# 关于C++中的extern "C"

本文将介绍C++中如何通过extern "C"关键字支持C语言。

**主要内容：**

**简介**

**1，问题的引出**

**2，关于extern "C"**

**3，用法举例**

**总结**

**简介**

**===============**

C++语言的创建初衷是"a better C"，但是这并不意味着C++中类似C语言的全局变量和函数所采用的编译和连接方式与C语言完全相同。作为一种欲与C兼容的语言，C++保留了一部分过程式语言的特点（被世人称为"不彻底地面向对象"），因而它可以定义不属于任何类的全局变量和函数。但是，C++毕竟是一种面向对象的程序设计语言，为了支持函数的重载，C++对全局函数的处理方式与C有明显的不同。

**1，问题的引出**

**===============**

（1）某企业曾经给出如下的一道面试题：

为什么标准头文件都有类似以下的结构？

//incvxworks.h

#ifndef \_\_INCvxWorksh

#define \_\_INCvxWorksh

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*...\*/

#ifdef \_\_cplusplus

}

#endif

#endif /\* \_\_INCvxWorksh \*/

（2）问题分析

对于上面问题，显然，头文件中的编译宏“#ifndef \_\_INCvxWorksh、#define \_\_INCvxWorksh、#endif” 的作用是防止该头文件被重复引用。

那么，

#ifdef \_\_cplusplus

extern "C" {

#endif

和

#ifdef \_\_cplusplus

}

#endif

的作用又是什么呢？我们将在后面对此进行详细说明。

**2，关于extern "C"**

**===============**

前面的题目中的\_\_cplusplus宏，是用来识别编译器的，也就是，将当前代码编译的时候，是否将代码作为C++进行编译。如果是，则定义了\_\_cplusplus宏。更多内容，这里就不详细说明了。

而题目中的extern "C" 包含双重含义，从字面上即可得到：首先，被它修饰的目标是"extern"的；其次，被它修饰的目标是"C"的。具体如下：

（1）被extern "C"限定的函数或变量是extern类型的。

extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。注意，语句"extern int a;" 仅仅是对变量的声明，其并不是在定义变量a，声明变量并未为a分配内存空间。定义语句形式为"int a;"，变量a在所有模块中作为一种全局变量只能被定义一次，否则会出现连接错误。

在引用全局变量和函数之前，必须要有这个变量或者函数的声明（或者定义）。通常，在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字extern声明。例如，如果模块B欲引用该模块A中定义的全局变量和函数时只需包含模块A的头文件即可。这样，模块B中调用模块A中的函数时，在编译阶段，模块B虽然找不到该函数，但是并不会报错；它会在连接阶段中从模块A编译生成的目标代码中找到此函数。

与extern对应的关键字是static，被它修饰的全局变量和函数只能在本模块中使用。因此，一个函数或变量只可能被本模块使用时，其不可能被extern "C"修饰。

（2）被extern "C"修饰的变量和函数是按照C语言方式编译和连接的。

(2.1)首先看看C++中，在未加extern "C"声明时，对类似C的函数是怎样编译的。

作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。所以，函数被C++编译后在符号库中的名字与C语言的有所不同。例如，假设某个函数的原型为：

void foo( int x, int y );

该函数被C编译器编译后在符号库中的名字为\_foo，而C++编译器则会产生像\_foo\_int\_int之类的名字（不同的编译器可能生成的名字不同，但是都采用了相同的机制，生成的新名字称为"mangled name"）。\_foo\_int\_int这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。例如，在C++中，函数void foo( int x, int y )与void foo( int x, float y )编译生成的符号是不相同的，后者为\_foo\_int\_float。

同样地，C++中的变量除支持局部变量外，还支持类成员变量和全局变量。用户所编写程序的类成员变量可能与全局变量同名，我们以"."来区分。而本质上，编译器在进行编译时，与函数的处理相似，也为类中的变量取了一个独一无二的名字，这个名字与用户程序中同名的全局变量名字不同。

(2.2)其次，看看在未加extern "C"声明时，是如何连接的。

假设在C++中，模块A的头文件如下：

//模块A头文件　moduleA.h

#ifndef MODULE\_A\_H

#define MODULE\_A\_H

int foo( int x, int y );

#endif

在模块B中引用该函数：

// 模块B实现文件　moduleB.cpp

#include "moduleA.h"

foo(2,3);

实际上，在连接阶段，连接器会从模块A生成的目标文件moduleA.obj中寻找\_foo\_int\_int这样的符号！

对于上面例子，如果B模块是C程序，而A模块是C++库头文件的话，会导致链接错误；同理，如果B模块是C++程序，而A模块是C库的头文件也会导致错误。

(2.3)再者，看看加extern "C"声明后的编译和连接方式

加extern "C"声明后，模块A的头文件变为：

// 模块A头文件　moduleA.h

#ifndef MODULE\_A\_H

#define MODULE\_A\_H

extern "C" int foo( int x, int y );

#endif

在模块B的实现文件中仍然调用foo( 2,3 )，其结果，将会是C语言的编译连接方式：

(a)模块A编译生成foo的目标代码时，没有对其名字进行特殊处理，采用了C语言的方式；

(b)连接器在为模块B的目标代码寻找foo(2,3)调用时，寻找的是未经修改的符号名\_foo。

如果在模块A中函数声明了foo为extern "C"类型，而模块B中包含的是extern int foo( int x, int y ) ，则模块B找不到模块A中的函数(因为这样的声明没有使用extern "C"指明采用C语言的编译链接方式)；反之亦然。

