关于extern “C”

在你工作过的系统里，不知能否看到类似下面的代码。

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

#ifdef __cplusplus
extern "C" {
#endif

#include <typedef.h>
#include <errcode.h>

typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);

#ifdef __cplusplus
}
#endif

#endif /*   MY HANDLE H   */
```

这好像没有什么问题，你应该还会想：“嗯...是啊，我们的代码都是这样写的，从来没有因此碰到过什么麻烦啊～”。

你说的没错，如果你的头文件从来没有被任何C++程序引用过的话。

这与C++有什么关系呢? 看看__cplusplus 的名字你就应该知道它与C++有很大关系。__cplusplus是一个C++规范规定的预定义宏。你可以信任的是：所有的现代C++编译器都预先定义了它；而所有C语言编译器则不会。另外，按照规范__cplusplus的值应该等于*199711L*，然而不是所有的编译器都照此实现，比如g++编译器就将它的值定义为1。

所以，如果上述代码被C语言程序引用的话，它的内容就等价于下列代码。

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

#include <typedef.h>
#include <errcode.h>

typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);

#endif /*   MY HANDLE H   */
```

在这种情况下，既然extern "C" {}经过预处理之后根本就不存在，那么它和#include指令之间的关系问题自然也就是无中生有。

但一旦最初的代码被一个C++程序引用的话，它就等价于：

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

extern "C" {

#include <typedef.h>
#include <errcode.h>

typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);
}

#endif /* __MY_HANDLE_H__ */
```

在这段代码里，所有的#include指令，类型定义及函数声明都统统的成了extern "C" { }的囊中之物。

但extern "C"是个什么东西？它会产生什么作用？这个我们要从头说起。

extern "C"的前世今生

在C++编译器里，有一位暗黑破坏神，专门从事一份称作**“名字粉碎”**(name mangling)的工作。当把一个C++的源文件投入编译的时候，它就开始工作，把每一个它在源文件里看到的外部可见的名字粉碎的面目全非，然后存储到二进制目标文件的符号表里。

之所以在C++的世界里存在这样一个怪物，是因为C++允许对一个名字给予不同的定义，只要在语义上没有二义性就好。比如，你可以让两个函数是同名的，只要它们的参数列表不同即可，这就是**函数重载**(function overloading)；甚至，你可以让两个函数的原型声明是完全相同的，只要它们所处的**名字空间**(namespace)不一样即可。事实上，当处于不同的名字空间时，所有的名字都是可以重复的，无论是函数名，变量名，还是类型名。

另外，C++程序的构造方式仍然继承了C语言的传统：编译器把每一个通过命令行指定的源代码文件看做一个独立的编译单元，生成目标文件；然后，链接器通过查找这些目标文件的符号表将它们链接在一起生成可执行程序。

编译和链接是两个阶段的事情；事实上，编译器和链接器是两个完全独立的工具。编译器可以通过语义分析知道那些同名的符号之间的差别；而链接器却只能通过目标文件符号表中保存的名字来识别对象。

所以，编译器进行**名字粉碎**的目的是为了让链接器在工作的时候不陷入困惑，将所有名字重新编码，生成全局唯一，不重复的新名字，让链接器能够准确识别每个名字所对应的对象。

但 C语言却是一门单一名字空间的语言，也不允许**函数重载**，也就是说，在一个编译和链接的范围之内，C语言不允许存在同名对象。比如，在一个编译单元内部，不允许存在同名的函数，无论这个函数是否用static修饰；在一个可执行程序对应的所有目标文件里，不允许存在同名对象，无论它代表一个全局变量，还是一个函数。所以，C语言编译器不需要对任何名字进行复杂的处理（或者仅仅对名字进行简单一致的修饰（decoration），比如在名字前面统一的加上单下划线_）。

C++的缔造者Bjarne Stroustrup在最初就把——能够兼容C，能够复用大量已经存在的C库——列为C++语言的重要目标。但两种语言的编译器对待名字的处理方式是不一致的，这就给链接过程带来了麻烦。

例如，现有一个名为my_handle.h的头文件，内容如下：

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

typedef unsigned int result_t;
typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);

#endif /* __MY_HANDLE_H__ */
```

函数的实现放置在一个叫做my_handle.c的C语言代码文件里，内容如下：

```
#include "my_handle.h"

my_handle_t create_handle(const char* name)
{
    return (my_handle_t)0;
}

result_t operate_on_handle(my_handle_t handle)
{
    return 0;
}

void close_handle(my_handle_t handle)
{
}
```

然后使用C语言编译器编译my_handle.c，生成目标文件my_handle.o。由于C语言编译器不对名字进行粉碎，所以在my_handle.o的符号表里，这三个函数的名字和源代码文件中的声明是一致的。

```
0000001a T _close_handle
00000000 T _create_handle
0000000d T _operate_on_handle
```

