# C In Practice

## 如何在工程中应用C语言

袁英杰　ThoughtWorks

# 目录

## 条款1. 不要把#include指令放在extern “C”里

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，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。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。

拿之前的例子来看，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？

## 应对遗留系统

如果一个C++程序想包含一个已有的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？

某些人可能会建议，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原因：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。

## 应对可移植性

在某些产品代码中，你可以看到如下的写法：

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

void foo();

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

我不知道这样编写的原因，因为一般来讲简单的使用#ifdef __cplusplus就够了。之所以又加上另外一个条件，或许是为了应对某些编译器的实现，以提高可一致性。但不幸的是，这样的处理方案仍然不能应对所有可能的实现。

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
```

## # include之外的元素

是的，#include不应该被放入extern "C"声明，但什么可以放入呢?

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。

## 尽量使用extern “C”

如果你可以判断，你的头文件永远不可能让C++代码来使用，你可以选择不在你的头文件中使用extern “C”。

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。

那么源代码又如何呢?

extern “C”是一个 C++语言元素。和头文件不一样，一般情况下，你不会选择使用C++编译器来编译C语言源代码。这种情况下，在源代码文件中使用它是没有什么意义的。

但，总会存在一些异常。比如，有时候，有些单元测试方案使用基于C++语言的xUnit框架，并且，为了测试的简便性，会通过把被测的源代码文件通过 # include指令包含到测试用例文件中，这时，你就必须把那些源代码中的函数定义用extern “C”包含起来。

为了防止这些意外，最好也在源代码中使用extern“C”，毕竟，这并不会增加太多的成本。

## Rules Made For Breaking

难道任何#include指令都不能放在extern "C"里面吗?

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（参见条款4）。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。

# 条款2. 谨慎处理数据对齐问题

对齐问题来源于由于硬件设计而导致的时间与空间的矛盾，发生于处理跨平台，网络通信，硬件依赖的场合。

绝大多数硬件平台都存在数据对齐问题。如果数据在内存中没有按照对齐的方式放置，重则导致硬件异常的发生，轻则导致性能问题。比如，在Intel 32位x86平台上，32位的数据总线每次存取内存总是以4的倍数的边界，并在以4为倍数的范围内进行。如果一个32位整数被放置在以4的倍数为起始地址（比如0，4，8等）的内存里，那么，存取它只需要一次总线操作；但如果它被放置在跨边界的位置上，则需要两次总线操作。

一般情况下，编译器的默认对齐选项总是以性能为目标。在空间代价最小的情况下，自动生成的数据对齐方式在目标平台上是性能最优的。

对齐问题分为几个具体情况：

- 基本数据类型（scalar data type）的对齐；
- 结构体（struct），共用体（union）的内部成员的对齐；
- 结构体，共用体的对齐；
- 数组（array）的对齐

## 基本数据类型的对齐

对于基本数据类型的对齐，通用的原则是：**自然对齐**（natural alignment）是移植性最好、性能最优的方式。所谓**自然对齐**指的是：当一个基本类型数据的大小为2的n次方时，它被放置的起始地址等于“自身大小的整数倍”；如果一个基本类型数据的大小不等于2的n次方，其自然对齐的值为“大于自身大小的最小的2的n次方”。

下表列出了在Intel x86平台上，各种C语言基本数据类型的大小及自然对齐的值。

| C基本数据类型 | IA-32<br>数据大小/自然对齐 | IA-64<br>数据大小/自然对齐 |
|---|---|---|
| signed char, unsigned char | 1/1 | 1/1 |
| signed short, unsigned short | 2/2 | 2/2 |
| signed int, unsigned int | 4/4 | 4/4 |
| signed long, unsigned long | 4/4 | 8/8 |
| singed long long, unsigned long long | 8/8 | 8/8 |
| float | 4/4 | 4/4 |
| double | 8/8 | 8/8 |
| long double | 12/16 | 12/16 |
| enum | 4/4 | 4/4 |
| * (pointer) | 4/4 | 8/8 |

但某些处理器架构的实现却未必要求基本数据类型必须以自然对齐的方式放置才能获得最优的性能。比如，对于IA-32平台，在性能最优的前提下，各种数据类型最小的对齐要求如下表所示：

| C基本数据类型 | 数据大小 | 对齐要求 |
|---|---|---|
| signed char, unsigned char | 1 | 1 |
| signed short, unsigned short | 2 | 2 |
| signed int, unsigned int | 4 | 4 |

| | | |
|---|---|---|
| signed long, unsigned long | 4 | 4 |
| singed long long, unsigned long long | 8 | 4 |
| float | 4 | 4 |
| double | 8 | 4 |
| long double | 12 | 4 |
| enum | 4 | 4 |
| * (pointer) | 4 | 4 |

问题的复杂之处在于：在不同的硬件体系架构上运行的不同操作系统平台上，使用不同的编译器编译所得到的对齐方式是不一样的。比如，在基于IA-32平台的Windows，Mac OS和Linux上，MSVC和gcc编译器对于部分基本数据的对齐处理是不一样的，如下表：

| 基本数据类型 | Windows MSVC2008 数据大小/对齐 | Windows gcc4 数据大小/对齐 | Mac OS gcc4 数据大小/对齐 | Linux gcc4 数据大小/对齐 |
|---|---|---|---|---|
| singed long long, unsigned long long | 8/8 | 8/8 | 8/4 | 8/4 |
| double | 8/8 | 8/8 | 8/4 | 8/4 |
| long double | 8/8 | 12/4 | 16/16 | 12/4 |

## 结构体的内部成员的对齐

在进一步讨论之前，我们先来定义一个概念：**偏移对齐**。它指的是：一个结构体成员相对于结构体起始地址的偏移量所应该进行的对齐。

当一个结构体内部成员都属于**基本数据类型**时，它们的**偏移对齐**方式和**基本数据类型**在内存中的对齐是一致的。

由于一个结构体保存于一个连续的内存区域，当它所有的数据成员都按照自己的方式进行对齐后，就可能出现数据成员之间的内存间隙，这就引出了**数据填充**（data padding）问题。

**数据填充**存在于两个地方：

- 结构体的成员之间；
- 结构体的末尾

结构体成员之间的**数据填充**方式是：先将第一个数据成员放置在偏移为0的位置，然后将第二个数据成员放置在第一个“大于或等于自己当前偏移量”的对齐位置；放置好第二个数据成员之后，第三个数据的偏移量可能已经发生了变化，此时，按照其新的偏移量，将其放置在第一个“大于或等于自己新的偏移量”的对齐位置。然后依次按照同样的规则放置第四个，第五个……数据成员，直到所有的数据成员都放置完成。这时，如果数据成员之间存在间隙，则进行填充。

比如，一个结构体定义如下：

```
struct A
{
    char    a;
    short   b;
    double  c;
};
```

在 IA-32 Windows平台上使用MSVC2008默认对齐方式编译后的二进制模型等价于：

```
struct A
{
    char    a;              // 按照1对齐
    char    padding_1[1];
    short   b;              // 按照2对齐
    char    padding_2[4];
    double  c;              // 按照8对齐
};
```

结构体尾部填充原则为：一个结构体经过对齐处理后，其大小应该等于其“成员最大对齐值”的整数倍；否则，将在尾部进行最少的填充来满足这一点。所谓“成员最大对齐值”，指的是一个结构体每个成员的“偏移对齐值”的最大值。

比如，一个结构体的定义为：

