

---

---

*详解* *Structure*(结构体)  $\longleftrightarrow$

Copyright © 2013 Long Luo.

---

---

## Revision History

### 版本历史

Version 版本	Editing Note 修订历史	Reviser/Date 修订人/日期	Confirm/Date 确认/日期	Approval/ Date 批准/日期	Implement Date 实施/日期
1.0	创建	罗龙	05/12/2012		
1.1	修改	罗龙	23/12/2012		
1.2	完善	罗龙	23/12/2013		

# Table Of Contents

Revision History.....	2
Table Of Contents.....	3
引言.....	4
一、从女孩怎么选男朋友开始...Struct 是为了解决什么问题? .....	5
二、Struct 的声明、定义及初始化.....	11
2.1 Struct 的声明.....	11
2.2 Struct 的定义.....	12
2.3 C99 标准的 Struct 的指定初始化方法.....	13
三、C++中 Class 与 Struct 的区别.....	15
四、C 语言 Struct 中的函数与函数指针.....	19
五、以空间换时间---Struct 中成员变量的对齐之道(上).....	23
六、以空间换时间---Struct 中成员变量的对齐之道(下).....	28
6.1 编译过程详解.....	28
6.1.1 预处理部分.....	29
6.1.2 编译成汇编代码.....	30
6.1.3 编译成目标代码.....	30
6.1.4 编译成执行代码.....	30
6.2 假如我们不对齐.....	31
6.2.1 对齐是为了换取效率.....	32
6.3 如何自定义对齐.....	32
七、无即是有---没有成员变量的 Struct.....	35
八、Struct 中 0 元素数组的意义.....	42
结束语.....	43

# 引言

这篇文档的构思起源于前几天在回家的火车上，当时带着一本《C 和指针》。回深圳之后，就在平时花了大概 2 周的时间将此文档完善。

# 一、从女孩怎么选男朋友开始...Struct 是为了解决什么问题？

“关关雎鸠，在河之洲。窈窕淑女，君子好逑”，《诗经》三百篇，开篇就是男女之间的恋情，可见几千年的古人也十分重视爱情。这也难怪，毕竟男女的婚姻是人伦之始，而且含有成家立业的意思。引用生物学的观点来解释，就是“求食求偶是关系到人类生存繁衍的大事”，能不重视么？

在我们的老祖宗还住在山洞里的那个时代，野外看到一个漂亮的女野人，一棍子敲晕，拖进洞里...不过那个年代已经一去不复返了。随着人类的进步，具体到现在这个社会，现代的女人都要求男方高富帅，有车有房...当然按照进化心理学的观点来看，**这些东西都代表着男性获取资源的能力**，而智人(人类)的后代是很脆弱的，所以女性将男性所获取的资源和获取资源的能力至于第一位的。

不过，由于拜国内的房地产所赐，身为一名 D 丝的话，想要追到一个女孩，也变得异常困难，一方面是硬件上的劣势，比如外表、车、房子、一份体面的工作灯；另外一方面又有软件上的劣势，比如幽默感，人品如何、性格等。付出的服务项目也越来越多，既要送花，要帮女孩做这个做那个表决心，还要送这个送那个表付出。

据说 20 年后国内将有 3000w 男性光棍，女孩也就成了卖方市场，眼前这么多追求者，高富帅各方面程度都不一样，应该把哪个放在第一位呢？该怎么选呢？

比如一位美女，就有 3 位男性追求者，比较来比较去，某天决定先按照“帅”的程度排个序，选一个最“帅”的：

```
1. /*****
   *****/
2. ** File: - Z:\work\code\c\Struct\WhyUsingStuct.c
3. **
4. ** Copyright (C) Long.Luo, All Rights Reserved!
5. **
6. ** Description:
7. **     WhyUsingStruct.c ---
8. **
9. ** Version: 1.1
10.** Date created: 22:25:44,20/12/2012
11.** Author: Long.Luo
12.**
13.** ----- Revision History: -----
   --
14.** <author> <data>          <desc>
15.**
16. *****/
17.
18.#include <stdio.h>
```

```

19.
20./* Number of the boys */
21.#define BOYS_NUM    (3)
22.
23.void main()
24.{
25.    int i, j;
26.
27.    /* Name */
28.    char name[BOYS_NUM][10];
29.
30.    /* height */
31.    int height[BOYS_NUM];
32.
33.    /* rich */
34.    int money[BOYS_NUM];
35.
36.    /* handsome */
37.    int handsome[BOYS_NUM];
38.
39.    /* pointer array of the boys' name */
40.    char *pName[BOYS_NUM];
41.
42.    /* the temporary */
43.    int heightTemp, moneyTemp, handsomeTemp;
44.    char *nameTemp;
45.
46.    for (i = 0; i < BOYS_NUM; i++)
47.    {
48.        pName[i] = name[i];
49.    }
50.
51.    for(i = 0; i < BOYS_NUM; i++)
52.    {
53.        printf("Pls input the Name of the No. %d Boys:", i + 1);
54.        gets(pName[i]);
55.        if (*pName[i] == '\0')
56.        {
57.            gets(pName[i]);
58.        }
59.
60.        printf("Pls input the Height of %s : ", pName[i]);
61.        scanf("%d", &height[i]);
62.        printf("Pls input the Money of %s : ", pName[i]);

```

```

63.         scanf("%d", &money[i]);
64.         printf("Pls input the Handsome of %s : ", pName[i]);
65.         scanf("%d", &handsome[i]);
66.     }
67.
68.     /* sort by Height */
69.     /* Only write one item. */
70.     for (i = 0; i < BOYS_NUM - 1; i++)
71.     {
72.         for (j = i + 1; j < BOYS_NUM; j++)
73.         {
74.             if (handsome[i] < handsome[j])
75.             {
76.                 nameTemp = pName[i];
77.                 pName[i] = pName[j];
78.                 pName[j] = nameTemp;
79.
80.                 handsomeTemp = handsome[i];
81.                 handsome[i] = handsome[j];
82.                 handsome[j] = handsomeTemp;
83.             }
84.         }
85.     }
86.
87.     for (i = 0; i < BOYS_NUM; i++)
88.     {
89.         printf("\nThe Boys's info: %s\t, height: %d\t, money: %d\t, handsome: %d\t");
90.     }
91.
92.     getchar();
93. }

```

但是上面的代码有很明显缺点：

1. 变量过多，同一追求者的各个数据无联系，没有整体概念，不便管理；
2. 操作不便，假如某天想把“富”作为第一考虑呢？或者根据不同面采取不同的加权来选择呢？

**男人，不止一面(七匹狼广告)。**

一个事物往往有很多的特征，但是人们往往去表达事物的时候，不是说特征，而是讲整体。零碎的信息、有时候很难替代一个整体信息结构。显然，选用一种能把一个追求者的数据构造成一个整体的构造型数据结构更合适，但不能是数组。对于这种情况，可以将一个追求者的数据定义为一个"ExpectedBoyFriend"结构体类型：

```

1. struct ExpectedBoyFriend
2. {
3.     char name[10];
4.     int height;
5.     int money;
6.     int handsome;
7. };

```

有时需要将不同类型的数据组合成一个有机的整体，以供用户方便地使用，而这些组合在一个整体中的数据是互相联系的。C 和 C++ 允许用户自己指定这样一种数据类型，它称为结构体，在一个组合项中包含若干个类型不同（当然也可以相同）的数据项。

类似上例，我们就声明了一个新的结构体类型 ExpectedBoyFriend，它向编译系统声明：这是一种结构体类型，它包括 name, height, money, handsome 等不同类型的数据项。ExpectedBoyFriend 是一个类型名，它和系统提供的标准类型（如 int、char、float、double 等）一样，都可以用来定义变量，只不过结构体类型需要事先由用户自己声明而已。

声明一个结构体类型的一般形式为

```

1. struct tag
2. {
3.     member-list
4. }
5. variable-list

```

结构体类型名用来作结构体类型的标志。上面的声明中 ExpectedBoyFriend 就是结构体类型名。大括号内是该结构体中的全部成员(member)，由它们组成一个特定的结构体。上例中的 name, height, money, handsome 等都是结构体中的成员。在声明一个结构体类型时必须对各成员都进行类型声明：

```

1. 即类型名 成员名;

```

每一个成员也称为结构体中的一个域(field)。成员表列又称为域表。成员名的定名规则与变量名的定名规则相同。

我们再看看使用 Struct 之后的代码：

```

1. /*****
   *****/
2. ** File: - Z:\work\code\c\Struct\UsingStruct.c
3. **
4. ** Copyright (C) Long.Luo, All Rights Reserved!
5. **
6. ** Description:
7. **     UsingStruct.c ---
8. **