随后，我们想让一个C++程序调用这些函数，所以，它也包含了头文件my_handle.h。假设这个C++源代码文件的名字叫my_handle_client.cpp，其内容如下：

```
#include "my_handle.h"
#include "my_handle_client.h"

void my_handle_client::do_something(const char* name)
{
    my_handle_t handle = create_handle(name);

    (void) operate_on_handle(handle);

    close_handle(handle);
}
```

然后对这个文件使用C++编译器进行编译，生成目标文件my_handle_client.o。由于C++编译器会对名字进行粉碎，所以生成的目标文件中的符号表会有如下内容：

```
0000002c s EH_frame1
         U __Z12close_handlePv
         U __Z13create_handlePKc
         U __Z17operate_on_handlePv
00000000 T __ZN16my_handle_client12do_somethingEPKc
00000048 S __ZN16my_handle_client12do_somethingEPKc.eh
```

其中，粗体的部分就是那三个函数的名字被粉碎后的样子。

然后，为了让程序可以工作，你必须将my_handle.o和my_handle_client.o放在一起链接。由于在两个目标文件对于同一对象的命名不一样，链接器将报告相关的“符号未定义”错误。

```
Undefined symbols:
  "close_handle(void*)", referenced from:
      my_handle_client::do_something(char const*)in
my_handle_client.o
  "create_handle(char const*)", referenced from:
      my_handle_client::do_something(char const*)in
my_handle_client.o
  "operate_on_handle(void*)", referenced from:
      my_handle_client::do_something(char const*)in
my_handle_client.o
```

为了解决这一问题，C++引入了**链接规范**(linkage specification)的概念，表示法为extern "language string"，C++编译器普遍支持的"language string"有"C"和"C++"，分别对应C语言和C++语言。

链接规范的作用是告诉C++编译：对于所有使用了**链接规范**进行修饰的声明或定义，应该按照指定语言的方式来处理，比如名字，**调用习惯**（calling convention）等等。

链接规范的用法有两种：

1.单个声明的**链接规范**，比如：

```
extern "C" void foo();
```

2. 一组声明的**链接规范**，比如：

```
extern "C"
{
    void foo();
    int bar();
}
```

对我们之前的例子而言，如果我们把头文件my_handle.h的内容改成：

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

extern "C" {

typedef unsigned int result_t;
typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);

}

#endif /* __MY_HANDLE_H__ */
```

然后使用C++编译器重新编译my_handle_client.cpp，所生成目标文件my_handle_client.o中的符号表就变为：

```
00000000 T __ZN16my_handle_client12do_somethingEPKc
00000048 S __ZN16my_handle_client12do_somethingEPKc.eh
         U _close_handle
         U _create_handle
         U _operate_on_handle
```

从中我们可以看出，此时，用extern "C" 修饰了的声明，其生成的符号和C语言编译器生成的符号保持了一致。这样，当你再次把my_handle.o和my_handle_client.o放在一起链接的时候，就不会再有之前的“符号未定义”错误了。

但此时，如果你重新编译my_handle.c，C语言编译器将会报告“语法错误”，因为extern "C"是C++的语法，C语言编译器不认识它。此时，可以按照我们之前已经讨论的，使用宏__cplusplus来识别C和C++编译器。修改后的my_handle.h的代码如下：

```
#ifndef __MY_HANDLE_H__
#define __MY_HANDLE_H__

#ifdef __cplusplus
extern "C" {
#endif

typedef void* my_handle_t;

my_handle_t create_handle(const char* name);
result operate_on_handle(my_handle_t handle);
void close_handle(my_handle_t handle);

#ifdef __cplusplus
}
#endif

#endif /* __MY_HANDLE_H__ */
```

小心门后的未知世界

在我们清楚了 extern "C" 的来历和用途之后，回到我们本来的话题上，为什么不能把 #include 指令放置在 extern "C" { ... } 里面?

我们先来看一个例子，现有a.h，b.h，c.h以及foo.cpp，其中foo.cpp包含c.h，c.h包含b.h，b.h包含a.h，如下：

```
#ifndef __A_H__
#define __A_H__

#ifdef __cplusplus
extern "C" {
#endif

void a();

#ifdef __cplusplus
}
#endif

#endif
```

```
#ifndef __B_H__
#define __B_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "a.h"

void b();

#ifdef __cplusplus
}
#endif

#endif
```

```
#ifndef __C_H__
#define __C_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "b.h"

void c();