```
struct B
{
    short a;
    int   b;
    char  c;
};
```

其填充后的结果等价于：

```
struct B
{
    short a;                // 按照2对齐
    char  padding_1[2];
    int   b;                // 按照4对齐
    char  c;                // 按照1对齐
    char  padding_2[3];
};
```

在这个例子中，将成员a，b，c对齐后，结构体的大小为9，而其“成员最大对齐值”为b的对齐值4，而9不是4的倍数，所以，需要在尾部填充3个字节以让结构体的大小等于4的倍数。

此时，你可能会觉得尾部填充是多余的，其规则制定的尤其显得莫名其妙。我们在后面会谈到背后的原因。

## 结构体的对齐

一个结构体在内存中的起始位置，应该是其“成员最大对齐值”的整数倍。

以之前例子中的结构体定义为例，一个类型为struct A的变量应该放置在以8对齐的位置；而一个类型为struct B的变量应该放置在以4对齐的位置。

这是因为，一个结构体经过对齐处理后，假设其“成员最大对齐值”为n，而“成员最大对齐值”所对应的成员名字为m；那么可以肯定的是，从结构体的起始位置，到m之间的空间大小也一定是n的整数倍。所以，只有把一个结构体变量的起始位置设置为n的整数倍，才能保证m在内存中是以n对齐的。

在之前谈到结构体内部成员的对齐时，我们仅仅讨论了由基本数据类型组成的结构体的对齐方式。现在，我们可以加上：如果一个结构的成员包含另外的结构体，那么被包含结构体的“偏移对齐”方式和其在内存中的对齐方式是一样的，即它的偏移量需要按照它所包含数据成员的最大对齐值来对齐。

现有一个结构体C的定义如下：

```
struct C
{
    short      a;
    struct A   b;
};
```

由于在之前的例子中，结构体A的成员最大对齐值为8，所以结构体C经过对齐处理后得到的二进制结果应该等价于：

```
struct C
{
    short      a;            // 按照2对齐
    char       padding[6];
    struct A   b;            // 按照8对齐
};
```

## 数组的对齐

一个数组在内存中的起始位置会按照其元素类型的对齐方式来安置。比如，一个int类型的数组，其起始位置应该为4的倍数；一个double类型的数组，其起始位置应该为8的倍数；而一个结构体A类型的数组，由于结构体A的对齐值为8，所以数组的起始位置而应该为8的倍数。

由于一个数组在内存中的空间应该是连续的，当一个结构的对齐是n时，如果结构体的大小不是n的倍数，则会造成这个结构体数组在空间上的不连续。这就是为什么在必要的时候要对一个结构体进行尾部填充的原因。

当一个结构体的某个成员是数组类型时，它的偏移对齐方式和内存对齐方式是一样的。当对齐之后，如果这个数组成员和随后的成员之间有间隙，也需要进行填充。当计算一个结构体的最大对齐值的时候，数组成员的对齐值等于数组元素类型的对齐值。

## 改变的不仅仅是形式

为了增强数据的可管理性，我们往往会定义一些嵌套的结构体。

比如，我们最初有一个结构体，定义如下：

```
struct Channels
{
   unsigned char  numberOfupLinkChannels;
   unsigned char  numberOfDownLinkChannels;
   unsigned short upLinkSpeed;
   unsigned short downLinkSpeed;
};
```

从中我们能够看到清晰的对称结构。为了提高其可管理性，我们将此结构体重新定义为：

```
struct UnidirectionChannels
{
   unsigned char  numberOfChannels;
   unsigned short speed;
};

struct Channels
{
   UnidirectionChannels upLink;
   UnidirectionChannels downLink;
};
```

从语义的角度而言，这和原来没有任何不同。但在默认的对齐方式下，二者的大小却不相等（前者占用6个字节，后者占用8个字节）。

对于这类问题，如果空间的布局和内存的消耗并不是一个关注点，仍然应该鼓励使用可管理性更好的结构体定义方法。否则，则需要仔细的考虑由对齐填充所引起的问题。

## Payload问题

另外一个常见的“错误”是动态payload分配相关的。假设，一个系统是基于消息进行进程间通信的。所有的消息都有统一的消息头定义，但每个消息体的大小则各不相同。为了能够提供统一的消息分配接口，我们常用的技巧是，定义一个如下的结构：

```
typedef struct
{
    unsigned int receiverId;
    unsigned int senderId;
    unsigned int msgLen;
    char         payload[1];
} Message;
```

在这个结构体定义中，前四个成员是消息头，而从payload开始，则是用户数据区。由于标准C语言不允许定义大小为0的数组（gcc的扩展是允许的），所以，我们将payload定义为大小为1的数组，而实际的大小，则根据指定的大小来分配。

我们常常可以看到这样的实现：

```
void* alloc_msg(size_t sizeOfPayload)
{
    return malloc(sizeof(Message) - 1 + sizeOfPayload);
}
```

这个实现包含着一个假设，即sizeof(Message) 的大小减去payload的大小之后等于消息头的大小。但其错误之处在于，结构体Message后面的padding被忽略了，所以每次传给malloc的参数总是比实际需要的要大。这虽然并不会导致任何致命的问题，却可能会造成潜在的内存浪费。

一个正确的实现如下：

```
void* alloc_msg(size_t sizeOfPayload)
{
    return malloc( offsetof(Message, payload) \
                + sizeOfPayload );
}
```

## payload问题的通用性

我们用char[1]的方式来定义payload并不具备通用性。

我们再来仔细看一下这个结构体：

```
typedef struct
{
    unsigned int receiverId;
    unsigned int senderId;
    unsigned int msgLen;
    char         payload[1];
} Message;
```

这中定义里面隐藏着另外一个与对齐相关的风险：如果payload数据的第一个成员是一个需要按照8字节或16字节进行对齐的类型，比如double，将会发生什么？

按照这个定义，你只能保证payload按照4个字节对齐。这样的话，那个数据将不能得到自然的对齐，从而在数据操作的时候，将存在发生问题的可能。

一个解决方案是，定义一个包含各种基本数据类型的共用体，让其来保证数据对齐。如下：

```
typedef union
{
   char        c;
   short       s;
   int         i;
   float       f;
   long        l;
   double      d;
   long long   ll;
   long double ld;
} alignment_util_t;
```

然后，用它在你消息结构体中来定义payload的起始地址。如下：

```
typedef struct
{
    unsigned int        receiverId;
    unsigned int        senderId;
    unsigned int        msgLen;
    alignment_util_t    payload;
} Message;
```

这样你就可以保证，你的payload数据对齐不会存在任何问题。

尽管这是适用性最好的方案。但对于这个例子，在某些平台上，消息头和消息体之间将会产生4个字节的padding。

尽管这在大多数情况并不是一个问题。但在某些系统里，根本就不允许在消息里使用double或long long等数据类型，这种防备根本没有必要。对此，你只需要将共用体alignment_util_t中那些肯定不会使用到的类型都去掉，仅仅保留那些可能会用的基本类型。

## 重新安排结构体成员

由于对齐的原因，一个结构体可能会存在一定的填充字节，它们对于你要处理的业务是毫无用处的，这当然是一种浪费。有时候这种浪费相当的巨大，可能会超过30%甚至40%。如果一个数据结构被大规模的分配，这将是一种难以承受的开销。

所幸，如果你可以按照如下方式定义你的结构体，你就可以 在不影响性能的前提下，将这种浪费降低到最低：**在一个结构体内，按照成员数据类型的对齐值，从大到小进行排列。**

但是，并非所有时候你都可以自由调整结构体成员的顺序。比如，当一个结构体是用来表现一个协议包或者磁盘节点的时候。

## 修改结构体成员的默认对齐方式

当结构体成员的对齐方式被指定为n时，其含意为：结构体成员的最大对齐必须小于等于n。这就意味着，一个默认对齐大于n的成员，也只能按照n来对齐，但默认对齐小于等于n的成员，其对齐方式则不受影响。

比如，对于之前例子中的结构体A:

```
struct A
{
   char   a;
   short  b;
   double c;
};
```

在默认对齐方式下，其大小为16。如果将其对齐值被修改为4，成员a，b的对齐不会受到影响，但成员c的对齐则由8变为4，结构体的大小也相应的变成12。

如果你想修改某个编译单元，甚至整个程序里结构体的默认对齐方式，你可以通过编译器的选项来做到这一点。比如gcc的相应选项为-fpack-struct[=n]，而cl编译器的选项为/Zp[n]，两者的n的取值范围均为1，2，4，8和16。

如果你不想进行全局的修改，或者想让某些结构体的对齐方式与总体设置不同，你可以使用预处理指令#pragma pack。比如：

```
#pragma pack(1)

struct A
{
    char     a;
    short    b;
    double   c;
};

#pragma pack()
```

其中#pragma pack(1)表示：自此以下的结构体的成员都以1来进行对齐，而#pragma pack()则表示结构体成员的对齐方式重新恢复到全局指定的方式。在这个例子中，结构体A将会不有任何填充，其在IA-32架构上的大小为11。

需要注意的是，这两种方式可能会带来较大的性能损失。在某些平台上，甚至可能会造成操作系统异常，或者程序崩溃。

## 只能做减法

另外需要注意的是，#pragma pack只能**保持**或**减少**全局指定的对齐值。比如，你通过/Zp4选项将全局的结构成员对齐值指定为4，然后你在代码中使用#pragma pack(8)来指定某个结构体的成员对齐方式为8。这种情况下，这个局部指定将会被忽略，指定结构体的成员对齐方式仍然是全局指定的4。然而，如果你使用#pragma pack(2)来指定某个结构的成员对齐方式为2，则2将会被用作指定结构体的成员对齐值。

## 不兼容的#pragma pack

C99除了规定了几个形式为#pragma STDC directive的标准指令外，还允许各个编译器厂商使用#pragma来进行自定义的语言处理。#pragma pack就是一种厂商自定义的指令，尚未标准化。尽管有人宣称#pragma pack已经是用来指定结构体成员对齐方式的事实标准， 但很不幸，并不是所有的编译器都支持这种方式。更糟糕的是，各种支持 # pragma pack的编译器支持的具体形式也不尽相同。

比如，许多编译器，比如gcc，msvc和SUN cc, IBM XL C/C++编译器用下列指令来表示“在遇到下一个#pragma pack指令之前，随后的所有结构体的最大偏移对齐为n”：

- #pragma pack(n)

但另外一些编译器，比如HP C/C++编译器则使用来达到相同的目的：

- #pragma PACK n
- #pragma pack n

甚至，有些编译器使用非常特别的指令来指定结构体对齐，比如：

- #pragma ZTC align n

- #pragma align = n

而对于如何恢复到全局设定的对齐方式，其指令也非常的丰富多彩：

- #pragma pack()
- #pragma pack(0)
- #pragma pack
- #pragma ZTC align
- #pragma unpack

等等等等。

许多编译器还支持以压栈的方式来操作结构体对齐，比如，有的用：

1. #pragma pack(push)
2. #pragma pack(n)

来进行压栈，并指定新的对齐。有的则将两种操作合二为一：

- #pragma pack(push, n)

需要出栈的时候，则用：

- #pragma pack(pop)

将对齐方式恢复到上一条压栈指令之前的状态。有些编译器仅仅支持使用压栈的方式，而有些则既支持压栈方式，也支持指定方式。

总之，编译器对此支持的方式千差万别。如果你打算使用多种编译器编译你的代码， 就必须使用复杂的宏控制来处理对齐相关的操作，比如：

```
#if defined(_MSC_VER) || defined(_QC) || defined(__WATCOM__)
    #pragma pack(1)
#elif defined(__ZTC__)
    #pragma ZTC align 1
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
    #pragma option -a-
#endif

struct foo
{
    char  code;
    float sum;
} ;

#if defined (_MSC_VER) || defined(_QC) ||
defined(__WATCOMC__)
 #pragma pack()
#elif defined (__ZTC__)
 #pragma ZTC align
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
 #pragma option -a.
#endif
```

这样的方式啰嗦麻烦，让人生厌。更加让人不能忍受的是，在每一个需要指定结构体对齐的地方，你都需要拷贝这么两大段。这种明显违背DRY原则的重复，有一天可能会带来致命的伤害，可以想象，如果有一天，你突然需要支持另外一种编译器，会有多少地方等着你去修改。

可能在大多数情况下，你并不需要支持这么多的编译器。但只要你要支持两种以上的编译器，而它们之间对于对齐的处理指令不一致，你就依然面临这样的问题。另外，即使当前你只需要支持一种编译器，或者你足够走运，尽管你需要支持两种以上的编译器，但它们对于对齐的支持方式是一样的，但你却永远无法回避当某天一种不一致的编译器需要支持的时候，你所面临的到处修改。

列宁的老问题——怎么办？！

## 无能为力的宏

面临这样的问题，作为富有经验的C语言工程师，你肯定会首先想到，为什么不发挥C语言强大的宏的能力？

比如这样：

```
#if defined(_MSC_VER) || defined(_QC) || defined(__WATCOM__)
    #define MY_PACK_1 pack(1)
    #define MY_UNPACK pack()
#elif defined(__ZTC__)
    #define MYPACK_1  ZTC align 1
    #define MY_UNPACK ZTC align
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
    #define MYPACK_1 option -a-
    #define MY_UNPACK option -a
#endif
```

这样，你就可以在你的代码中这样使用：

```
#pragma MY_PACK_1

struct foo
{
    char  code;
    float sum;
} ;

#pragma MY_UNPACK
```

很巧妙，是吧？但……很不幸，这行不通。

C99规定，形如#pragma STDC directive的指令，是不允许 C预处理器进行宏替换的。而对于其它形式的#pragma指令，是否进行宏替换，C99并不禁止，而是由各个厂商自己决定。

虽然#pragma pack不属于规定形式的#pragma指令，但绝大多数编译器厂商仍然遵从了相关的约束，即，不对＃pragma指令中的词汇进行宏替换。所以，对于这些编译器而言，你无法使用上述方案。

即使个别编译器确实支持宏替换，但由于大多数别的编译器不支持，这种内在的不一致，使你不可能使用上述方案。

## 令人爱恨交加的#include

上帝在关闭一扇门的时候，往往会再打开一扇窗。 是的，你无法使用宏替换。但却可能利用其它其它预处理指令，比如……#include。

什么？#include？C语言的的通过头文件输出接口的模型一直让人诟病，且不说很多程序员都会错误混乱的使用它，就算你在正确的用它，它也会给编译带来很多额外的开销。

但，针对我们这个问题，它却会展示它可爱的一面——

我们先定义一个名为pack_1的文件，内容如下：

```
#if defined(__SUPPORT_PACK_PUSH__)
   #pragma pack(push)
   #pragma pack(1)
#elif defined(__SUPPORT_COMPACT_PACK_PUSH_)
   #pragma pack(push, 1)
#elif defined(__SUPPORT_ENCLOSED_PACK_N__)
   #pragma pack(1)
#elif defined(__SUPPORT_NON_ENCLOSED_PACK_N__)
   #pragma pack 1
#elif defined(__ZTC__)
   #pragma ZTC align 1
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
   #pragma option -a-
#else
   #error "unknown pack directive"
#endif
```

再定义一个名为unpack的文件：