```



```

9.  ** Version: 1.2
10. ** Date created: 22:52:29,20/12/2012
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
14. ** <author> <data> <desc>
15. **
16. *****/
17.
18. #include <stdio.h>
19.
20. /* Number of the boys */
21. #define BOYS_NUM    (3)
22.
23.
24. struct ExpectedBoyFriend
25. {
26.     char name[10];
27.     int height;
28.     int money;
29.     int handsome;
30. };
31.
32.
33. void main()
34. {
35.     struct ExpectedBoyFriend ebf[BOYS_NUM];
36.     struct ExpectedBoyFriend ebfTemp;
37.     int i, j;
38.
39.     for (i = 0; i < BOYS_NUM; i++)
40.     {
41.         printf("Pls input the Name of the No. %d Boys:", i + 1);
42.         gets(ebf[i].name);
43.         if (ebf[i].name[0] == '\0')
44.         {
45.             gets(ebf[i].name);
46.         }
47.
48.         printf("Pls input the Height of %s : ", ebf[i].name);
49.         scanf("%d", &ebf[i].height);
50.         printf("Pls input the Money of %s : ", ebf[i].name);

```

```

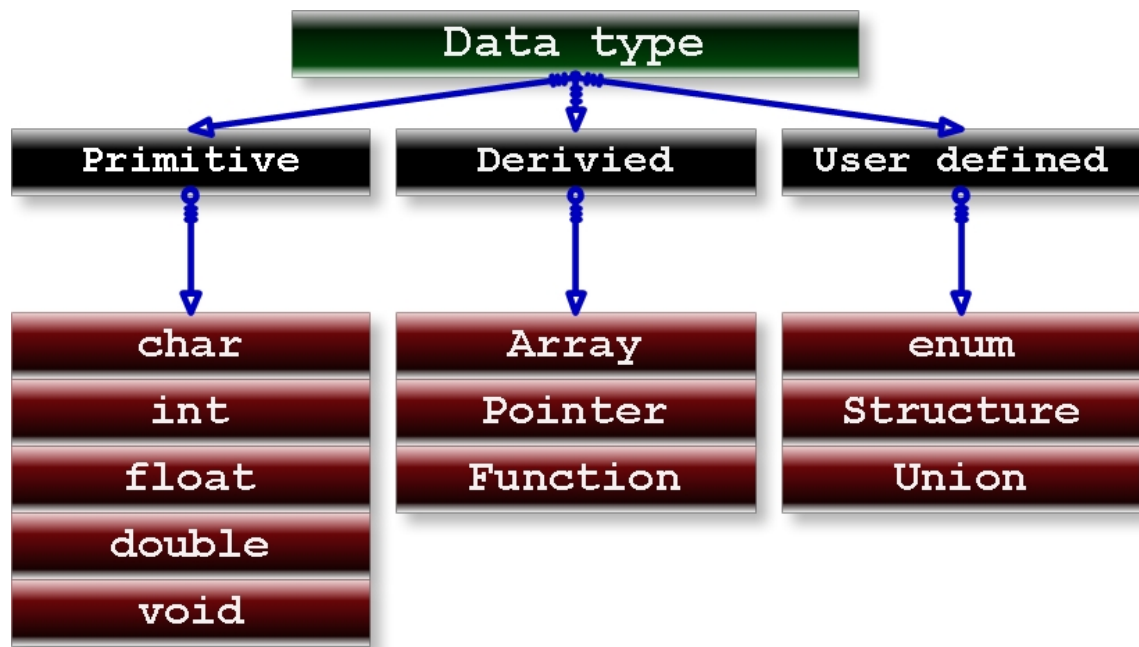
51.     scanf("%d", &ebf[i].money);
52.     printf("Pls input the Handsome of %s : ", ebf[i].name);
53.     scanf("%d", &ebf[i].handsome);
54. }
55.
56. /* Sort by Height */
57. /* Only write one item. */
58. for (i = 0; i < BOYS_NUM - 1; i++)
59. {
60.     for (j = i + 1; j < BOYS_NUM; j++)
61.     {
62.         if (ebf[i].height < ebf[j].height)
63.         {
64.             ebfTemp = ebf[i];
65.             ebf[i] = ebf[j];
66.             ebf[j] = ebfTemp;
67.         }
68.     }
69. }
70.
71. for (i = 0; i < BOYS_NUM; i++)
72. {
73.     printf("%s: \tHeight:%d, \tMoney: %d, \tHandsome:%d\n",
74.         ebf[i].name, ebf[i].height, ebf[i].money, ebf[i].handsome);
75. }
76.
77. getchar();
78.}

```

代码是不是变得简洁多了呢？

## 二、Struct 的声明、定义及初始化

上一篇里我们讲了为什么我们要引入 Struct 这个数据类型，我们了解到 Struct 是一种**聚合数据类型**，是为了用户描述和解释一些事物的方便而提出的，Struct 是一种用户自定义数据类型，如下图所示：



其实从理论上讲，**数据类型就是人为制订的如何解释内存中的二进制数的协议**，也就是说一个数字对应着一块内存（可能 4 字节，也可能 20 字节），而这个数字的类型则是附加信息，**以告诉编译器当发现有对那块内存的操作语句（即某种操作符）时，要如何编写机器指令以实现那个操作**。比如两个 char 类型的数字进行加法操作符操作，编译器编译出来的机器指令就和两个 long 类型的数字进行加法操作的不一样，也就是所谓的“如何解释内存中的二进制数的协议”。

具体到我们之前的例子来说，只是指定了一种结构体类型，它相当于一个模型，但其中并无具体数据，系统也不为之分配实际的内存单元。为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并在其中存放具体的数据。

本篇将详细对 Struct 的声明、定义和初始化进行分析。

### 2.1 Struct 的声明

要了解 Struct 的声明，我们需要首先了解**声明的含义**到底是什么？

---**声明是要求编译器产生映射元素的语句**。所谓的映射元素，就是前面介绍过的变量及函数，都只有 3 栏（或 3 个字段）：类型栏、名字栏和地址栏（成员变量类型的这一栏就放偏移值）。即编译器每当看到声明语句，就生成一个映射元素，并且将对应的地址栏空着，然后留下一些信息以告诉连接器——此\*.obj 文件（编译器编译源文件后生成的文件）需要一些符号，将这些符号找到后再修改并完善此\*.obj 文件，最后链接。

回想之前说过的符号的意思，它就是一字符串，用于编译器和链接器之间的通信。注意符号没有类型，

因为连接器只是负责查找符号并完善（因为有些映射元素的地址栏还是空的）中间文件不进行语法分析，也就没有什么类型。

具体到上一回的例子，我们假如在另外一个源文件中需要使用`struct ExpectedBoyFriend`，那么就需要在该源文件使用之前处使用下面的声明语句：

```
1. extern struct ExpectedBoyFriend;
```

## 2.2 Struct 的定义

上一小节我们了解了声明的定义，那么定义是什么呢？

---定义是要求编译器填充前面声明没有书写的地址栏。也就是说某变量对应的地址，只有在其定义时才知道。因此**实际的在栈上分配内存等工作都是由变量的定义完成的**，所以**才有声明的变量并不分配内存**。但应注意一个重点，定义是生成映射元素需要的地址，因此定义也就说明了它生成的是哪个映射元素的地址，而如果此时编译器的映射表（即之前说的编译器内部用于记录映射元素的变量表、函数表等）中没有那个映射元素，即还没有相应元素的声明出现过，那么编译器将报错。

在这里我们需要说下 C 和 C++在定义 Struct 的区别， 先看下面 2 段代码：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct SIMPLE
6. {
7.     int a;
8.     char b;
9.     float c;
10. };
11.
12. SIMPLE x;
```

再看下面一段源码：

```
1. #include <stdio.h>
2.
3. struct S0
4. {
5.     char mName[10];
6.     int mBornYear;
7. };
8.
9. typedef struct _S1
```

```

10. {
11.     char mName[10];
12.     int mBornYear;
13. }
14. S1;
15.
16.
17. S0 sa;
18. S1 sb;

```

那么上面的代码中对 Struct 的定义都对了吗？

熟悉 C/C++ 的同学应该能够马上知道下面的代码错了？

为什么呢？

---因为 C 语言中对于 Struct 的定义是需要使用 `struct S0 sa` 这种方式。

## 2.3 C99 标准的 Struct 的指定初始化方法

Struct 的常见初始化方法我们可以在任何一本关于 C 语言书里面都可以找到，这里就不赘述了。我们先看下面一段代码：