#ifdef __cplusplus
}
#endif

#endif
```

foo.cpp的内容如下：

```
#include "c.h"
```

现使用C++编译器的预处理选项来编译foo.cpp，得到下面的结果：

```
extern "C" {
  extern "C" {
    extern "C" {
      void a();
    }
    void b();
  }
  void c();
}
```

正如你看到的，当你把#include指令放置在extern "C" { }里的时候，则会造成extern "C" { } 的嵌套。这种嵌套是被C++规范允许的。当嵌套发生时，以最内层的嵌套为准。比如在下面代码中，函数foo会使用C++的链接规范，而函数bar则会使用C的链接规范。

```
extern "C" {
  extern "C++" {
    void foo();
  }
  void bar();
}
```

如果能够保证一个C语言头文件直接或间接依赖的所有头文件也都是C语言的，那么按照C++语言规范，这种嵌套应该不会有什么问题。但具体到某些编译器的实现，比如MSVC2005，却可能由于 extern "C" { } 的嵌套过深而报告错误。不要因此而责备微软，因为就这个问题而言，这种嵌套是毫无意义的。你完全可以通过把#include指令放置在extern "C" { }的外面来避免嵌套。拿之前的例子来说，如果我们把各个头文件的 #include 指令都移到extern "C" { } 之外，然后使用C++编译器的预处理选项来编译foo.cpp，就会得到下面的结果：

```
extern "C" {
   void a();
}

extern "C" {
   void b();
}

extern "C" {
   void c();
}
```

这样的结果肯定不会引起编译问题的结果——即便是使用MSVC。

把 #include 指令放置在extern "C" { }里面的另外一个重大风险是，你可能会无意中改变一个函数声明的链接规范。比如：有两个头文件a.h，b.h，其中b.h包含a.h，如下：

```
#ifndef __A_H__
#define __A_H__

#ifdef __cplusplus

void foo(int);

#define a(value) foo(value)

#else

void a(int);

#endif

#endif /* __A_H__ */
```

```
#ifndef __B_H__
#define __B_H__

#ifdef __cplusplus
extern "C" {
#endif

#include "a.h"

void b();

#ifdef __cplusplus
}
#endif

#endif /* __B_H__ */
```

此时，如果用C++预处理器展开b.h，将会得到：

```
extern "C" {

void foo(int);

void b();

}
```

按照a.h作者的本意，函数foo是一个C++自由函数，其**链接规范**为"C++"。但在b.h中，由于#include "a.h"被放到了extern "C" { }的内部，函数foo的**链接规范**被不正确地更改了。

由于每一条 #include 指令后面都隐藏这一个未知的世界，除非你刻意去探索，否则你永远都不知道，当你把一条条#include指令放置于extern "C" { }里面的时候，到底会产生怎样的结果，会带来何种的风险。或许你会说，“我可以去查看这些被包含的头文件，我可以保证它们不会带来麻烦”。但，何必呢？毕竟，我们完全可以不必为不必要的事情买单，不是吗？

Q&A

Q: 难道任何#include指令都不能放在extern "C"里面吗？

A: 正像这个世界的大多数规则一样，总会存在特殊情况。

有时候，你可能利用头文件机制“巧妙”的解决一些问题。比如，#pragma pack的问题。这些头文件和常规的头文件作用是不一样的，它们里面不会放置C的函数声明或者变量定义，链接规范不会对它们的内容产生影响。这种情况下，你可以不必遵守这些规则。

更加一般的原则是，在你明白了这所有的原理之后，只要你明白自己在干什么，那就去做吧。

Q: 你只说了不应该放入extern "C"的，但什么可以放入呢？

A: 链接规范仅仅用于修饰函数和变量，以及函数类型。所以，严格的讲，你只应该把这三种对象放置于extern "C"的内部。

但，你把C语言的其它元素，比如非函数类型定义（结构体，枚举等）放入extern "C"内部，也不会带来任何影响。更不用说宏定义预处理指令了。

所以，如果你更加看重良好组织和管理的习惯，你应该只在必须使用extern "C"声明的地方使用它。即使你比较懒惰，绝大多数情况下，把一个头件自身的所有定义和声明都放置在extern "C"里面也不会有太大的问题。

Q: 如果一个带有函数/变量声明的C头文件里没有extern "C"声明怎么办?

A: 如果你可以判断，这个头文件永远不可能让C++代码来使用，那么就不要管它。

但现实是，大多数情况下，你无法准确的推测未来。你在现在就加上这个extern "C"，这花不了你多少成本，但如果你现在没有加，等到将来这个头文件无意中被别人的C++程序包含的时候，别人很可能需要更高的成本来定位错误和修复问题。

Q: 如果我的C++程序想包含一个C头文件a.h，它的内容包含了C的函数/变量声明，但它们却没有使用extern "C"链接规范，该怎么办?

A: 在a.h里面加上它。

某些人可能会建议你，如果a.h没有extern "C"，而b.cpp包含了a.h，可以在b.cpp里加上：