```
#if defined(__SUPPORT_PACK_POP__)
   #pragma pack(pop)
#elif defined(__SUPPORT_ENCLOSED_EMPTY_PACK__)
   #pragma pack()
#elif defined(__SUPPORT_NON_ENCLOSED_EMPTY_PACK__)
   #pragma pack
#elif defined(__SUPPORT_ENCLOSED_PACK_0__)
   #pragma pack(0)
#elif defined(__ZTC__)
   #pragma ZTC align
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
   #pragma option -a
#else
   #error "unknown pack directive"
#endif
```

然后，在需要按1进行对齐处理的结构体前后使用它们，例如：

```
#include <pack_1>

struct Foo
{
   char   type;
   short  length;
};

#include <unpack>
```

众所周知，#Include指令会打开指定的文件，并将其内容展开到当前#include指令所在的位置。这样，那些对齐相关的大块代码就被压缩为只带有抽象含意的一行指令。

这样的解决方案，除了让你的代码简洁之外，最重要的是，当你需要新增一种编译器的对齐方式时，你只需要在一个地方修改。够DRY，不是吗？

## 将DRY进行到底

是的，我撒谎了。除了pack_1之外，你还需要定义名为pack_2，pack_4，pack_8，pack_16的文件。它们的内容和pack1非常相似。这就意味着，当你需要增加一种对齐方式的支持时，你需要在这5个文件中——甚至6个文件中，如果unpack也需要修改的话——进行一致的修改。这还不够DRY。

正如前面已经讨论过的，你不能使用宏来定义pack_n文件中的内容。但，你可以编写一个代码生成器，用以自动生成pack_1，pack_2等文件的内容。这并不是一件多么困难的工作。

## 非主流的头文件

上述的几个pack相关的文件，虽然都是通过#include指令包含的，但它们不是头文件，至少不是常规意义上的头文件。它们需要被重复包含，每一次包含都必须保证能够将内容展开在包含的位置。

这就决定了，你不能像定义一个常规头文件那样，在文件的头尾加上防卫宏的控制。也就是说，一定不要这样定义这些文件：

```
#ifndef __PACK_1_H__  // 错误：一定不要定义防卫宏
#define __PACK_1_H__  // 错误：一定不要定义防卫宏

#if defined(__SUPPORT_PACK_PUSH__)
   #pragma pack(push)
   #pragma pack(1)
#elif defined(__SUPPORT_COMPACT_PACK_PUSH_)
   #pragma pack(push, 1)
#elif defined(__SUPPORT_ENCLOSED_PACK_N__)
   #pragma pack(1)
#elif defined(__SUPPORT_NON_ENCLOSED_PACK_N__)
   #pragma pack 1
#elif defined(__ZTC__)
   #pragma ZTC align 1
#elif defined(__TURBOC__) && (__TURBOC__ > 0x202)
   #pragma option -a-
#else
   #error "unknown pack directive"
#endif

#endif /* __PACK_1_H__ */ // 错误：一定不要定义防卫宏
```

## “!DRY”

有些人喜欢对多个结构体定义统一用一个pack指令来约束。比如：

```
#include <pack_1>

struct Foo
{
    int    a;
    double b;
    long   c;
};

struct Bar
{
    int    a;
    double b;
    long   c;
};

struct Another
{
    int    a;
    double b;
    long   c;
};

#include <unpack>
```

毫无疑问，这样可以节省一些重复的工作。但这样的节省却可能导致一些潜在的风险。

比如，随后，由于某些原因，你想让Bar按照2个字节进行pack，由于pack和unpack都放置在很远的地方，你没有注意到它们，所以你仅仅在Bar的前后加上了pack 2相关的指令。如下：

```
#include <pack_1>

struct Foo
{
    int    a;
    double b;
    long   c;
};

#include <pack_2>
struct Bar
{
    int    a;
    double b;
    long   c;
};
#include <unpack>

struct Another
{
    int    a;
    double b;
    long   c;
};

#include <unpack>
```

这样，结构体Another的pack值将会受到影响。

或者，某一天，你重构了你的模块结构，将结构体Bar移到了另外一个文件中。由于pack和unpack都制定在很远的地方，你很难注意到它们，所以你忘了在Bar所在的另一个文件中对Bar指定响应的pack。最终导致行为的不一致。

所以，一个好的实践应该是：对每一个和系统全局pack方式不一致的结构体/共用体，都分别进行pack的指定。如下：

```
#include <pack_1>
struct Foo
{
    int    a;
    double b;
    long   c;
};
#include <unpack>

#include <pack_1>
struct Bar
{
    int    a;
    double b;
    long   c;
};
#include <unpack>

#include <pack_1>
struct Another
{
    int    a;
    double b;
    long   c;
};
#include <unpack>
```

这似乎违背了DRY的原则。DRY的定义是：“任何一项知识，在全系统中仅仅应该有唯一的，不含糊的、权威的表示”[1]。

而事实上，这三个结构体并非必然使用相同的pack方式，只是恰巧在某种情况都属于以1的方式来pack罢了，所以它们从本质上并不属于同一项知识。另外，由于绝大多数结构体都是使用默认的pack方式而不需要通过pack指令来指定，所以很容易造成维护上的忽视。一定程度的“重复”反而会降低维护的代价。

---

[1] 原文为“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”，From “The Pragmatic Programmer” by Andrew Hunt & David Thomas, 1999.

## 条款3. 尽量避免extern的使用

在你所工作的项目中，随意的打开一个代码文件，无论是头文件还是源代码文件，你是否能够看到到处存在的extern？如果答案是肯定的，很不幸，你的团队已经患上“extern依赖症”，而你的项目很可能已经处于混乱糟糕的状态。

extern几乎是C语言元素中最容易理解，却又最容易误用和滥用的。事实上，除了极少数的几种场景外[2]，其它对于extern的使用的都可以归于滥用，至少是无用(useless)。如果你尽力消除掉你项目中的extern，你的项目几乎肯定会转化到更好的状态——不！不是几乎，是确定、一定、以及肯定。

### extern + 变量声明

对于变量而言，extern只能用于修饰全局变量；并且，它只能用于全局变量声明，而不应该用于变量定义。它的作用是为当前的编译单元引入一个定义在其它编译单元的变量。从而让你可以在当前的编译单元使用这个变量，却不会招来编译器的抱怨。

```
extern int global_var_in_another_c_file;
```

由于通过extern引入的只是一个声明，编译器不会在当前编译单元内创建这个对象。如果和当前编译单元所生成的目标文件链接的其它目标文件里也没有定义这个对象，将会造成“符号缺失”的链接错误。

一旦你去掉了那个extern，上面的语句就变成了变量定义[3]。编译器将会在当前编译单元所产生的目标文件内创建这个对象，随后，当把这个目标文件和另外一个定义了同名对象的目标文件进行链接时，将会产生“符号冲突”。

从语法的角度而言，用extern引入一个全局变量声明是没有问题的。但从设计的角度而言，横跨多个编译单元的全局变量是一个糟糕的选择，它会造成不必要的模块间耦合。

绝大多数情况下，你都可以通过设计手段将全局变量消除，或者把它们控制在一个编译单元内部。这样你就没有必要通过extern把它们引入其它编译单元。

限制extern用于变量声明，可以迫使开发团队仔细的考虑对于代码的设计，模块的划分，从而达到降低系统耦合的目的。

### extern + 变量定义

下面形式的代码是一个全局变量定义，而不是声明：