```

1. static struct usb_driver usb_storage_driver = {
2.     .owner = THIS_MODULE,
3.     .name = "usb-storage",
4.     .probe = storage_probe,
5.     .disconnect = storage_disconnect,
6.     .id_table = storage_usb_ids, };

```

我们在阅读 GNU/Linux 内核代码时，我们会经常遇到上述这样一种特殊的结构初始化方式，这种方式称为指定初始化。这种初始化方法源自 C99 标准。

以下文字摘录了 C Primer Plus 第五版中相关章节的内容，从而就可以很好的理解 Linux 内核采用这种方式的优势就在于由此初始化不必严格按照定义时的顺序。

已知一个结构，定义如下

```

1. struct Book
2. {
3.     char title[MAXTITLE];
4.     char author[MAXAUTHOR];
5.     float value;
6. };

```

C99 支持结构的指定初始化项目，其语法与数组的指定初始化项目近似。只是，结构的指定初始化项目使用点运算符和成员名（而不是方括号和索引值）来标识具体的元素。例如，只初始化 book 结构的成员 value，可以这样做：

```
1. struct Book surprise =  
2. {  
3.     .value = 10.99  
4. };
```

可以按照任意的顺序使用指定初始化项目：

```
1. struct Book gift =  
2. {  
3.     .value = 25.99,  
4.     .author = \"James Broadfool\",  
5.     .title = \"Rue for the Toad\"  
6. };
```

正像数组一样，跟在一个指定初始化项目之后的常规初始化项目为跟在指定成员后的成员提供了初始值。另外，对特定成员的最后一次赋值是它实际获得的值。例如，考虑下列声明：

```
1. struct Book gift =  
2. {  
3.     .value = 18.90,  
4.     .author = \"hillionna pestle\",  
5.     0.25  
6. };
```

这将把值 0.25 赋给成员 value，因为它在结构声明中紧跟在 author 成员之后。新的值 0.25 代替了早先的赋值 18.90。

### 三、C++中 Class 与 Struct 的区别

之前一篇我们了解了 C++ 中 Struct 的定义方法和 C 中有点不一样，而且增加了一种新的类型---Class。

从 C++ 的名字我们就可以知道，C++ 是从 C 进化而来，“++”就是在 C 的基础上加了一些东西：[面向对象的东西](#)。

虽然 C++ 作为一种面向对象语言，要区别于面向过程的 C 语言，但是在设计时，[一个很重要的原则是 C++ 必须向前兼容 C，必须是 C 的超集](#)。这样一来就可以带来好多好处：

第一个嘛，首先呢，C++ 就可以站在 C 这个巨人的肩膀上，大量过去用 C 编写的程序可以不加修改地在 C++ 环境下使用；

第二，把很多 C 程序员忽悠进 C++ 这个大坑里，为 C++ 之崛起而加班，上了贼船可就由不得你了 XD

.....

也正是因为这个原因，C++ 中保留了 struct 结构类型，并使得 struct 的功能更强大，不仅仅是简单继承 C 的结构体，而且扩充了 struct，使得它也具有类的特点，那么在 C++ 中，class 和 struct 到底有什么区别呢？

Talk is cheap, show me the Code!

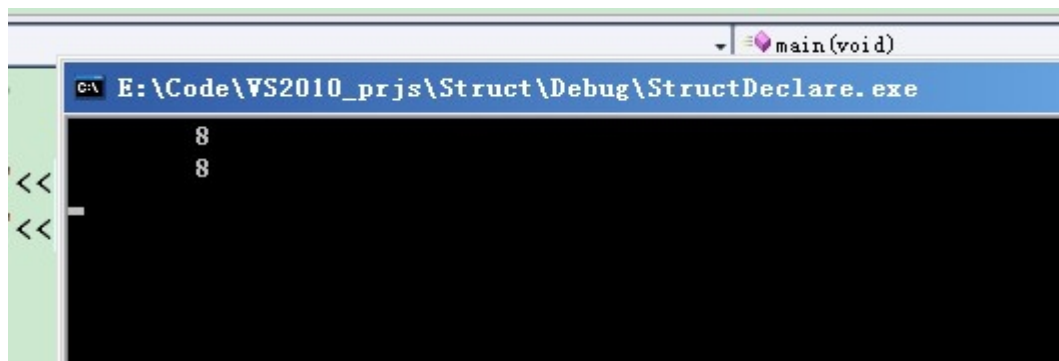
```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct S1
6. {
7.     char mA;
8.     int mB;
9. };
10.
11. class C2
12. {
13.     char mA;
14.     int mB;
15. };
16.
17.
18. int main(void)
19. {
20.     S1 a;
21.     C2 b;
22.     cout<<sizeof(a)<<&a<<endl;
23.     cout<<sizeof(b)<<&b<<endl;
```

```

24.
25.     getchar();
26.     return 0;
27. }

```

上面这段代码非常简单，分别定义了一个 Struct 类型和 class 类型，并输出其大小和地址，我们先看看输出结果：



从结果我们可以看出，没啥区别啊！

且慢，再看下面这段代码：

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. struct S1
6. {
7.     char mA;
8.     int mB;
9. };
10.
11. class C1
12. {
13.     char mA;
14.     int mB;
15. };
16.
17. struct S2
18. {
19.     char mA;
20.     int mB;
21.
22.     void foo()
23.     {
24.         cout<<"mA="<<mA<<endl;
25.         cout<<"mB="<<mB<<endl;

```



```

26.     }
27. };
28.
29. class C2
30. {
31.     char mA;
32.     int mB;
33.
34.     void foo()
35.     {
36.         cout<<"mA="<<mA<<endl;
37.         cout<<"mB="<<mB<<endl;
38.     }
39. };
40.
41.
42. int main(void)
43. {
44.     S1 a;
45.     C1 b;
46.
47.     S2 c;
48.     C2 d;
49.
50.     cout<<sizeof(S1)<<"\t"<<&a<<endl;
51.     cout<<sizeof(C1)<<"\t"<<&b<<endl;
52.
53.     c.foo();
54.     d.foo();
55.
56.     getchar();
57.     return 0;
58. }

```

编译，结果报错了，如下所示：

```

1. 1>e:\code\vs2010_prjs\struct\structdeclare\classandstruct.cpp(54): error C2248:
   "C2::foo": 无法访问 private 成员(在“C2”类中声明)
2. 1>          e:\code\vs2010_prjs\struct\structdeclare\classandstruct.cpp(34) :
   参见“C2::foo”的声明
3. 1>          e:\code\vs2010_prjs\struct\structdeclare\classandstruct.cpp(30) :
   参见“C2”的声明

```

编译提示说 C2 中函数 foo 是没有权限访问 C2 的成员变量的，是因为 **Class C2 里面的成员变量是 private**

的，而 Struct S2 就没有这个问题，因为默认成员变量都是 **public** 的。

Class 可以完全替代 Struct，只不过是由于 C++是在 C 的基础上设计的，所以保留了 struct 这个遗产。而 Java 作为一种完全面向对象设计的语言，就没有 Struct 了。

最后总结下，C++中的 struct 与 class 的区别：

1. class 中默认的成员访问权限是 **private** 的，而 struct 中则是 **public** 的；
2. 从 class 继承默认是 **private** 继承，而从 struct 继承默认是 **public** 继承。

细心的同学也会发现，C++中的 Struct 是可以拥有函数体的，这是 C 所不允许的，那么 C 语言中要实现函数的话，应该以一种什么样的形态出现呢？

请看下回《C 语言 Struct 中的函数和函数指针》。

## 四、C 语言 Struct 中的函数与函数指针

在上一回的文末中我们留了一个悬念，我们了解到 C 语言规范是 struct 里面是不能有函数体的，但是在应用中假如 struct 中没有函数的话，我们会遇到很多问题：

第一数据往往是依附于函数来进行操作的；  
其二是我们需要用 C 来实现面向对象的思想。

比如下面这段代码：

```
1. #include <stdio.h>
2.
3. struct FuncInside
4. {
5.     int mA;
6.     void func()
7.     {
8.         printf("Hello, function inside!\n");
9.     }
10.};
11.
12.
13.void main(void)
14.{
15.    struct FuncInside f;
16.
17.    f.mA = 99;
18.    f.func();
19.
20.    getchar();
21.}
```

编译会提示：

```
1. 1>e:\learn\vs\struct\struct\funcpointer.c(7) : error C2032: “func”：函数不能
   是 struct“FuncInside” 的成员
```

那么这个问题应该如何解决呢？

一刹那，一句话在脑海中闪现，“指针是 C 语言的精华。”

啊哈，灵机一动！

虽然 Struct 中不能有函数体，但是我们可以使用函数指针来实现同样的目的。

先来讲一讲什么叫 函数指针？

函数指针，注意要区别于指针函数，声明格式如下：

**函数类型 (标志符 指针变量名) (形参列表);**

象某一数据变量的内存地址可以存储在相应的指针变量中一样，函数的首地址也以存储在某个函数指针变量里的。这样，我们就可以通过这个函数指针变量来调用所指向的函数了。

举个例子来说明吧：

**int (\*pfun)(int, int);**

通过括号强行将 pfun 首先与“\*”结合，也就意味着，pfun 是一个指针，接着与后面的“()”结合，说明该指针指向的是一个函数，然后再与前面的 int 结合，也就是说，该函数的返回值是 int。由此可见，**pfun 是一个指向返回值为 int 的函数的指针。**