所以，综上可知，extern "C"这个声明的真实目的，就是实现C++与C及其它语言的混合编程。

**3，用法举例**

**===============**

（1）C++引用C函数的具体例子

在C++中引用C语言中的函数和变量，在包含C语言头文件（假设为cExample.h）时，需进行下列处理：

extern "C"

{

#include "cExample.h"

}

因为，C库的编译当然是用C的方式生成的，其库中的函数标号一般也是类似前面所说的\_foo之类的形式，没有任何参数信息，所以当然在C++中，要指定使用extern "C"，进行C方式的声明（如果不指定，那么C++中的默认声明方式当然是C++方式的，也就是编译器会产生\_foo\_int\_int之类包含参数信息的、C++形式的函数标号，这样的函数标号在已经编译好了的、可以直接引用的C库中当然没有）。通过头文件对函数进行声明，再包含头文件，就能引用到头文件中声明的函数(因为函数的实现在库中呢，所以只声明，然后链接就能用了)。

而在C语言中，对其外部函数只能指定为extern类型，因为C语言中不支持extern "C"声明，在.c文件中包含了extern "C"时，当然会出现编译语法错误。

下面是一个具体代码：

/\* c语言头文件：cExample.h \*/

#ifndef C\_EXAMPLE\_H

#define C\_EXAMPLE\_H

extern int add(int x,int y);

#endif

/\* c语言实现文件：cExample.c \*/

#include "cExample.h"

int add( int x, int y )

{

return x + y;

}

// c++实现文件，调用add：cppFile.cpp

extern "C"

{

#include "cExample.h"

}

int main(int argc, char\* argv[])

{

add(2,3);

return 0;

}

可见，如果C++调用一个C语言编写的.DLL时，在包含.DLL的头文件或声明接口函数时，应加extern "C" {　}。这个时候，其实extern "C"是在告诉C++，链接C库的时候，采用C的方式进行链接（即寻找类似\_foo的没有参数信息的标号，而不是默认的\_foo\_int\_int这样包含了参数信息的C++标号了）。

（2）C引用C++函数的具体例子

在C中引用C++语言中的函数和变量时，C++的头文件需添加extern "C"，但是在C语言中不能直接引用声明了extern "C"的该头文件(因为C语言不支持extern "C"关键字，所以会报编译错误)，应该仅在C文件中用extern声明C++中定义的extern "C"函数(就是C++中用extern "C"声明的函数，在C中用extern来声明一下，这样C就能引用C++的函数了，但是C中是不用用extern "C"的)。

下面是一个具体代码：

//C++头文件 cppExample.h

#ifndef CPP\_EXAMPLE\_H

#define CPP\_EXAMPLE\_H

extern "C" int add( int x, int y );

#endif

//C++实现文件 cppExample.cpp

#include "cppExample.h"

int add( int x, int y )

{

return x + y;

}

/\* C实现文件 cFile.c

/\* 这样会编译出错：#include "cExample.h" \*/

extern int add( int x, int y );

int main( int argc, char\* argv[] )

{

add( 2, 3 );

return 0;

}

上面例子，C实现文件cFile.c不能直接用#include "cExample.h"因为C语言不支持extern "C"关键字。这个时候，而在cppExample.h中使用extern "C"修饰的目的是为了让C++编译时候能够生成C形式的符号(类似\_foo不含参数的形式)，然后将其添加到对应的C++实现库中，以便被C程序链接到。

（3）对\_\_BEGIN\_DECLS 和 \_\_END\_DECLS 的理解

在C语言代码中头文件中，经常看到充斥着下面的代码片段：

1. \_\_BEGIN\_DECLS

2. .....

3. .....

4. \_\_END\_DECLS

其实，这些宏一般都是在标准库头文件中定义好了的，例如我当前机器的sys/cdefs.h中大致定义如下：

1. #if defined(\_\_cplusplus)

2. #define \_\_BEGIN\_DECLS extern "C" {

3. #define \_\_END\_DECLS }

4. #else

5. #define \_\_BEGIN\_DECLS

6. #define \_\_END\_DECLS

7. #endif

这目的当然是扩充C语言在编译的时候，按照C++编译器进行统一处理，使得C++代码能够调用C编译生成的中间代码。

由于C语言的头文件可能被不同类型的编译器读取，因此写C语言的头文件必须慎重。

**总结**

**===============**

(1)extern "C"只是C++的关键字，不是C的。所以，如果在C程序中引入了extern "C"会导致编译错误。

(2)被extern "C"修饰的目标一般是对一个C或者C++函数的声明。从源码上看extern "C"一般对头文件中函数声明进行修饰。无论C程序中的还是cpp中的头文件，其函数声明的形式都是一样的（因为两者语法基本一样），对应声明的实现却可能由于相应的程序特性而不同了(C库和C++库里面当然会不同)。

(3)extern "C"这个关键字声明的真实目的，就是实现C++与C及其它语言的混合编程。一旦被extern "C"修饰之后，它便以C的方式工作，可以实现在C中引用C++库的函数，也可以C++中引用C库的函数。

**参考资料：**

http://hi.baidu.com/xiaoxiaolq/blog/item/1edc2af30dd4915a342acc5e.html

http://www.cnblogs.com/xiaoxibo/

以上，是对extern "C"这个关键字的理解和总结，如果具体问题想要讨论或者发现有何遗漏之处，可以与我联系。谢谢！^\_^

作者：QuietHeart

Email:quiet\_heart000@126.com

日期：2011年11月15日