C/C++混合编程的问题

Q1：如果 C++程序要调用已经被编译后的 C 函数，该怎么办？

A1: 假设某个 C 函数的声明如下：

void foo(int x, int y);

该函数被 C 编译器编译后在库中的名字为_foo，而 C++编译器则会产生像_foo_int_int 之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，C++程序不能直接调用 C 函数。C++提供了一个 C 连接交换指定符号 extern“C”来解决这个问题。

例如：

```
extern “C”
{
    void foo(int x, int y);
    … // 其它函数
}
```

或者写成

```
extern “C”
{
    #include “myheader.h”
    … // 其它 C 头文件
}
```

这就告诉 C++编译译器，函数 foo 是个 C 连接，应该到库中找名字_foo 而不是找_foo_int_int。

另外说明一点：C++编译器开发商已经对 C 标准库的头文件作了 extern“C”处理，所以我们可以用＃include 在 C++程序中直接引用这些头文件。

-------C++直接可以引用 C 标准库的原因，反之不可。

Q2 ：如何理解 extern "C"

A2：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。让我们来详细解读这两重含义。

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。记住，下列语句：

extern int a;

仅仅是一个变量的声明，其并不是在定义变量 a，并未为 a 分配内存空间。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。

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。

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；

未加 extern “C”声明时的编译方式

首先看看 C++编译器中对类似 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。函数被 C++编译后在符号库中的名字与 C 语言的不同。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。

未加extern "C"声明时的连接方式

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！

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)，其结果是：

（1）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；

（2）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；反之亦然。

所以，可以用一句话概括 extern “C”这个声明的真实目的（任何语言中的任何语法特性的诞生都不是随意而为的，来源于真实世界的需求驱动。我们在思考问题时，不能只停留在这个语言是怎么做的，还要问一问它为什么要这么做，动机是什么，这样我们可以更深入地理解许多问题）：实现 C++与 C 及其它语言的混合编程。

明白了 C++中 extern "C"的设立动机，我们下面来具体分析 extern "C"通常的使用技巧。

Q3：extern "C"的惯用法？

A3：

（1）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"
}
```

而在 C 语言的头文件中，对其外部函数只能指定为 extern 类型，C 语言中不支持 extern "C"声明，在.c 文件中包含了 extern "C"时会出现编译语法错误。因此，我们经常会做如下处理：

```
#ifdef _cplusplus
    #define GENERAL_EXTERN extern “C”
    #elseif
```

```
        #define GENERAL_EXTERN extern
    #endif
```

使用：

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

说明：使用条件编译，可以有选择题告诉编译器，采用哪种方式编译。如果使用的是C++编译器，那么采用 extern “C”（实现混合编程）；如果使用的是 C 编译器，那么采用 extern（编译不会出错）

笔者编写的 C++引用 C 函数两个例子如下：

实例 1

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;
```

```
}
```

如果 C++调用一个 C 语言编写的.DLL 时，当包括.DLL 的头文件或声明接口函数时，应加 extern "C" { }。

实例 2：

1、C++代码　类声明 substract.h

```
#ifndef _SUBSTRACT_H
#define _SUBSTRACT_H
class SUB
{
public:
    SUB();
    ~SUB();
    int mySubtract(int m, int n, int *sub_result);
protected:
private:
    int *sub_result;
};
#endif
```

2、C++代码　类定义 substract.cpp

```
#include <iostream>
#include "substract.h"
using namespace std;
SUB::SUB()
{
    sub_result = new int;
}
SUB::~SUB()
{
    if ( NULL != sub_result)
```

```
    {
        delete sub_result;
        sub_result = NULL;
    }
}
int SUB::mySubtract(int m, int n, int *sub_result)
{
    *sub_result = m - n;
    return 0;
}
```

3、C 代码　函数声明 add.h

```
#ifndef _ADD_H
#define _ADD_H
extern "C" int myAdd(int a, int b, int *sum);
#endif
```