```
extern int global_var = 10;
```

---

[2] 这些少数场景包括：调用使用汇编编写（非inline汇编）的函数、使用编译器/链接器生成的符号等。

[3] 在C99规范中，这种定义称为tentative definition，如果在其之前没有一个同类型、非static的同名定义，它就是一个未初始化的变量定义；如果在它之前已经存在一个同类型、非static的同名定义，则它是一个引用之前定义的一则声明。

其中的差别在于变量的初始化。变量的声明是不予许进行初始化的，一旦你进行初始化，编译器将把它看做一个定义。而全局变量的定义是不应该用extern来修饰的，虽然编译器仍然让其通过编译，却往往会给出一个警告。

## extern + 函数声明

当extern用于函数声明时，其用意和变量一样，都是为当前的编译单元引入一个外部函数声明。但这同样是一个糟糕的设计选择 ——

你之所以用extern引入一个函数声明，不外乎如下几种原因：

1. 你懒得去找对方的头文件；
2. 对方的头文件里没有声明这个函数；
3. 对方的头文件内容庞大复杂，一旦 # include它，轻则带来庞大的编译开销，重则导致莫名奇妙的编译错误和运行时错误，有时候，为了引入一个头文件，由于它不是自完备的，你还不得不自己再去寻找它所依赖的其它头文件。而你想使用的只有这个函数。

先看第一个原因。尽管Larry Wall的名言“优秀程序员的三大美德：骄傲，懒惰，缺乏耐心”字字珠玑，但你的这种懒惰却不在他的范畴。

因为，你一旦通过extern在自己的源代码中重新声明了这个函数，你就让系统中存在了两个对于同一函数的声明。这明显的违背了DRY原则。其后果是：如果有一天对方的函数原型发生了变化——既可能是参数个数变化，也可能是参数类型变化——而你的源代码文件不能感知到这种变化[4]。因为对这个函数的调用是按照你通过extern对这个函数进行声明方式来进行的。于是，编译器、链接器都没有发出任何警告，但一运行，却出现了异常。然后，你就不得不牺牲本来应该和家人共度的时光，没日没夜的加班解决这个本来在编译阶段就可以发现和避免的问题。

对于第二个原因，按照头文件设计原则，对方会把需要暴露给客户的函数接口都声明在他的头文件里。如果他没有声明，意味着他不想让你使用，你私自使用那些未公开的函数[5]，会破快掉对方信息隐藏的意图，引入不恰当的依赖，从而阻碍对方的重构和设计演进，让系统趋于僵化。另外，你同样冒着第一个原因所导致的风险。

至于最后一个，你这么做的理由合情合理，但解决问题的办法却很粗糙。是的，对方的头文件是很糟糕，但如果每个人都试图绕开，对方始终得不到任何反馈，只会让那个头文件保持糟糕的状态，或者越来越糟。另外，这种绕开也会让自己处于第一种原因所导致的风险之下。

正确的解决之道是：如果被依赖头文件属于自己的团队，则重构它，直到它达到简单、正确、自完备的状态为止。如果它属于别的团队，只要它是正确的，无论是否复杂和自完备 ，就先使用它，同时给对方提出相关的改进建议。如果对方连正确都保证不了，就要毫不妥协的当作bug给对方提问题单，迫使对方改进。只有这样，才能让整个系统的质量向健康的方向演化。

还有一种场景——自己在头文件里暴露给别人的函数声明要不要用extern?

```
#ifndef __FOO_H__
#define __FOO_H__

extern int foo(void); // 正确，却无用的extern

#endif
```

---

[4] 如果使用C++编译器，并且函数没有用extern “C”修饰，则链接器可以感知这种变化，因为C++会根据函数签名对函数名进行“名字粉碎”，最终存储在符号表中的名字是与函数签名紧密相关的。

[5] 虽然可以通过static的方式禁止外部的访问，但某些系统可能由于热补丁方案的，串口调试等等的需要而禁止程序员对于static的使用。

答案和在函数定义时使用extern是一样的：没有语法上的错误，却完全多余。因为一个函数声明默认的存储类别（storage class）就是extern，无论这个函数是否在本编译单元内有定义。

## extern + 函数定义

当一个函数定义用extern修饰时，其用意是告诉编译器，这个函数是外部可见的。这虽然合法，却完全多余。因为，即使你不使用extern，编译器也会让这个被定义的函数名字外部可见。