注意要区别于指针函数：

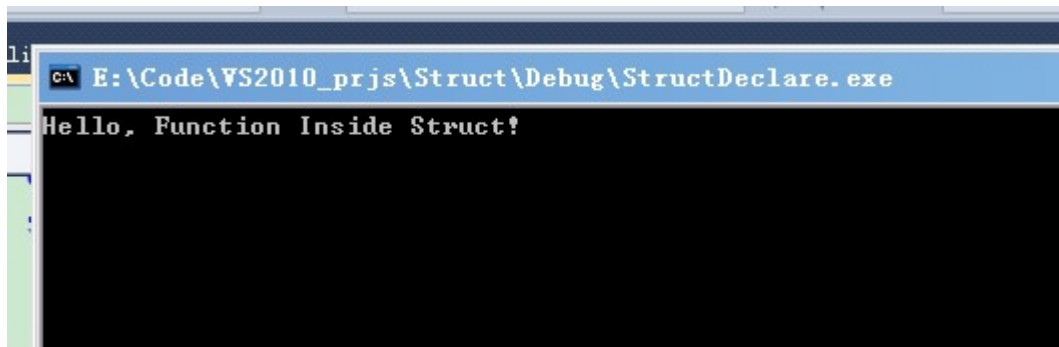
**int \*pfun(int, int);**

你注意到它们之间的区别了吗？

我们修改下之前的代码：

```
1. #include <stdio.h>
2.
3. struct FuncInside
4. {
5.     int mA;
6.     void (*pFunc)();
7. };
8.
9. void Foo()
10. {
11.     printf("Hello, Function Inside Struct!\n");
12. }
13.
14.
15. void main(void)
16. {
17.     struct FuncInside f;
18.
19.     f.mA = 99;
20.     f.pFunc = Foo;
21.
22.     f.pFunc();
23.
24.     getchar();
25. }
```

编译顺利通过，输出也是我们期望的结果。



之前 `int (*pFun)(int, int)`, 其中 `pFun` 是一个函数指针。而事实上, 为了代码的移植考虑, 一般使用 `typedef` 定义函数指针类型:

**`typedef int (*pFun)(int, int);`**

当使用 `typedef` 声明后, 则 `pFun` 就成为了一个函数指针“类型”, 即一种函数回调类型。这其实是一种回调函数的实现方法。

一个简单的例子如下:

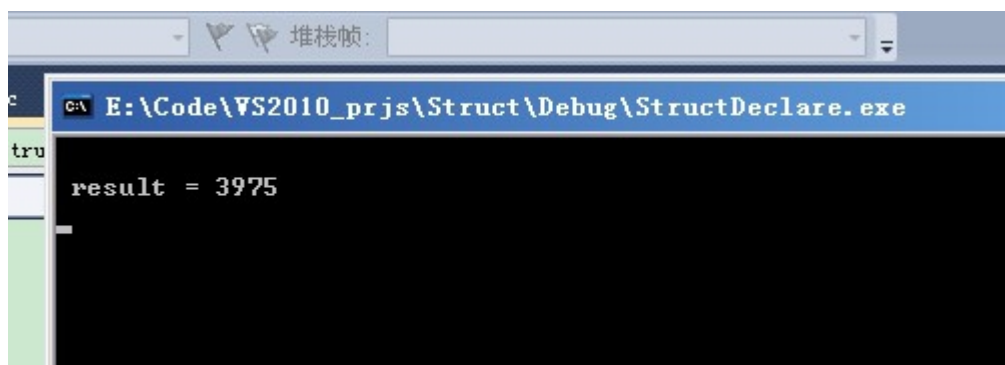
```
1. /*****
   *****/
2. ** File: - E:\Code\VS2010_prjs\Struct\StructDeclare\pFunCallBack.c
3. **
4. ** Copyright (C), Long.Luo, All Rights Reserved!
5. **
6. ** Description:
7. **     pFunCallBack.c - A demo show the usage of function pointer.
8. **
9. ** Version: 1.0
10. ** Date created: 01:03:04,09/12/2012
11. ** Author: Long.Luo
12. **
13. ** ----- Revision History: -----
   --
14. ** <author> <data>          <desc>
15. **
16. *****/
17.
18. #include <stdio.h>
19.
20. // Defines the function callback type.
21. typedef int (*pFun)(int paraA, int paraB);
22.
23. struct FuncPtr
24. {
```

```

25.     int x;
26.     int y;
27.
28.     // A function pointer to the implementation of the Summary.
29.     pFun GetSum;
30.};
31.
32.// The function of summary.
33.int GetSum(int paraA, int paraB)
34.{
35.     return (paraA + paraB);
36.}
37.
38.void main(void)
39.{
40.     struct FuncPtr fp;
41.     int result = 0;
42.
43.     fp.x = 1987;
44.     fp.y = 1988;
45.
46.     fp.GetSum = GetSum;
47.     result = fp.GetSum(fp.x, fp.y);
48.     printf("\n result = %d\n", result);
49.
50.     getchar();
51.}

```

输出结果如下：



至此，我们了解了 C 语言 struct 中是如何通过函数指针来实现函数的。

## 五、以空间换时间---Struct 中成员变量的对齐之道(上)

请先看一道面试题：

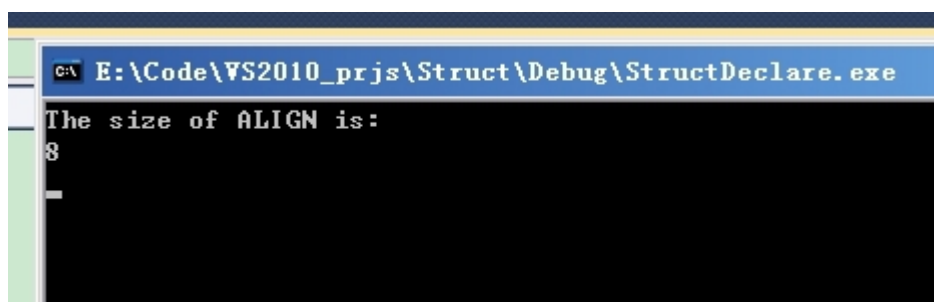
问题： 阅读下面一段代码并回答题目之后的问题：

```
1. struct ALIGN
2. {
3.     int mA;
4.     int mB;
5. };
```

请问在 32 位系统下 sizeof(ALIGN)的结果是多少？

当然这道题目是难不到广大程序员同学们滴！

在 32 位机器上 int 类型占 4 个字节，structALIGN 里面有 2 个 int 型变量，那么总共就是 8 个字节喽！Bingo，在这个例子中，sizeof(ALIGN)的结果的确是 8.



下面，我们把代码修改一下：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct ALIGN
6. {
7.     int mA;
8.     int mB;
9. };
10.
11. struct ALIGN1
12. {
13.     int mA;
14.     short mB;
```

```

15.};
16.
17.
18.int main()
19.{
20.    cout<<sizeof(short)<<endl;
21.    cout<<sizeof(ALIGN1)<<endl;
22.
23.    getchar();
24.    return 0;
25.}

```

请问输出是多少？

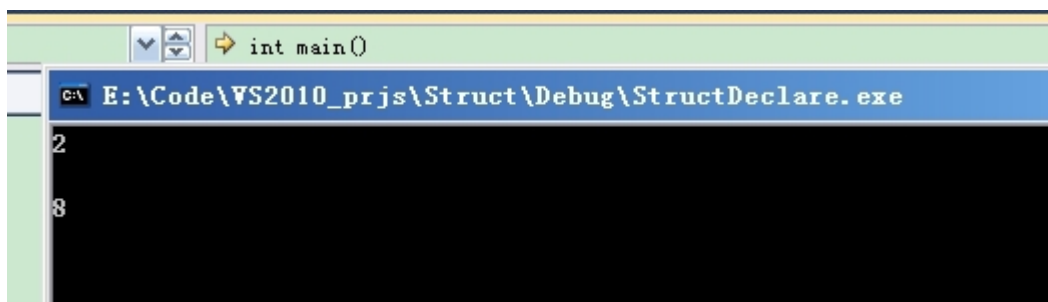
这还不简单，小 case 嘛！

mA 占 4 个字节，mB 占 2 个字节，所以 Struct ALIGN1 应该是 4+2=6 个字节，所以答案是 2 和 6。

---你确定么？（小丫的语言）

---我确定！

好的，请看大屏幕：



咦？结构体的大小不是将结构体元素单纯相加就可以的吗？怎么结果却变成 8 了呢？

要回答这个问题，需要从计算机的地址对齐讲起。至于为什么需要对齐，当然是对齐能够带来很多好处的。

第一，可以大大简化处理器和存储器之间接口的硬件设计；

第二，提高处理器访问数据的效率。

首先讲下，**对齐(alignment)**就是计算机系统对基本数据类型的可允许地址做了限制，某种类型的对象的地址必须是某个值 **k** 的倍数。

以 IA32 为例，在自然对齐方式下，基本数据类型（如 **short**，**int**，**double**）变量的地址必须被他们的大小整除。通俗的说，对于 **int** 类型的变量，因为宽度为 4，因此存放 **int** 类型变量的起始地址必须能被 4 整除，宽度为 2 的基本数据类型（**short** 等）则位于能被 2 整除的地址上，以此类推对于 **char** 和 **bool** 类型的变量，由于其只占用一个字节，则没有特别要求。



我们修改下程序，让其输出成员变量的地址：

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. struct ALIGN
6. {
7.     int mA;
8.     int mB;
9. };
10.
11. struct ALIGN1
12. {
13.     int mA;
14.     short mB;
15. };
16.
17.
18. int main()
19. {
20.     ALIGN aln0;
21.     ALIGN1 aln1;
22.
23.     cout<<"\n"<<&aln0.mA<<"\t"<<&aln0.mB<<endl;
24.     cout<<"\n"<<&aln1.mA<<"\t"<<&aln1.mB<<endl;
25.
26.     getchar();
27.     return 0;
28. }