4、C 代码　函数定义 add.cpp

```
#include"add.h"
int myAdd(int a, int b, int *sum)
{
    *sum = a + b;
    return 0;
}
```

5、c++代码　main 函数

```
#include <iostream>
#include "add.h"
#include "substract.h"
using namespace std;
int main()
{
    int sub = 0;
```

```
        int sum = 0;
        SUB *mySub = new SUB();
        mySub->mySubtract(5,2,&sub);
        //调用 C 函数
        myAdd(5,2,&sum);
        cout << sub << endl;
        cout << sum << endl;
        return 0;
    }
```

总结：C++调用 C 代码关键是需要告诉 C++编译器那些代码是用 C 语言实现的。方法是在 C 的函数声明前加 extern “C”即可。如：extern "C" int myAdd(int a, int b, int *sum);之后就可以被 C++代码任意调用了。

（2）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，应该仅将 C 文件中将 C++中定义的 extern "C"函数声明为 extern 类型。

笔者编写的 C 引用 C++函数一个例子如下：

实例 1：

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;
}
```

如果深入理解了 extern "C"在编译和连接阶段发挥的作用，就能真正理解从 C++引用 C 函数和 C 引用 C++函数的惯用法。这个例子仅仅是从表面上实现了 C 调用 C++的例子，并没有真正涉及到 C++的类。如果涉及到 C++的类，那么就需要采用“C 封装 C++接口”的技术，这样才可以实现 C 调用 C++接口函数。

Q3：如何实现 C 调用 C++接口库？

A3：C 一般不能直接调用 C++函数库，需要将 C++库封装成 C 接口后，才可以使用 C 调用之。所以要了解 C 封装 C++接口策略。

1、C++代码， 类申明 substract.h

```
#ifndef _SUBSTRACT_H
#define _SUBSTRACT_H
class SUB
{
public:
    SUB();
    ~SUB();
    // 类SUB的外部接口
    int SUB_mySubtract(int m, int n, int *sub_result);protected:
private:
    int *sub_result;
};
#endif
```

2、C++代码， 类实现 substract.cpp

```
#include <iostream>
#include "substract.h"
using namespace std;
SUB::SUB()
{
    sub_result = new int;
```

```
}
SUB::~SUB()
{
    if ( NULL != sub_result)
    {
        delete sub_result;
        sub_result = NULL;
    }
}
// 类SUB的外部接口
int SUB::SUB_mySubtract(int m, int n, int *sub_result)
{
    *sub_result = m - n;
    return 0;
}
```

上面的两个文件是用 C++实现减法运算，并对外提供了 SUB 类的接口：SUB_mySubtract（），我们将其生成动态库 libSUB.so（linux 和 solaris 的动态库是.so 文件，window 的动态库是.dll 文件）放到 /usr/lib 目录下，编译命令如下：

sudo g++ -fpic -shared -g -o /usr/lib/libSUB.so add.cpp -I ./

由于在 C 中不能识别类，所以要将上面类的成员函数封装成 C 接口函数才能被调用。下面进行封装，将输出接口转换成 C 接口。

Q3:对由 __cplusplus 和 extern "C" 组成的 C、C++编译器编译标准 C 函数的通用写法的理解

A3:由于 C++编译器需要支持函数的重载，会改变函数的名称，因此 dll 的导出函数通

常是标准 C 定义的。这就使得 C 和 C++的互相调用变得很常见。但是有时可能又会直接用 C 来调用，不想重新写代码，让标准 C 编写的 dll 函数定义在 C 和 C++编译器下都能编译通过，通常会使用以下的格式：（这个格式在很多成熟的代码中很常见）