```
extern int foo(void) // 正确，却多余的extern
{
    return 0;
}
```

## 忘了它吧，忘了它的一切……

所以，在正常的情况下，几乎没有真正需要extern的场景。在一些场景下，它是多余的，在另外一些场景下，使用它给你带来的潜在问题和长远的麻烦，要大大超出它给你带来的即时便利。

在实践中，关掉这扇“后门”，克服掉extern依赖症，最终会让你的团队受益匪浅。

# 条款4. 充分利用static来改善设计

从语法的角度看，static和extern一样，都属于**存储类别限定符**（storage class specifier）；但从语义上，二者用来指定完全互斥的约束；而从工程学的价值上，两者亦有天壤之别。

你应该竭力避免extern的使用。然而，如果你是一个严谨的软件工程师，并在为一个真正的商业项目在工作，而不是仅仅把C语言当作玩具来开发一个微不足道的小程序，你就需要充分理解并利用static来改善你的设计。

众所周知，解决耦合问题的关键原则之一就是信息隐藏，即把客户所有不需要关心的信息尽力的隐藏起来，仅仅让客户依赖它应该依赖的东西。

static正是为此而生。它是C语言提供的唯一用来达到这一目标的机制。它可以用来帮助你强行隐藏模块间和模块内的不需要扩散的信息。

## static + 全局变量

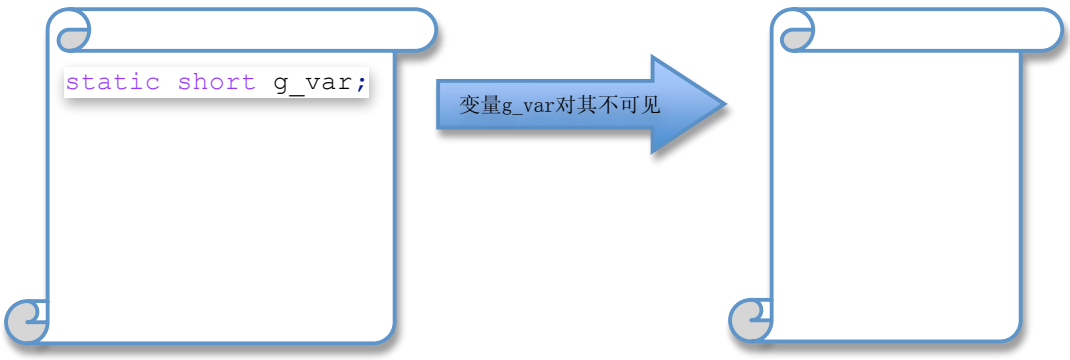
全局变量就像是毒品，用起来的时候很痛快（因为你根本不需要绞尽脑汁的进行设计），但会很快让你的系统处于病态，而你却无法从中轻易脱身。

一个设计良好的系统，应该是利用一系列的抽象，通过行为组合起来的产物。但全局变量作为一种实现细节，对其不加约束的使用是对这一原则的严重违背。

但基于性能，存储，及实现的简便性等方面的的考虑，全局变量有时候确实是较好的选择。

这种情况下，可以考虑使用static来修饰一个全局变量，结果是，这个所谓的“全局变量”的可见范围大大缩小，仅仅限于定义它的编译单元内，别的编译单元根本无法访问和使用它。

然后你把围绕这个全局变量的操作[6]都放置在这个编译单元内，仅仅向外暴露接口，就可以即隐藏实现细节，又满足你其它的目标。




当一个全局变量使用了static，就杜绝了别人在其它编译单元通过extern来引入此变量的可能。从语言机制上保证了别人破坏封装的可能性。

通过合理的模块划分和代码组织，你可以很容易的把那些全系统可见的全局变量转化为仅在模块内可见“全局变量”。这样，你对一个数据的所有相关操作都被高度内聚到一个模块[7]内部，而不是任由它像癌细胞一样扩散到系统的各处。

---

[6] 在绝大多数情况下，如果设计合理，这些操作不仅，也不应该是简单的set/get函数。

## static + 局部变量

用于全局变量的static用来控制其可见范围为当前编译单元。而用于局部变量的static则把变量的可见范围控制在函数内部。

它们之间的区别仅在于访问域的不同，其它的则完全一样——都处于数据段，都用来永久的保存状态。

但一个局部变量如果没有用static进行修饰，每次函数被调用时它都会重新从栈中分配出来，在函数调用结束后，将空间归还给栈空间。因此，常规的局部变量不能用于永久的保存状态。

如果一个需要永久保存状态的变量只需要被一个函数操作，那么不要客气，把它从全局域转移到这个函数内部。这样，既可以让代码的阅读者对于这个变量的使用范围一目了然，又可以防止某个粗心的维护者误用它。

## static + 函数

模块同过一组的函数向其它模块提供服务。为了完成这些函数，基于可维护性，模块内复用等等目的，我们往往需要实现一系列的其它函数。作为模块内的实现细节，我们不应该允许其它的模块对它们进行直接的调用。这种情况下，就需要使用static来进行这种约束。

和全局变量一样，当一个函数用static修饰时，此函数就仅仅在当前编译单元可见。杜绝了别人使用extern引入并调用这个函数的可能。

## 潜在的名字空间

由于C语言没有提供显式的名字空间概念，所有的外部符号都在一个空间里。为了避免冲突，C语言项目往往规定了一系列的规则，比如给名字统统加上“模块名_”、“子系统名_”等前缀。

但，这些前缀往往会占用多个字符，挤压了函数名有含义部分的字符数空间。尽管现代C语言编译器很少会对名字长度有限制，但当一个名字过长时，仍然可能会造成阅读和布局时的困难。这种情况下，程序员往往会倾向于使用缩写。这就对“名字自注释”的价值带来一定的冲击。

而static会给开发者带来这方面的便利。由于static限制了函数和变量的可见范围，就意味着不同模块的编写者在进行内部实现时，可以选择完全相同的名字，却不会造成链接时符号冲突。

所以，当你确定一个变量或函数仅仅是模块内实现细节时，就可以不加任何前缀，尽情的在长度合理的范围内对变量和函数进行命名。

## 但，世界总是不完美的……

为了提高竞争力，大部分7*24小时的服务器系统都具备在不需要重启系统的前提下给系统打补丁，以修复缺陷。这种方式被称为“热补丁”（hot patching）。

既然在运行时对系统进行打补丁，就要求其方案一定不要有副作用。一般而言，一个补丁是一组对原有函数的重新实现，而这些函数被放置在一个新的编译单元内，却需要对原有的数据或函数进行引用。

由于static对其编译单元隐藏了自己修饰的变量或函数，在全局的符号表中将不会存在这些变量或函数的名字。所以补丁所在的编译单元无法对它们进行访问。

---

[7] 这里所说的模块不是指业务模块，而是指功能高度内聚的软件功能模块。

而一旦这些补丁在自己的编译单元内再次定义这些名字所代表的变量和函数，则编译器会把它们生成全新的对象，它们和原有系统中的对象是不同的东西。这样的补丁将会导致系统运行至错误的结果。

为了避免这样的情况，相关团队往往会制定出类似“无条件禁止static的使用”的纪律。为了满足商业价值，牺牲一些编程的便利性应该是值得的。

但，它们真的非此即彼，水火不容吗?

## 和而不同

由于热补丁是发布版本里才需要的特性。那么我们有没有可能在开发版本里继续使用static，让它时时的约束和规范我们，帮助我们进行更好的设计呢?

宏，依然是宏，邪恶而强大的宏再次充当了救世主的角色。

我们可以定义一个宏，名字为STATIC，PRIVATE，INTERNAL都可以，重要的是，它要起到语义提示的作用，并能帮助我们在两种版本模式下进行切换。比如:

```
#if defined(__RELEASE_ENV__) && (__RELEASE_EVN__ != 0)
#define INTERNAL
#else
#define INTERNAL static
#endif
```

这样我们就可以在全局变量和函数前使用这个宏:

```
INTERNAL int g_var = 0;

INTERNAL int foo()
{
    return 0;
}
```

在开发的状态下，INTERNAL会自动替换为static，如果系统中存在私有访问的情况，链接过程将会失败，从而帮助团队防止耦合状况的恶化。但一旦要构建发布版本，只需要在加上一条-D__RELEASE_ENV__=1即会把INTERNAL替换为空，避免了static给热补丁带来的负面影响。

## 特别的方案给特别的局部变量

和全局变量不同，我们不能用INTERNAL来代替局部变量的static。

对于局部变量，影响的不仅仅是符号可见范围，更重要的是分配方式，以及状态永久性的不同。这会造成性能的差异，甚至导致程序行为的错误。

对于static的局部变量，我们需要不同的解决方案。

最简单的方案是，把这个局部变量变为全局变量。这行得通，但不够完美。因为，它让我们丧失了static局部变量所带来的价值。

更好的解决方案是：把static局部变量所在的函数降格为一个没有实在逻辑的函数，把真正的逻辑都提取到一个参数化的新函数里。比如一个函数的原有定义如下:

```
unsigned int handle(unsigned int id)
{
    static IdHandlerMap mapTable[] =
    {
      {ID_FOO, handle_foo},
      {ID_BAR, handle_bar}
    };

    size_t i = 0;
    for(i = 0; i < SIZEOF_ARRAY(mapTable); i++)
    {
        if(id == mapTable[i].id)
            return mapTable[i].handler();
    }

    return FAILED;
}
```

经过重构后，代码如下：

```
INTERNAL unsigned int __handle
             ( unsigned int id
             , IdHandlerMap* mapTable
             , size_t sizeOfTable)
{
    size_t i = 0;

    for(i = 0; i < sizeofTable; i++)
    {
        if(id == mapTable[i].id)
            return mapTable[i].handler();
    }

    return FAILED;
}

unsigned int handle(unsigned int id)
{
    static IdHandlerMap mapTable[] =
    {
        {ID_FOO, handle_foo},
        {ID_BAR, handle_bar}
    };

    return __handle( id, mapTable
                   , SIZEOF_ARRAY(mapTable));
}
```

由于重构后的原函数没有任何逻辑，也就不存在发生逻辑错误的可能。而它的错误只可能发生于数据错误和函数原型错误，而热补丁方案是不支持对于原有数据和函数原型修改的。所以就算这个函数在这些方面真的存在错误，你也不可能给它打热补丁。所以，以这种方式来使用static不会给热补丁方案带来实质性的影响。

这种方案带来的另外一个好处就是：那个带有逻辑的，你可以打补丁的函数变成了一个没有副作用的函数，也就是说，它是没有状态的（state free）的。如果它的逻辑发生了错误，你在对其打补丁的时候，无需引用任何系统已经存在的全局变量。这会让你的补丁更安全。

## 依然不完美的世界

经过这么多的努力，却仍然没有赢回全部。

对于那些由于热补丁或串口调试而禁止使用static的团队，将失去static所带来的名字空间的作用。尽管之前的方案让你可以在发布环境和开发环境之间进行切换，但由于发布环境最终没有static，所以，你还是要继续使用团队所制定的前缀规则来修饰所有的函数名。

# 条款5. 谨慎的使用位域

当我们需要用C语言数据类型来表示软硬件平台指定的描述符结构，以及某些网络协议的包格式时；或者描述为了节省内存而自定义的紧凑数据结构时；为了可读性，编码的方便性，我们会使用使用**位域**（Bit-Field）。例如：