```

程序调试我们可以看到：

自动窗口		
名称	值	类型
@aln0	0x0012ff5c {mA=0xffffffff mB=0xccc}	ALIGN *
@aln0.mA	0x0012ff5c	int *
@aln0.mB	0x0012ff60	int *
@aln1	0x0012ff4c {mA=0xffffffff mB=0xccc}	ALIGN1 *
@aln1.mA	0x0012ff4c	int *
@aln1.mB	0x0012ff50	short *
aln0	{mA=0xffffffff mB=0xffffffff }	ALIGN
aln1	{mA=0xffffffff mB=0xccc }	ALIGN1

从上述结果中，可以看出在 struct ALIGN1 中，int mA 的起始地址为 0x0012ff4c 可以被 4 整除，short mB 的起始地址为 0x0012ff50 可以被 2 整除。

再看下列代码：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct ALIGN
6. {
7.     int mA;
8.     int mB;
9. };
10.
11. struct ALIGN1
12. {
13.     int mA;
14.     short mB;
15. };
16.
17. struct ALIGN2
18. {
19.     char mA;
20.     int mB;
21.     short mC;
22. };
23.
24. struct ALIGN3
25. {
26.     int mB;
27.     char mA;
28.     short mC;
29. };
30.
31.
32. int main(void)
33. {
34.     ALIGN aln0;
35.     ALIGN1 aln1;
36.     ALIGN2 aln2;
37.     ALIGN3 aln3;
38.
39.     cout<<"The size of struct ALIGN is:"<<sizeof(ALIGN)<<endl;
40.     cout<<"\t"<<&aln0.mA<<"\t"<<&aln0.mB<<endl;
41.
42.     cout<<"The size of struct ALIGN1 is:"<<sizeof(ALIGN1)<<endl;
```

```

43.     cout<<"\t"<<&aln1.mA<<"\t"<<&aln1.mB<<endl;
44.
45.     cout<<"The size of struct ALIGN2 is:"<<sizeof(ALIGN2)<<endl;
46.     cout<<"\t"<<&aln2.mA<<"\t"<<&aln2.mB<<"\t"<<&aln2.mC<<endl;
47.
48.     cout<<"The size of struct ALIGN3 is:"<<sizeof(ALIGN3)<<endl;
49.     cout<<"\t"<<&aln3.mA<<"\t"<<&aln3.mB<<"\t"<<&aln3.mC<<endl;
50.
51.     getchar();
52.     return 0;
53.}

```

输出结果如下：

```

E:\Code\VS2010_prjs\Struct\Debug\StructDeclare.exe
The size of struct ALIGN is: 8
    0012FF58    0012FF5C
The size of struct ALIGN1 is:8
    0012FF48    0012FF4C
The size of struct ALIGN2 is: 12
    烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫z狂鸥  ↑    0012FF38
    0012FF3C
The size of struct ALIGN3 is: 8
    烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫z狂鸥  ↑
    0012FF24    0012FF2A

```

是不是觉得很奇怪？

ALIGN2 和 ALIGN3 都是 1 个 int 型，1 个 char 型，1 个 short 型，可是它们所占的空间却 1 个是 8，一个是 12？

这非常非常不科学啊！

2 个结构体都拥有一样的成员变量，可是所占的大小却有很大的区别。这一切的一切，是计算机中的幽灵在作祟，还是另有隐情？编译器如此厚此薄彼，到底是为什么？被偷去的内存，到底去了哪里？

下一篇我们将使用 gcc 编译，分析编译生成的每一步，了解编译器具体是怎么做的，为什么需要这么做，为你揭开这些谜团!!!

## 六、以空间换时间---Struct 中成员变量的对齐之道(下)

在上一篇[以空间换时间，Struct\(结构体\)中的成员对齐之道\(上\)](#)中，我们了解到 struct ALIGN2 和 struct ALIGN3 的成员变量都是 1 个 int 型，1 个 char 型及 1 个 short 型，可是它们所占的空间却 1 个是 8 字节，一个是 12 字节。

为什么会有这样的区别呢？

通过上篇关于对齐的介绍，我们已经猜测这是因为编译器对其作了对齐的处理所致，但是编译器处理的细节具体是什么呢？

### 6.1 编译过程详解

一般情况下，C 程序的编译过程为：

- ❖ 预处理
- ❖ 编译成汇编代码
- ❖ 汇编成目标代码
- ❖ 链接

这一篇我们将使用 gcc 对上述几个细节进行仔细分析，了解其处理细节：

源码如下：

```
1. /*****
   *****/
2. ** File: - Z:\code\c\Alignment\Align.c
3. **
4. ** Copyright (C), Long.Luo, All Rights Reserved!
5. **
6. ** Description:
7. **     Align.c --- To learn the details of Alignment by the compiler.
8. **
9. ** Version: 1.1
10.** Date created: 22:33:50,10/12/2012
11.** Author: Long.Luo
12.**
13.** ----- Revision History: -----
   --
14.** <author> <data>          <desc>
15.**
16. *****/
17.
18.#include <stdio.h>
19.
```

```

20. struct ALIGN2
21. {
22.     char mA;
23.     int mB;
24.     short mC;
25. };
26.
27. struct ALIGN3
28. {
29.     int mB;
30.     char mA;
31.     short mC;
32. };
33.
34.
35. int main(void)
36. {
37.     struct ALIGN2 aln2;
38.     struct ALIGN3 aln3;
39.
40.     printf("The size of struct ALIGN2 is: %d\n", sizeof(aln2));
41.     printf("\t aln2.mA=0x%x, aln2.mB=0x%x, aln2.mC=0x%x\n", &aln2.mA, &aln2.mB,
        &aln2.mC);
42.
43.     printf("The size of struct ALIGN3 is: %d\n", sizeof(aln3));
44.     printf("\t aln3.mA=0x%x, aln3.mB=0x%x, aln3.mC=0x%x\n", &aln3.mA, &aln3.mB,
        &aln3.mC);
45.
46.     return 0;
47. }

```

### 6.1.1 预处理部分

输出文件的后缀为: \*.cpp 文件。

```

root@long-desktop:/work/code/c/Alignment# gcc -E -o align.cpp Align.c
root@long-desktop:/work/code/c/Alignment# ll
total 52
drwxrwxrwx 2 root root 4096 2012-12-10 22:45 ./
drwxrwxrwx 3 nobody nogroup 4096 2012-12-10 22:25 ../
-rwxr--r-- 1 nobody nogroup 1116 2012-12-10 22:44 Align.c*
-rw-r--r-- 1 root root 38929 2012-12-10 22:45 align.cpp
root@long-desktop:/work/code/c/Alignment# wc align.cpp Align.c
1817 4506 38929 align.cpp
 48 117 1116 Align.c
1865 4623 40045 total
root@long-desktop:/work/code/c/Alignment#

```

### 6.1.2 编译成汇编代码

<1>. 使用-x 参数说明根据指定的步骤进行工作，cpp-output 指明从预处理得到的文件开始编译；

<2>. 使用-S 说明生成汇编代码后停止工作

**gcc -x cpp-output -S -o align.s align.cpp**