```
extern "C"
{
#include "a.h"
}
```

这是一个邪恶的方案，原因在之前我们已经阐述。但值得探讨的是，这种方案这背后却可能隐含着一个假设，即我们不能修改a.h。不能修改的原因可能来自两个方面：

1. 头文件代码属于其它团队或者第三方公司，你没有修改代码的权限；
2. 虽然你拥有修改代码的权限，但由于这个头文件属于遗留系统，冒然修改可能会带来不可预知的问题。

对 于第一种情况，不要试图自己进行workaround，因为这会给你带来不必要的麻烦。正确的解决方案是，把它当作一个bug，发送缺陷报告给相应的团队 或第三方公司。如果是自己公司的团队或你已经付费的第三方公司，他们有义务为你进行这样的修改。如果他们不明白这件事情的重要性，告诉他们。如果这些头文 件属于一个免费开源软件，自己进行正确的修改，并发布patch给其开发团队。

在 第二种情况下，你需要抛弃掉这种不必要的安全意识。因为，首先，对于大多数头文件而言，这种修改都不是一种复杂的，高风险的修改，一切都在可控的范围之 内；其次，如果某个头文件混乱而复杂，虽然对于遗留系统的哲学应该是：“在它还没有带来麻烦之前不要动它”，但现在麻烦已经来了，逃避不如正视，所以上策 是，将其视作一个可以整理到干净合理状态的良好机会。

Q: 我们代码中关于extern "C"的写法如下，这正确吗?

```
#ifdef __cplusplus
   #if __cplusplus
   extern "C" {
   #endif
#endif

void foo();

#ifdef __cplusplus
   #if __cplusplus
   }
   #endif
#endif
```

A: 不确定。

按照C++的规范定义，__cplusplus 的值应该被定义为199711L，这是一个非零的值；尽管某些编译器并没有按照规范来实现，但仍然能够保证__cplusplus的值为非零——至少我到目前为止还没有看到哪款编译器将其实现为0。这种情况下，#if __cplusplus ... #endif完全是冗余的。

但，C++编译器的厂商是如此之多，没有人可以保证某款编译器，或某款编译器的早期版本没有将__cplusplus的值定义为0。但即便如此，只要能够保证宏__cplusplus只在C++编译器中被预先定义，那么，仅仅使用#ifdef __cplusplus ... #endif就足以确保意图的正确性；额外的使用#if __cplusplus ... #endif反而是错误的。

只有在这种情况下：即某个厂商的C语言和C++语言编译器都预先定义了__cplusplus，但通过其值为0和非零来进行区分，使用#if __cplusplus ... #endif才是正确且必要的。

既然现实世界是如此复杂，你就需要明确自己的目标，然后根据目标定义相应的策略。比如：如果你的目标是让你的代码能够使用几款主流的、正确遵守了规范的编译器进行编译，那么你只需要简单的使用#ifdef __cplusplus ... #endif就足够了。

但如果你的产品是一个雄心勃勃的，试图兼容各种编译器的（包括未知的）跨平台产品， 我们可能不得不使用下述方法来应对各种情况，其中__ALIEN_C_LINKAGE__是为了标识那些在C和C++编译中都定义了__cplusplus宏的编译器。

```
#ifdef __cplusplus
  #if !defined((__ALIEN_C_LINKAGE__) || \
      (defined(__ALIEN_C_LINKAGE__) && __cplusplus)
    extern "C" {
  #endif
#endif

// Here is your declarations.

#ifdef __cplusplus
  #if !defined((__ALIEN_C_LINKAGE__) || \
      (defined(__ALIEN_C_LINKAGE__) && __cplusplus)
    }
  #endif
#endif
```

这应该可以工作，但在每个头文件中都写这么一大串，不仅有碍观瞻，还会造成一旦策略进行修改，就会到处修改的状况。违反了DRY(Don't Repeat Yourself)原则，你总要为之付出额外

的代价。 解决它的一个简单方案是，定义一个特定的头文件——比如clinkage.h，在其中增加这样的定义：

```
#if defined(__cplusplus) && \
    (!defined(__AN_ALIEN_IMPL__ || \
     (defined(__AN_ALIEN_IMPL__) && __cplusplus)

   #define __BEGIN_C_DECLS extern "C" {
   #define __END_C_DECLS }

#else

   #define __BEGIN_C_DECLS
   #define __END_C_DECLS

#endif
```

然后让你系统中的所有其它头文件头都使用这两个宏，例如：

```
#include "clinkage.h"

__BEGIN_C_DECLS

void foo();

__END_C_DECLS
```