```
struct Foo
{
   unsigned int  a:4;
   unsigned int  b:8;
};
```

至少到目前为止，绝大多数计算机（如果不是全部的话）都以字节为单位进行编址；另外，当我们进行sizeof运算时，得到的结果也是以字节为单位。

而位域的分配是基于比特的。这就意味着，你不能

- 对一个位域进行取地址操作
- 对一个位域进行指针操作；
- 定义位域数组
- 进行sizeof运算

这都合情合理，容易理解，当然不是问题。有问题的是：当你选择使用位域时，将冒着不确定的风险：位域相关的绝大多数行为都是未定义的，不同的编译器在实现这些未定义行为时，相互不兼容，从而导致严重的可移植性问题。而这正是我们这个专题将要讨论的内容。

## 不兼容的存储单元类型

**存储单元**（storage unit）指的是用来声明位域的整数类型。在C语言规范（无论是C89还是C99）中规定的**存储单元**只有int, signed int和unsigned int。至于其它整数类型，如(unsigned) char, (unsigned) short, (unsigned) long, (unsigned) long long等，是否可用作位域的**存储单元**，都取决于编译器的实现。

不过这一点在大多数情况下并不会带来困扰。因为大多数主流编译器都允许：在目标体系架构上所支持的所有的整数类型，均可用作位域的**存储单元**。

问题的真正麻烦之处在于：

- 当存储单元类型是有符号（signed）整数时，位域是否有符号，不同编译器有不同的实现。
- 在不同的平台上，C语言数据类型的长度都可能是不一样的。

对于第一个问题，会影响位域的取值范围。比如如果一个位域的定义为int a:1，在某些编译器的实现里，a的取值范围为[-1,0]，而另外一些编译器的取值范围则是[0,1]。如果一个位域的定义位int b:2，则其取值范围可能为[-2,1]或[0,3]，取决于编译器是否允许有符号的位域值。

而第二个问题，则会导致结构体大小，以及位域布局的不同。原因我们随后讨论。

## 不兼容的位域分配

C99规定：“在一个存储单元内，如果之前的位域分配之后，还剩下足够的比特位以分配紧随其后定义的下一个位域，则应该这个存储单元内，从第一个空闲比特位开始为这个位域进行分配”。

所以，在前面给出的例子里，位域a，b会在一个unsigned int的存储单元内分配。

但除此之外，C99标准没有再定义任何与位域分配相关的标准。一切都留给了编译器的实现者。

### 跨越边界时的不兼容

当一个存储单元分配了之前的位域后，如果剩下的比特位不足以放置紧随其后的下一个位域，那么下一个位域被放置在下一个存储单元还是让那个位域横跨两个存储单元的边界，将由编译器的实现者来决定。

比如在下面的定义中，由于a的位域宽度为28，在32位的存储单元里为止分配比特位之后，就只剩下4个比特位。而随后的位域b的宽度为8，第一个存储单元剩下的比特位小于其需要。

```
struct Foo
{
    uint32_t  a:28;
    uint32_t  b:8;
};
```

在自然对齐的方式下，gcc 4会将b分配在新的存储单元里，即不会让b横跨存储单元边界；但如果使用#pragma pack(n)来指定成员对齐，无论n为何值，b都将横跨存储单元的边界，即在第一个存储单元里分配其剩余的4个比特位，然后在新的存储单元里分配剩下的比特位。

而MSVC 9则无论何种方式，都会在新的存储单元里分配b。

### pack导致的不兼容

#pragma pack指令除了影响位域的存储单元边界问题，还会影响存储单元的大小。

比如，下面的结构体：

```
struct Foo
{
    uint32_t    a:4;
    uint32_t    b:4;
};
```

在自然对齐方式下，gcc 4和MSVC 9两种编译器对sizeof(Foo)的求值都是4。

但如果用#pragma pack(1)指定这个结构体的成员对齐方式，结果将会不同。如下：

```
#pragma pack(1)
struct Foo
{
    uint32_t    a:4;
    uint32_t    b:4;
};
#pragma pack()
```

如果用MSVC 9进行编译，则sizeof(Foo)依然等于4；但gcc 4对sizeof(Foo)的求值结果为1。也就是说，在 # pragma pack指令的影响下，gcc并不在意用户指定的存储单元，而是基于事实需要来自动选择合适的存储单元。

### 跨类型所导致的不兼容

如果两个连续位域定义的存储单元类型不一致，则位域分配算法在不同的编译器之间存在不兼容。比如下面的定义：

```
struct Foo
{
    uint32_t  a:4;
    uint32_t  b:12;
    uint16_t  c:4;
    uint16_t  d:12;
};
```

在IA-32平台上，a，b的存储单元类型为32位的unsigned int，而c，d的存储单元类型为16位的unsigned short。由于a，b的位域总宽度为16，而c，d的位域总宽度也是16，所以从理论上，c，d也可以在a，b所在的存储单元里分配，尽管它们所声明的存储单元类型和a，b并不相同。一些编译器确实是这样实现的。

但另外一些编译器却不这样理解。在它们看来，既然c，d的存储单元类型和a，b不一样，那么它们就理应在新的类型为unsigned short的存储单元内分配。

把上面的例子放在MSVC 9里进行编译，默认的情况下，一个Foo对象的大小为8，即使在pack(2)的情况下，其大小也是6 。如果用gcc 4编译，其大小则为4。

和pack(1)时采取的策略一样，gcc总是根据实际的需要来进行位域的分配，而MSVC则更加在意用户所指定的存储单元。

### 无名位域处理方式的不兼容

**无名位域**（unnamed bit-field）正如它名字本身所描述的那样，指的是没有名字的位域定义。

无名位域分为两种：

1. 宽度非零的无名位域
2. 宽度为零的无名位域

如果一个无名位域宽度非零，其位域分配方式和有名位域没有任何不同。只是，程序员并不需要关心它的名字。宽度非零的无名位域被用做**占位符**，用以映射外部定义中的保留比特。

比如在下面的定义中：

```
struct foo
{
    uint32_t    a:3;
    uint32_t    b:7;
    uint32_t     :22;  // 用作填充占位符的无名位域
    uint32_t    c:8;
    uint32_t    d:24;
};
```

其中第三个位域就是一个起到占位符作用的无名位域。

但一个宽度为零的无名位域却表示完全不同的作用。它就像一个**分割符**，以告诉编译器，随后的位域需要以新的方式进行分配。

但问题是，不同编译器对于新的方式的定义是不一致的。一般来说，分为两种方式：

1. 从新的存储单元进行分配；
2. 按照指定存储单元类型的对齐位置

比如对于下面的结构体：