```
#if defined(__cplusplus)

extern "C" {

#endif

// 在这里写标准 C 程序，例如 dll 导出函数的定义

#ifdef __cplusplus

    }

#endif
```

下面解释一下上面的代码：

首先__cplusplus 是 C++编译器内部定义的宏，如果使用的 C 编译器，__cplusplus 宏不会被定义。它可以作为区分使用的是 C 编译器还是 C++编译器的标志。在标准 C 中 C 代码直接写就可以了。而在 C++中，需要加 extern "C"或包含在 extern "C"块中。由于标准 C 是不支持 extern "C"的，会产生变异错误，所以使用预编译指令通过__cplusplus 来判断只有在使用 C++编译器的时候才定义 extern "C"。

火龙果·整理
uml.org.cn

文章出处：飞诺网(http://www.diybl.com/course/3_program/c++/cppjs/20090517/167150.html)

（最好用 g++编译，我在自己的项目中使用，使用 gcc 碰到过一个莫名其妙的问题）

（上面的奇妙问题，好像是在 C 里面包含了 c++的头文件所引起的，后来不包含这个头文件，而使用 extern 声明头文件中的函数就可以了。）

C 一般不能直接调用 C++函数库，需要将 C++库封装成 C 接口后，才可以使用 C 调用。

下面举例，说明一个封装策略：

```
1. //code in add.cxx
2. #include "add.h"
3. int sample::method()
4. {
5.     cout<<"method is called!\n";
6. }
```

```
1. //code in add.h
2. #include <iostream>
3. using namespace std;
4. class sample
5. {
6.     public:
7.     int method();
8. };
```

将上面的两个文件生成动态库 libadd.so 放到 /usr/lib 目录下，编译命令如下：

```
sudo g++ -fpic -shared -g -o /usr/lib/libadd.so add.cxx -I ./
```

由于在 C 中不能识别类，所以要将上面类的成员函数，要封装成 C 接口函数才能被调用。下面进行封装，将输出接口转换成 C 接口。

```
1. //code in mylib.cxx
2. #include "add.h"
3. #ifndef _cplusplus
```

```
4.  #define _cplusplus
5.  #include "mylib.h"
6.  #endif
7.
8.  int myfunc()
9.  {
10.     sample ss;
11.     ss.method();
12.     return 0;
13. }
```

```
1.  //code in mylib.h
2.  #ifdef _cplusplus
3.  extern "C"
4.  {
5.  #endif
6.
7.  int myfunc();
8.
9.  #ifdef _cplusplus
10. }
11. #endif
```

在 linux 下，gcc 编译器并没用变量_cplusplus 来区分是 C 代码还是 C++代码(没有宏定义)，如果使用 gcc 编译器，这里我们可以自己定义一个变量_cplusplus 用于区分 C 和 C++代码，所以在 mylib.cxx 中定义了一个变量_cplusplus 用于识别是否需要“extern "C"”将函数接口封装成 C 接口。但是如果使用 g++编译器则不需要专门定义_cplusplus，编译命令如下：

```
g++ -fpic -shared -g -o mylib.so mylib.cxx -la -l ./
```

```
1.  main.c
2.  #include <stdio.h>
3.  #include <dlfcn.h>
4.  #include "mylib.h"
5.
6.  int
7.  main()
8.  {
9.      int (*dlfunc)();
10.     void *handle; //定义一个句柄
11.     handle = dlopen("./mylib.so", RTLD_LAZY);//获得库句柄
12.     dlfunc = dlsym(handle, "myfunc"); //获得函数入口
13.     (*dlfunc)();
14.     dlclose(handle);
```

```
15.
16.     return 0;
17. }
```

编译命令如下：

gcc -o main main.c ./mylib.so -ldl

下面就可以执行了。

需要说明的是，由于 main.c 和 mylib.cxx 都需要包含 mylib.h，并且要将函数 myfunc 封装成 C 接口函数输出需要“extern "C"”，而 C 又不识别“extern "C"”，所以需要定义_cplusplus 来区别处理 mylib.h 中的函数 myfunc。

在 main.c 的 main 函数中直接调用 myfunc（）函数也能执行，这里介绍的是常规调用库函数的方法。