行二进制代码，而这些代码实际存在于test.cpp所编译成的test.obj中。在main.obj中对f的调用只会生成一行call指令，像这样：

call f [C++中这个名字当然是经过mangling[处理]过的]

在编译时，这个call指令显然是错误的，因为main.obj中并无一行f的实现代码。那怎么办呢？这就是连接器的任务，连接器负责在其它的.obj中[本例为test.obj]寻找f的实现代码，找到以后将call f这个指令的调用地址换成实际的f的函数进入点地址。需要注意的是：连接器实际上将工程里的.obj“连接”成了一个.exe文件，而它最关键的任务就是上面说的，寻找一个外部连接符号在另一个.obj中的地址，然后替换原来的“虚假”地址。

这个过程如果说的更深入就是：

call f这行指令其实并不是这样的，它实际上是所谓的stub，也就是一个

jmp 0x23423[这个地址可能是任意的，然而关键是这个地址上有一行指令来进行真正的call f动作。也就是说，这个.obj文件里面所有对f的调用都jmp向同一个地址，在后者那儿才真正”call”f。这样做的好处就是连接器修改地址时只要对后者的call XXX地址作改动就行了。但是，连接器是如何找到f的实际地址的呢[在本例中这处于test.obj中]，因为.obj于.exe的格式都是一样的，在这样的文件中有一个符号导入表和符号导出表[import table和export table]其中将所有符号和它们的地址关联起来。这样连接器只要在test.obj的符号导出表中寻找符号f[当然C++对f作了mangling]的地址就行了，然后作一些偏移量处理后[因为是将两个.obj文件合并，当然地址会有一定的偏移，这个连接器清楚]写入main.obj中的符号导入表中f所占有的那一项。

这就是大概的过程。其中关键就是：

编译main.cpp时，编译器不知道f的实现，所有当碰到对它的调用时只是给出一个指示，指示连接器应该为它寻找f的实现体。这也就是说main.obj中没有关于f的任何一行二进制代码。

编译test.cpp时，编译器找到了f的实现。于是乎f的实现[二进制代码]出现在test.obj里。

连接时，连接器在test.obj中找到f的实现代码[二进制]的地址[通过符号导出表]。然后将main.obj中悬而未决的call XXX地址改成f实际的地址。

完成。

然而，对于模板，你知道，模板函数的代码其实并不能直接编译成二进制代码，其中要有一个“具现化”的过程。举个例子：

//----------main.cpp------//

template<class T>

void f(T t)

{}

int main()

{

…//do something

f(10); //call f<int> 编译器在这里决定给f一个f<int>的具现体

…//do other thing

}

也就是说，如果你在main.cpp文件中没有调用过f，f也就得不到具现，从而main.obj中也就没有关于f的任意一行二进制代码！！如果你这样调用了：

f(10); //f<int>得以具现化出来

f(10.0); //f<double>得以具现化出来

这样main.obj中也就有了f<int>,f<double>两个函数的二进制代码段。以此类推。

然而具现化要求编译器知道模板的定义，不是吗？

看下面的例子：[将模板和它的实现分离]

//-------------test.h----------------//

template<class T>

class A

{

public:

void f(); //这里只是个声明

};

//---------------test.cpp-------------//

#include”test.h”

template<class T>

**void A<T>::f() //模板的实现，但注意：不是具现**

{

…//do something

}

//---------------main.cpp---------------//

#include”test.h”

int main()

{

A<int> a;

a. f(); //编译器在这里并不知道A<int>::f的定义，因为它不在test.h里面

//于是编译器只好寄希望于连接器，希望它能够在其他.obj里面找到

//A<int>::f的实现体,在本例中就是test.obj，然而，后者中真有A<int>::f的

//二进制代码吗？NO！！！因为C++标准明确表示，当一个模板不被用到的时

//侯它就不该被具现出来，test.cpp中用到了A<int>::f了吗？没有！！所以实

//际上test.cpp编译出来的test.obj文件中关于A::f的一行二进制代码也没有

//于是连接器就傻眼了，只好给出一个连接错误

//但是，如果在test.cpp中写一个函数，其中调用A<int>::f，则编译器会将其//具现出来，因为在这个点上[test.cpp中]，编译器知道模板的定义，所以能//够具现化，于是，test.obj的符号导出表中就有了A<int>::f这个符号的地

//址，于是连接器就能够完成任务。

}

关键是：**在分离式编译的环境下，编译器编译某一个.cpp文件时并不知道另一个.cpp文件的存在**，**也不会去查找**[当遇到未决符号时它会寄希望于连接器]。这种模式在没有模板的情况下运行良好，但遇到模板时就傻眼了，因为模板仅在需要的时候才会具现化出来，所以，当编译器只看到模板的声明时，它不能具现化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。然而当实现该模板的.cpp文件中没有用到模板的具现体时，编译器懒得去具现，所以，整个工程的.obj中就找不到一行模板具现体的二进制代码，于是连接器也黔

/////////////////////////////////

http://dev.csdn.net/develop/article/19/19587.shtm

C++模板代码的组织方式 ——包含模式（Inclusion Model） 选择自 sam1111 的 Blog

关键字 Template Inclusion Model

出处 C++ Template: The Complete Guide

说明：本文译自《C++ Template: The Complete Guide》一书的第6章中的部分内容。最近看到C++论坛上常有关于模板的包含模式的帖子，联想到自己初学模板时，也为类似的问题困惑过，因此翻译此文，希望对初学者有所帮助。