```
root@long-desktop:/work/code/c/Alignment# gcc -x cpp-output -S -o align.s align.cpp
Align.c: In function 'main':
Align.c:42: warning: format '%x' expects type 'unsigned int', but argument 2 has type 'char *'
Align.c:42: warning: format '%x' expects type 'unsigned int', but argument 3 has type 'int *'
Align.c:42: warning: format '%x' expects type 'unsigned int', but argument 4 has type 'short int *'
Align.c:45: warning: format '%x' expects type 'unsigned int', but argument 2 has type 'char *'
Align.c:45: warning: format '%x' expects type 'unsigned int', but argument 3 has type 'int *'
Align.c:45: warning: format '%x' expects type 'unsigned int', but argument 4 has type 'short int *'
root@long-desktop:/work/code/c/Alignment# ll
total 64
drwxrwxrwx 2 root root 4096 2012-12-10 23:08 ./
drwxrwxrwx 3 nobody nogroup 4096 2012-12-10 22:25 ../
-rw-r--r-- 1 root root 1093 2012-12-10 23:04 align1.s
-rwxr--r-- 1 nobody nogroup 1116 2012-12-10 22:44 Align.c*
-rw-r--r-- 1 root root 38929 2012-12-10 22:45 align.cpp
-rw-r--r-- 1 root root 1093 2012-12-10 23:10 align.s
-rw-r--r-- 1 root root 1093 2012-12-10 23:03 Align.s
root@long-desktop:/work/code/c/Alignment#
```

我们也可以直接编译到汇编代码

**gcc -S Align.c**

得到 align.s 文件之后，在最开始之处我们可以看到：

```
1. .file "Align.c"
2. .section .rodata
3. .align 4
```

其中的 .align 4 就表明了其后面所有的数据都遵守 4 字节对齐的限制。

### 6.1.3 编译成目标代码

汇编代码-->目标代码

```
root@long-desktop:/work/code/c/Alignment# gcc -x assembler -c align.s
root@long-desktop:/work/code/c/Alignment# ll
total 68
drwxrwxrwx 2 root root 4096 2012-12-10 23:28 ./
drwxrwxrwx 3 nobody nogroup 4096 2012-12-10 22:25 ../
-rw-r--r-- 1 root root 1093 2012-12-10 23:04 align1.s
-rwxr--r-- 1 nobody nogroup 1116 2012-12-10 22:44 Align.c*
-rw-r--r-- 1 root root 38929 2012-12-10 22:45 align.cpp
-rw-r--r-- 1 root root 1164 2012-12-10 23:28 align.o
-rw-r--r-- 1 root root 1093 2012-12-10 23:10 align.s
-rw-r--r-- 1 root root 1093 2012-12-10 23:03 Align.s
root@long-desktop:/work/code/c/Alignment#
```

### 6.1.4 编译成执行代码

目标代码-->执行代码

最终的输出结果如下所示：

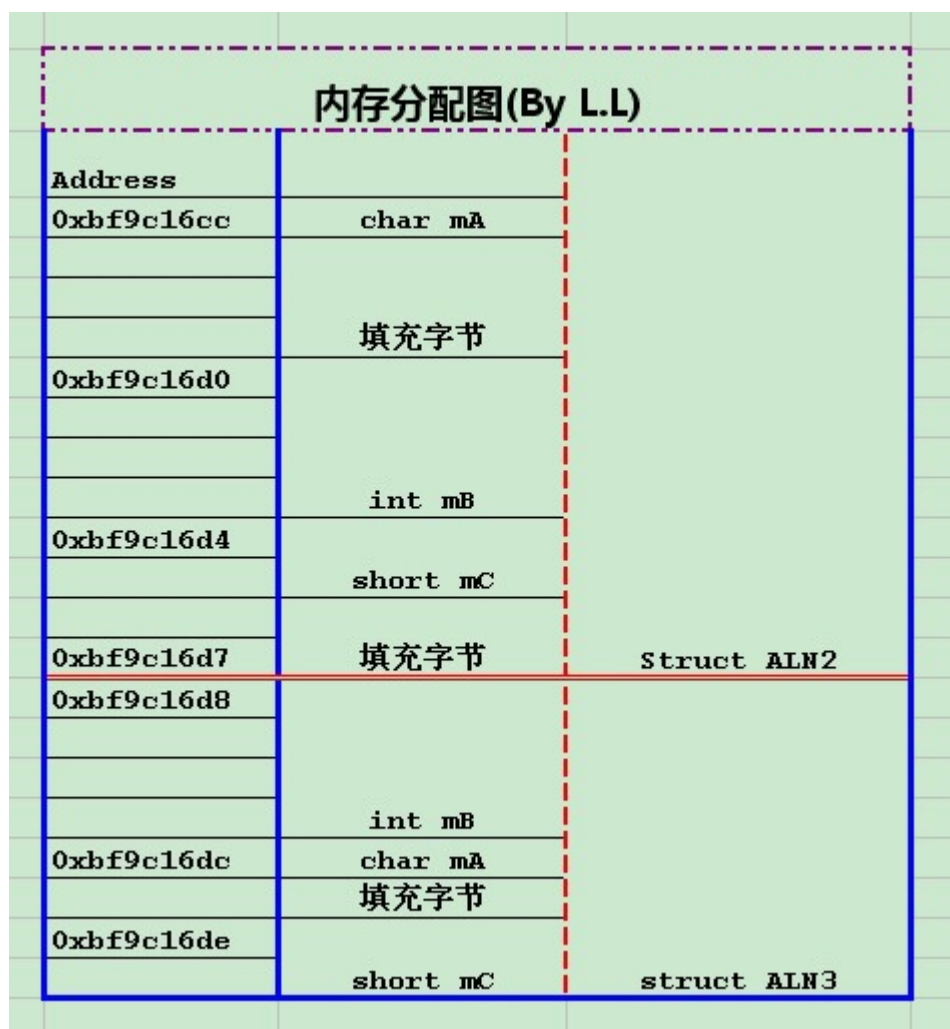
```

root@long-desktop:/work/code/c/Alignment# gcc -o align align.o
root@long-desktop:/work/code/c/Alignment# ls
align align1.s Align.c align.cpp align.o align.s Align.s
root@long-desktop:/work/code/c/Alignment# ./align
The size of struct ALIGN2 is: 12
  aln2.mA=0xbf9c16cc, aln2.mB=0xbf9c16d0, aln2.mC=0xbf9c16d4
The size of struct ALIGN3 is: 8
  aln3.mA=0xbf9c16dc, aln3.mB=0xbf9c16d8, aln3.mC=0xbf9c16de
root@long-desktop:/work/code/c/Alignment#

```

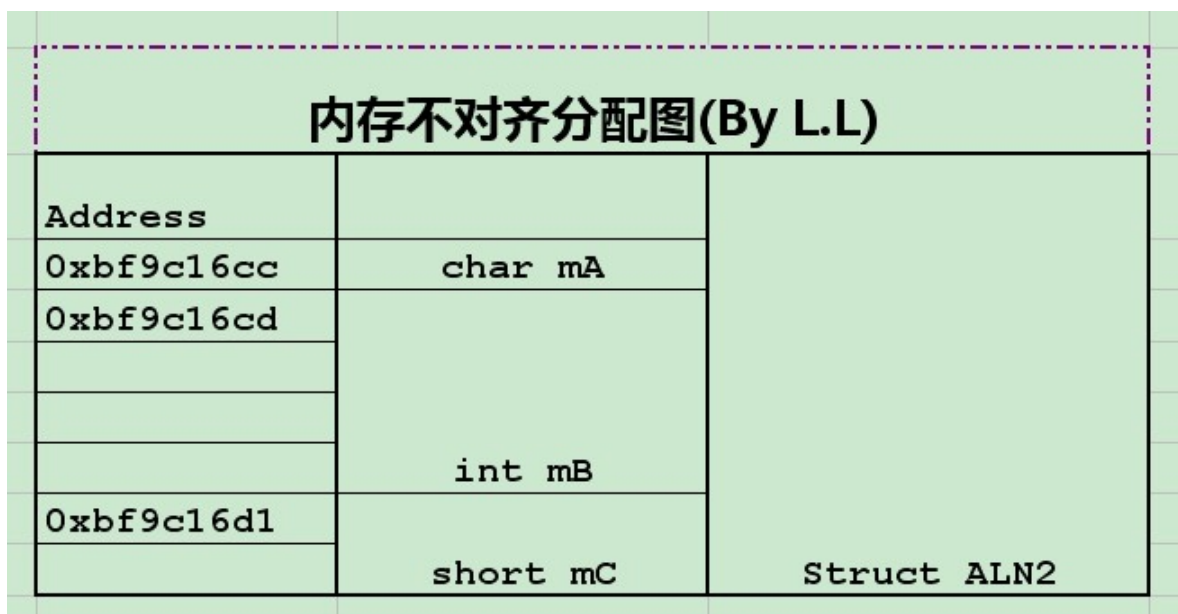
## 6.2 假如我们不对齐

我们可以绘出其内存分配图：



上图是内存对齐的 struct ALN2 和 struct ALN3 的内存分配情况，假如我们不对齐呢？其内存分配如下所示：





很明显，int mB 和 short mC 都不满足对齐要求。

### 6.2.1 对齐是为了换取效率

对齐可以提高取数据的效率

在 IA32 架构中，数据总线是 32 位，即一次可以存取 4 个字节的数据。

在对齐的情况下，struct ALN2 的每个成员都可以在一个指令周期内完成：

而假设我们的 struct ALN2 没有对齐，那么对于 struct ALN2 中 char mA，CPU 可以一次取出 4 个字节获得低位的一个字节，同时需要将高位的 3 个字节保存在寄存器中，之后的 int mB，CPU 必须再取得低位的 1 个字节并通之前保存在寄存器中的数据结果组合在一起，每一个都需要好几条指令，是不是相当麻烦？

## 6.3 如何自定义对齐

那肯定有同学要问了，有没有办法让处理器按照自己的要求进行地址对齐呢？

---当然可以，我们可以通过预编译命令 `#pragma pack(n)`，`n=1,2,4,8,16` 来改变这一系数，其中的 `n` 就是你要指定的“对齐系数”。

比如，我想让处理器按照 1 个字节的方式对齐，则代码如下：