```
struct foo
{
    uint32_t    a:8;
    uint32_t    b:12;
    uint8_t      :0;  // 宽度为0的无名位域
    uint8_t     c:4;
};
```

gcc 4采用的是第二种，将c的分配放置在以uint_8对齐的位置，即从相对于foo起始地址的第3个字节（以0开始）开始为c分配位域。

但MSVC 9则采用了第一种方式，即从相对于foo起始地址的第4个字节（以0开始）开始为c分配位域。

### 布局的不兼容

位域在存储单元内的布局，取决于位域在存储单元原内的分配顺序。但其分配顺序也是一种未定义行为。

一般而言，编译器实现的分配顺序有两种：

1. 从MSB到LSB[8]；
2. 或者相反，从LSB到MSB。

具体使用那种分配顺序，往往取决于目标平台的的大小端：当目标平台是大端时，则按照MSB到LSB的顺序进行分配；当目标平台是小端时，则按照相反的顺序进行分配。

---

[8] 在这里，LSB和MSB分别代表Least Significant Bit和Most Significant Bit。在其它上下文里，它们还可以代表Least Significant Byte和Most Significant Byte。

分配顺序的不同，将会导致程序员在定义跨CPU通信相关的数据结构时针对不同的情况进行不同的定义，比如：

```
struct foo
{
#if defined(__LITTLE_ENDIAN__) && !defined(__BIG_ENDIAN__)
   uint32_t   a:8;
   uint32_t   b:24;
#elif defined(__BIG_ENDIAN__) && !defined(__LITTLE_ENDIAN__)
   uint32_t   b:24;
   uint32_t   a:8;
#else
#error "no endian specified!"
#endif
};
```

## 应对可移植性问题

基于我们之前所讨论的种种不兼容问题，我们可以指定一系列的约束来让我们的位域相关的定义更具可移植性。

所以在选择位域存储单元类型的时候，应该：

1. **仅仅使用无符号类型，**
2. **使用uint8_t ，uint16_t，uint32_t等能够明确指定存储单元宽度的自定义类型。**

当我们需要用位域定义一个结构时，为了应对位域分配问题的不兼容，我们应该首先为这个结构定义一个平台无关的二进制布局。然后根据其二进制布局，应用如下的原则来进行实现：

1. **选择正确的存储单元类型**
2. **不要定义可能会进行跨越分配单元的位域；**
3. **不要使用宽度为0的无名位域；**
4. **明确的对一个存储单元的所有无用的比特进行填充；**
5. **处理大小端问题**

我们随后的例子重点放在前面四个原则。我们会在后面的小节里专门讨论第五个原则。

## 一些例子

### 例1

```
struct Foo
{
    uint32_t  a:4;
    uint32_t  b:12;
    uint16_t  c:4;
    uint16_t  d:12;
};
```

如果你想让a，b，c，d都放在一个32位的存储单元内，就不应该在在定义c和d的时候使用uint16_t，而是和a，b一样，都使用uint32_t来定义。如下：

```
struct Foo
{
    uint32_t  a:4;
    uint32_t  b:12;
    uint32_t  c:4;
    uint32_t  d:12;
};
```

但如果你确实想让c和d在一个新的16位存储单元内来分配，则应该将其定义为：

```
struct Foo
{
    uint32_t  a:4;
    uint32_t  b:12;
    uint32_t   :16; // 明确的指明填充
    uint16_t  c:4;
    uint16_t  d:12;
};
```

### 例2

```
struct Foo
{
    uint32_t    a:4;
    uint32_t    b:4;
};
```

如果你的意图是想让a和b在一个字节内分配，则应该将定义改为：

```
struct Foo
{
    uint8_t  a:4;
    uint8_t  b:4;
};
```

而如果你的意图是想使用32位的存储单元，则应该将其定义修改为：

```
struct Foo
{
    uint32_t  a:4;
    uint32_t  b:4;
    uint32_t   :24; // 显示的定义填充
};
```

例3

```
struct Foo
{
    uint16_t  a:12;
    uint16_t  b:8;
};
```

这里b可能横跨存储单元uint16_t的边界。

如果你的意图是让a和b分别使用自己分配单元，则应该将定义改为：

```
struct Foo
{
    uint16_t  a:12;
    uint16_t   :4; // 明确的指明4个比特的填充

    uint16_t  b:8;
};
```

但如果你的意图是想让b和a之间没有填充，则应该将其定义如下：

```
struct Foo
{
    uint32_t  a:12;
    uint32_t  b:8;  // 将存储单元合并为32位
    uint32_t   :12; // 并明确的指定填充
};
```

## 简单的处理大小端问题

由于大小端的位域分配顺序正好相反，这种规律性让你不必对同一组位域定义两次。

而是可以定义一组下面的宏：

```
#if   defined(__BIG_ENDIAN__) && !defined(__LITTLE_ENDIAN__)

# define BITFIELD2(type, bf1, bf2)       type bf2, bf1
# define BITFIELD3(type, bf1, bf2, bf3) type bf3, bf2, bf1

#elif defined(__LITTLE_ENDIAN__) && !defined(__BIG_ENDIAN__)

# define BITFIELD2(type, bf1, bf2)       type bf1, bf2
# define BITFIELD3(type, bf1, bf2, bf3) type bf1, bf2, bf3

#else
#error "no endian specified!"
#endif
```

然后，在每个定义位域的地方使用这些宏。比如下面的定义：

```
struct foo
{
#if defined(__LITTLE_ENDIAN__) && !defined(__BIG_ENDIAN__)
    uint32_t    a:8;
    uint32_t    b:24;
    uint16_t    c:4;
    uint16_t     :4;
    uint16_t    d:8;
#elif defined(__BIG_ENDIAN__) && !defined(__LITTLE_ENDIAN__)
    uint32_t    b:24;
    uint32_t    a:8;
    uint16_t    d:8;
    uint16_t     :4;
    uint16_t    c:4;
#else
#error "no endian specified!"
#endif
    uint16_t    e;
};
```

可以修改为：

```
struct foo
{
    BITFIELD2(uint32_t, a:8, b:24);
    BITFIELD3(uint16_t, c:4, :4, d:8);

    uint16_t    e;
};
```

## 终极解决方案

我们之前所有的应对措施都并没有彻底的解决，而是仅仅提高了可移植性。

比如，如果一种编译器不允许指定(unsigned) int之外的其它存储单元类型，我们便无法使用uint8_t或uint16_t来定义位域。

而最具有可移植性的解决方案是：不要使用位域，而是通过对相应的整数类型进行**移位**（shift >>和 <<），以及对比特位的“**与或非**”（& | ~）运算来达到相同的效果。要知道，C语言的位操作是大小端无关的。编译器会在背后会帮你处理这些差异。

比如，一个结构原来的定义如下：

```
struct foo
{
    BITFIELD2(uint16_t, a:4, b:12);
};
```

我们可以定义一组工具宏：

```
#define __BITMASK(offset, length) \
          (((1<<length)-1)<<offset)

#define __CLEAR(target, offset, length) \
          ((target)&~__BITMASK(offset, length))
#define __VALUE(value, offset, length) \
          ((value<<offset)&__BITMASK(offset, length))

#define __SET_VALUE(target, value, offset, length) \
          (__CLEAR(target, offset, length) \
          | __VALUE(value, offset, length))
#define __GET_VALUE(target, offset, length) \
          ((target&__BITMASK(offset, length)) >> offset)
```

然后使用这组工具宏将上面的结构处理为：

```
struct foo
{
    uint16_t value;
};

#define SET_A(foo, a)  do { \
      (foo).value = __SET_VALUE((foo).value, a, 0, 4); \
} while(0)

#define SET_B(foo, b)  do { \
      (foo).value = __SET_VALUE((foo).value, b, 4, 12); \
} while(0)

#define GET_A(foo)   __GET_VALUE((foo).value, 0, 4)
#define GET_B(foo)   __GET_VALUE((foo).value, 4, 12)
```