模板代码有几种不同的组织方式，本文介绍其中最流行的一种方式：包含模式。

链接错误

大多数C/C++程序员向下面这样组织他们的非模板代码：

·类和其他类型全部放在头文件中，这些头文件具有.hpp（或者.H, .h, .hh, .hxx）扩展名。

·对于全局变量和（非内联）函数，只有声明放在头文件中，而定义放在点C文件中，这些文件具有.cpp（或者.C, .c, .cc, .cxx）扩展名。

这种组织方式工作的很好：它使得在编程时可以方便地访问所需的类型定义，并且避免了来自链接器的“变量或函数重复定义”的错误。

由于以上组织方式约定的影响，模板编程新手往往会犯一个同样的错误。下面这一小段程序反映了这种错误。就像对待“普通代码”那样，我们在头文件中定义模板：

// basics/myfirst.hpp

#ifndef MYFIRST\_HPP

#define MYFIRST\_HPP

// declaration of template

template <typename T>

void print\_typeof (T const&);

#endif // MYFIRST\_HPP

print\_typeof()声明了一个简单的辅助函数用来打印一些类型信息。函数的定义放在点cpp文件中：

// basics/myfirst.cpp

#include <iostream>

#include <typeinfo>

#include "myfirst.hpp"

// implementation/definition of template

template <typename T>

void print\_typeof (T const& x)

{

std::cout << typeid(x).name() << std::endl;

}

这个例子使用typeid操作符来打印一个字符串，这个字符串描述了传入的参数的类型信息。

最后，我们在另外一个点C文件中使用我们的模板，在这个文件中模板声明被#include：

// basics/myfirstmain.cpp

#include "myfirst.hpp"

// use of the template

int main()

{

double ice = 3.0;

print\_typeof(ice); // call function template for type double

}

大部分C++编译器（Compiler）很可能会接受这个程序，没有任何问题，但是链接器（Linker）大概会报告一个错误，指出缺少函数print\_typeof()的定义。

这个错误的原因在于，模板函数print\_typeof()的定义还没有被具现化（instantiate）。为了具现化一个模板，编译器必须知道哪一个定义应该被具现化，以及使用什么样的模板参数来具现化。不幸的是，在前面的例子中，这两组信息存在于分开编译的不同文件中。因此，当我们的编译器看到对print\_typeof()的调用，但是没有看到此函数为double类型具现化的定义时，它只是假设这样的定义在别处提供，并且创建一个那个定义的引用（链接器使用此引用解析）。另一方面，当编译器处理myfirst.cpp时，该文件并没有任何指示表明它必须为它所包含的特殊参数具现化模板定义。

头文件中的模板

**解决上面这个问题的通用解法是，采用与我们使用宏或者内联函数相同的方法：我们将模板的定义包含进声明模板的头文件中**。对于我们的例子，我们**可以通过将#include "myfirst.cpp"添加到myfirst.hpp文件尾部，或者在每一个使用我们的模板的点C文件中包含myfirst.cpp文件，来达到目的。**当然，**还有第三种方法**，**就是删掉myfirst.cpp文件，并重写myfirst.hpp文件，使它包含所有的模板声明与定义**：

// basics/myfirst2.hpp

#ifndef MYFIRST\_HPP

#define MYFIRST\_HPP

#include <iostream>

#include <typeinfo>

// declaration of template

template <typename T>

void print\_typeof (T const&);

// implementation/definition of template

template <typename T>

void print\_typeof (T const& x)

{

std::cout << typeid(x).name() << std::endl;

}

#endif // MYFIRST\_HPP

这种组织模板代码的方式就称作包含模式。经过这样的调整，你会发现我们的程序已经能够正确编译、链接、执行了。

从这个方法中我们可以得到一些观察结果。最值得注意的一点是，这个方法在相当程度上增加了包含myfirst.hpp的开销。在这个例子中，这种开销并不是由模板定义自身的尺寸引起的，而是由这样一个事实引起的，即我们必须包含我们的模板用到的头文件，在这个例子中是<iostream>和<typeinfo>。你会发现这最终导致了成千上万行的代码，因为诸如<iostream>这样的头文件也包含了和我们类似的模板定义。

这在实践中确实是一个问题，因为它增加了编译器在编译一个实际程序时所需的时间。我们因此会在以后的章节中验证其他一些可能的方法来解决这个问题。但无论如何，现实世界中的程序花一小时来编译链接已经是快的了（我们曾经遇到过花费数天时间来从源码编译的程序）。

**抛开编译时间不谈，我们强烈建议如果可能尽量按照包含模式组织模板代码。**

另一个观察结果是，非内联模板函数与内联函数和宏的最重要的不同在于：它并不会在调用端展开。相反，当模板函数被具现化时，会产生此函数的一个新的拷贝。由于这是一个自动的过程，编译器也许会在不同的文件中产生两个相同的拷贝，从而引起链接器报告一个错误。理论上，我们并不关心这一点：这是编译器设计者应当关心的事情。实际上，大多数时候一切都运转正常，我们根本就不用处理这种状况。然而，对于那些需要创建自己的库的大型项目，这个问题偶尔会显现出来。

**最后，需要指出的是，在我们的例子中，应用于普通模板函数的方法同样适用于模板类的成员函数和静态数据成员，以及模板成员函数**