```

/*****
***
1. ** File: - Z:\code\c\Alignment\AlignPackOne.c
2. **
3. ** Copyright (C), Long.Luo, All Rights Reserved!
4. **
5. ** Description:
6. **     Align.c --- To learn the details of Alignment by the compiler.

```



```

7. **
8. ** Version: 1.1
9. ** Date created: 23:39:05,10/12/2012
10.** Author: Long.Luo
11.**
12.** ----- Revision History: -----
    --
13.** <author> <data>          <desc>
14.**
15.*****/
16.
17.#include <stdio.h>
18.
19.
20.#pragma pack(1)
21.
22.struct ALIGN2
23.{
24.    char mA;
25.    int mB;
26.    short mC;
27.};
28.
29.struct ALIGN3
30.{
31.    int mB;
32.    char mA;
33.    short mC;
34.};
35.
36.
37.int main(void)
38.{
39.    struct ALIGN2 aln2;
40.    struct ALIGN3 aln3;
41.
42.    printf("The size of struct ALIGN2 is: %d\n", sizeof(aln2));
43.    printf("\t aln2.mA=0x%x, aln2.mB=0x%x, aln2.mC=0x%x\n", &aln2.mA, &aln2.mB,
        &aln2.mC);
44.
45.    printf("The size of struct ALIGN3 is: %d\n", sizeof(aln3));
46.    printf("\t aln3.mA=0x%x, aln3.mB=0x%x, aln3.mC=0x%x\n", &aln3.mA, &aln3.mB,
        &aln3.mC);

```

```
47.  
48.     return 0;  
49. }
```

编译之后输出结果如下：

```
root@long-desktop:/work/code/c/Alignment# ./AlignPackOne  
The size of struct ALIGN2 is: 7  
    aln2.mA=0xbfc700f9, aln2.mB=0xbfc700fa, aln2.mC=0xbfc700fe  
The size of struct ALIGN3 is: 7  
    aln3.mA=0xbfc700f6, aln3.mB=0xbfc700f2, aln3.mC=0xbfc700f7  
root@long-desktop:/work/code/c/Alignment#
```

可以看出，在我们要求的 1 字节对齐方式下，aln2 和 aln3 的结果都是 7，只占了 4+2+1 个字节，内存空间一个字节都利用到极致。

至此，关于内存对齐就此告一段落了，你弄明白了吗？

## 七、无即是有---没有成员变量的 Struct

在开始本篇之前，想问大家一个问题：

---0 是什么？

---呵呵，就是没有呗！

---那好，这 5 块钱拿去，就当抵我上次向你借的 500 块钱。

---什么？这哪和哪啊！这不一样

---可是你自己说的，0 就是“没有”。

---我说不清，反正不行，你必须还我 500。

0 是什么？起什么作用呢？为什么  $500 \neq 5$ ？

这节我们来讨论 0 的作用。例如，500 块钱，它后面 0 起到了什么作用呢？500 的 0，表示十和个位“没有”。虽说“没有”，但这个 0 却不能省略。因为如果省略了 0，一件 500 块的衣服，你只给 5 块，小心遭到暴打。

那原因是什么呢？

在按位计数法中，数位具有很重要的意义。即使十位的数“没有”，也不能不写数字。这时就轮到 0 出场了，即 0 的作用就是占位。换言之，0 占着一个位置以保证数位高于它的数字不会产生错位。

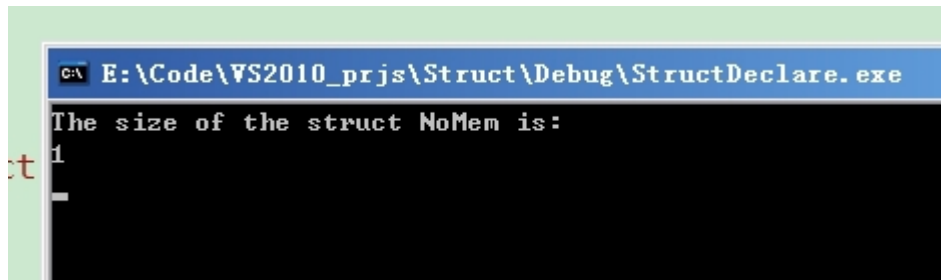
正因为有了表示“没有”的 0，数值才能正确地表现出来。可以说在按位计数法中 0 是不可或缺的。

打住，这和我们讲的 struct 有什么关系？

当然有关系了，请问下面这段代码输出的是什么呢？

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct NoMember
6. {
7.
8. };
9.
10. int main(void)
11. {
12.     cout<<"The size of the struct NoMem is:"<<endl;
13.     cout<<sizeof(NoMember)<<endl;
14.
15.     getchar();
16.     return 0;
17. }
```

是 0 呢？还是 1？2？3？想必大部分人还是说不出的，那我们先看看输出结果：



一个没有任何变量的 **Struct** 居然占了 1 个字节的空间。

这不科学呀！

那这是为什么呢？

不急，让子弹先飞一会儿.....

再看下面一段代码：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct NoMember
6. {
7.
8. };
9.
10. struct NoMember1
11. {
12.     int mA[0];
13. };
14.
15. int main(void)
16. {
17.     //cout<<endl<<"The size of the struct NoMember is: "<<sizeof(NoMember)<<endl;
18.     cout<<endl<<"The size of the struct NoMember1 is: "<<sizeof(NoMember1)<<endl;
19.
20.     getchar();
21.     return 0;
22. }
```

一个结构体里面定义了一个空的数组，那这次的输出会是什么呢？

厄， 这还是一个空的结构体，所以输出还是 1 吧？

好的，你还真蒙对了，请看结果：



```
C:\ E:\Code\VS2010_prjs\Struct\Debug\StructDeclare.exe

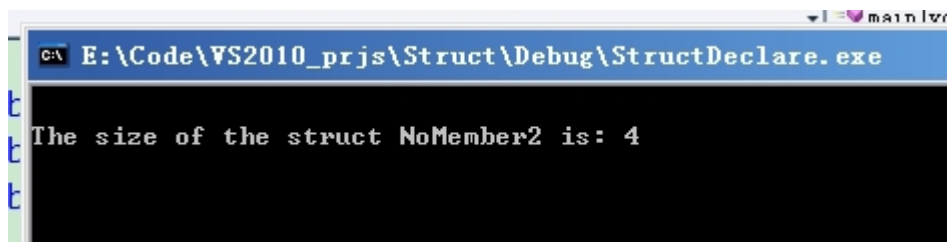
The size of the struct NoMember1 is: 1
```

但是，如果 struct 里面还有另外的变量呢？会是什么情况呢？

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct NoMember
6. {
7.
8. };
9.
10. struct NoMember1
11. {
12.     int mA[0];
13. };
14.
15. struct NoMember2
16. {
17.     char mB;
18.     int mA[0];
19. };
20.
21. int main(void)
22. {
23.     //cout<<endl<<"The size of the struct NoMember is: "<<sizeof(NoMember)<<endl;
24.     //cout<<endl<<"The size of the struct NoMember1 is: "<<sizeof(NoMember1)<<endl;
25.     cout<<endl<<"The size of the struct NoMember2 is: "<<sizeof(NoMember2)<<endl;
26.
27.     getchar();
28.     return 0;
29. }
```

这一次的输出是什么呢？

char mB 占据 4 个字节，然后 int mA[0]占据 1 个字节，所以结果应该是 5 吧？



但是结果是 4，为什么呢？

其实结合  $500 \neq 5$  的话也很好理解，虽然结构体包含 0 个成员变量，但是结构体起到“占位”的作用，“空结构体”变量必须被存储，编译器为其分配一个字节的空間用于占位了。这样一来，不光可以取地址，两个不同的“空结构体”变量又可以得以区分。

我们可以输出各个结构体的地址：

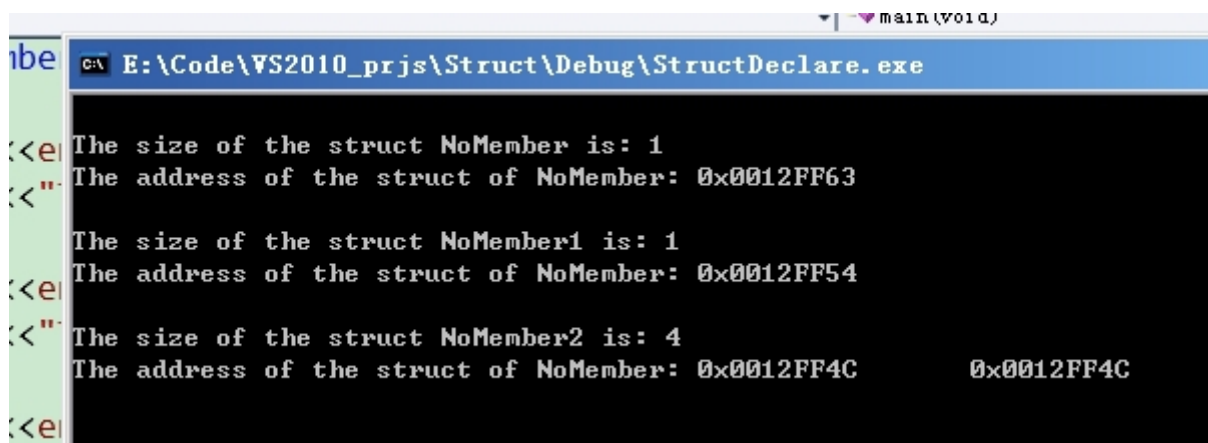
```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct NoMember
6. {
7.
8. };
9.
10. struct NoMember1
11. {
12.     int mA[0];
13. };
14.
15. struct NoMember2
16. {
17.     char mB;
18.     int mA[0];
19. };
20.
21. int main(void)
22. {
23.     NoMember nm0;
24.     NoMember1 nm1;
25.     NoMember2 nm2;
26.
27.     cout<<endl<<"The size of the struct NoMember is: "<<sizeof(NoMember)<<endl;
28.     cout<<"The address of the struct of NoMember: 0x"<<&nm0<<endl;
29.
```

```

30.     cout<<endl<<"The size of the struct NoMember1 is: "<<sizeof(NoMember1)<<endl;
31.     cout<<"The address of the struct of NoMember: 0x"<<&nm1.mA<<endl;
32.
33.     cout<<endl<<"The size of the struct NoMember2 is: "<<sizeof(NoMember2)<<endl;
34.     cout<<"The address of the struct of NoMember: 0x"<<&nm2.mA<<"\t0x"<<&nm2.mA
    <<endl;
35.
36.     getchar();
37.     return 0;
38. }

```

输出结果如下：



```

C:\E:\Code\VS2010_prjs\Struct\Debug\StructDeclare.exe
The size of the struct NoMember is: 1
The address of the struct of NoMember: 0x0012FF63

The size of the struct NoMember1 is: 1
The address of the struct of NoMember: 0x0012FF54

The size of the struct NoMember2 is: 4
The address of the struct of NoMember: 0x0012FF4C      0x0012FF4C

```

注意下面这份简谱，注意到乐谱上也有很多的休止符，在简谱上也是用 0 来表示，但是它们不是“没有”，而是表示不发音！

## 我的未来不是梦

(大众乐谱网制谱)

陈家麓 词  
翁孝良 曲

1=G 4/4

<p>0 2 3 3 3 3 2. 2 2 3 2   2 1 1 1 - -   0 1 6 1 6 6 3 3 3 2 1  </p> <p>你 是 不 是 像 我 在 太 阳 下 着 低 头 求 流 着 汗 水 默 默 辛 苦 地</p> <p>你 是 不 是 像 我 整 天 忙 着 追 追 求 一 种 意 想 不 到 的</p>	<p>2 3 2 2 - -   0 3 5 5 5 5 6 3 3 2 3 2   2 1 1 1 - -  </p> <p>工 作 柔 你 是 不 是 像 我 就 算 受 茫 了 然 冷 失 落 措</p>
---	---

这些 0 都不能去掉的，因为去掉，节奏肯定乱了。

0 与其说是“空”，还不如说是“填空”更恰当。因为它的作用是占位。

结合这个例子，想必大家就理解了，“空结构体”为什么需要占用一个字节的空間。

可见，科学和艺术不分家的啊！

对数组比较熟悉的同学可能会问，怎么可以有 0 元素数组呢？

是的，如果我们在 struct 和 class 之后定义 0 元素数组，会编译报错，如下代码所示：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct NoMember
6. {
7.
8. };
9.
10. struct NoMember1
11. {
12.     int mA[0];
13. };
14.
15. struct NoMember2
16. {
17.     char mB;
18.     int mA[0];
19. };
20.
21.
22. struct NoMember3
23. {
24.     int arr[0];
25. };
26.
27.
28. int arr[0];
29.
30.
31. int main(void)
32. {
33.     NoMember nm0;
34.     NoMember1 nm1;
35.     NoMember2 nm2;
36.
37.     cout<<endl<<"The size of the struct NoMember is: "<<sizeof(NoMember)<<endl;
```



```

38.     cout<<"The address of the struct of NoMember: 0x"<<&nm0<<endl;
39.
40.     cout<<endl<<"The size of the struct NoMember1 is: "<<sizeof(NoMember1)<<endl;
41.     cout<<"The address of the struct of NoMember: 0x"<<&nm1.mA<<endl;
42.
43.     cout<<endl<<"The size of the struct NoMember2 is: "<<sizeof(NoMember2)<<endl;
44.     cout<<"The address of the struct of NoMember: 0x"<<&nm2.mA<<"\t0x"<<&nm2.mA<<endl;
45.
46.     getchar();
47.     return 0;
48. }

```

出错信息如下:

```

1. 1>e:\code\vs2010_prjs\struct\structdeclare\nomems.cpp(28): error C2466: 不能分配
   常量大小为 0 的数组

```

而 Struct 里面的 0 元素数组报的是 warning, 如下所示:

```

1. 1>e:\code\vs2010_prjs\struct\structdeclare\nomems.cpp(24): warning C4200: 使用了
   非标准扩展 : 结构/联合中的零大小数组
2. 1>          当 UDT 包含大小为零的数组时, 无法生成复制构造函数或副本赋值运算符

```

那么, 问题又来了, 这些 struct 里面的 0 元素数组有什么意义呢?

## 八、Struct 中 0 元素数组的意义

上一回我们在[无既是有---没有成员变量的 Struct（结构体）](#) 文章的结尾留了一个悬念：

---为什么 0 元素数组在 class 和 struct 结构体之外定义就是错误的，而在 class 和 struct 中定义就是 Okay 的，那么结构体中的 0 元素数组意义何在？

打个通俗的比喻，比如一个部门，有部门经理、PM、以及数量众多的苦逼程序猿们，某天部门接到一个项目，于是乎，拉出一个 PM 及数量未知的程序猿们，开工了：

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. struct Department
6. {
7.     int Manager;
8.     int PM;
9.     int ProgrammerNo[0];
10.};
11.
12.
13.int main(void)
14.{
15.    Department *pt = NULL;
16.    int Num = 5;
17.    pt = (Department *)malloc(sizeof(Department) + sizeof(int) * Num);
18.
19.    pt->ProgrammerNo[0] = 006;
20.    pt->ProgrammerNo[1] = 99;
21.
22.
23.    getchar();
24.    return 0;
25.}
```

做一个项目，程序猿的数量是变化的，因此我们也就知道

**struct 中 0 元素数组意义就是可以作数量未知的扩展，又称柔性数组。**

## 结束语

本文档对 C/C++使用的 **Structure**(结构体)做了一番仔细的分析,并对其中的很多细节如对齐、域、成员变量等进行了重点剖析,同时结合很多代码实例,用通俗易懂的语言描述以便于大家的理解。

希望本文档能够对大家有所帮助,如有错误或者不当之处,敬请不吝指正!谢谢!这是详解 **Struct**(结构体)系列的第一部分,在接下来的时间里我会完成第二部分,敬请期待。

*罗龙于 2013.12.23 @Shenzhen*