



## SIZEOF

一般的参考书都对 `sizeof` 一带而过，不过在实践中，对其考究一下对深入理解机器行为大有好处。

### Concept

`sizeof` 并不是函数而是 C 语言的一种单目操作符，如 C 语言的其他操作符 `++`、`--` 等。它以字节形式给出其操作数的存储大小，操作数可以是一个表达式或一个数据类型名。

`sizeof` 通常在编译时刻进行计算，所以它可以被看作是一个常量表达式。C99 中规定 `sizeof` 也可以在运行时计算，这就意味着 `sizeof` 也可以用来计算动态数组的大小。可惜的是并不是所有 C 编译器都很好遵循 C99，所以，考虑到可移植性，最好还是不要使用 `sizeof` 这个理论上合理的特性。

如果你使用表达式操作数，表达式本身并不会被计算。编译器仅会对表达式类型结果进行确定。因此你也不必担心表达式中的任何 side effect。`sizeof` 也可以对一个函数调用求值，其结果是函数返回类型的大小，函数并不会被调用。

### Usage

`Sizeof` 可以有两种用法：

1. 用于数据类型：`sizeof (type)`

数据类型必须用括号括住，如 `sizeof (int)`。而 `sizeof int` 的使用则是错误的。

2. 用于变量：`sizeof (var_name)` 或 `sizeof var_name`

变量名可以不用括号括住。如 `sizeof (var_name)`，`sizeof var_name` 等都是正确形式。带括号的用法更普遍，大多数程序员采用这种形式。应该清楚，虽然带了个括号但也仅仅是看起来像个函数而已。

所以下面三种表示都是正确的：

```
int i;
sizeof( i );      // ok
sizeof i;         // ok
sizeof( int );   // ok
sizeof int;       // error
```

### Examination

先测试一下，看看小小 `sizeof` 能有什么名堂。：)

```
void primitive(void)
{
    char c = 'A';
```



```
printf("sizeof(c)      is [%d]\n", sizeof c);
printf("sizeof(char)   is [%d]\n", sizeof(char));
printf("sizeof(short)  is [%d]\n", sizeof(short));
printf("sizeof(int)    is [%d]\n", sizeof(int));
printf("sizeof(double) is [%d]\n", sizeof(double));
printf("sizeof(long double) is [%d]\n", sizeof(long double));

return;
}

int add(int *p)
{
    return (*p = 3);
}

void sideeffect(void)
{
    int i = 0;

    printf("sizeof(3+4)      is [%d]\n", sizeof(3+4));
    printf("sizeof(3+4.0)    is [%d]\n", sizeof(3+4.0));

    printf("sizeof(++i+i++)   is [%d]\n", sizeof(++i+i++));
    printf("sizeof(sizeof(++i+i++)) is [%d]\n", sizeof(sizeof(++i+i++)));
    printf("i                is [%d]\n", i);
    printf("sizeof(add())     is [%d]\n", sizeof(add(&i)));
    printf("i                is [%d]\n", i);
}

void arrayp(char arrp[10])
{
    printf("sizeof(arrp)      is [%d]\n", sizeof(arrp));
    return;
}

void array(void)
{
    char (*pa)[10], arr[10] = {5};

    pa = arr; /* pa = &arr; */

    printf("sizeof(arr)      is [%d]\n", sizeof(arr));
    printf("*(arr+1) is [%d], *(&arr+1) is [%d]\n", arr[0], *(&arr+1));

    return;
}

typedef struct
{
    char    c1;
    double  d;
    char    c2;
}cdc_t;

typedef struct
{
```



```

}empty_t;

typedef struct
{
    char    c;
    char    ca[];
}ca_t;

void stru(void)
{
    cdc_t    t;
    printf("sizeof(cdc_t)      is [%d]\n", sizeof(cdc_t));
    printf("sizeof(&t)         is [%d]\n", sizeof(&t));
    printf("sizeof(empty_t)    is [%d]\n", sizeof(empty_t));
    printf("sizeof(ca_t)       is [%d]\n", sizeof(ca_t));
    return;
}

int main()
{
    primitive();
    sideeffect();
    array();
    stru();

    return 0;
}

```

上述代码在 Red Hat Linux 8.0 3.2-7 及 Xscale 上运行结果如下:

```

/* run in gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7) */
<yug>[/mnt/hgfs/share]%gcc sizeof.c
<yug>[/mnt/hgfs/share]%a.out
sizeof.c: In function `array':
sizeof.c:45: warning: assignment from incompatible pointer type
sizeof(c)          is [1]
sizeof(char)       is [1]
sizeof(short)      is [2]
sizeof(int)        is [4]
sizeof(double)     is [8]
sizeof(long double) is [12]
sizeof(3+4)        is [4]
sizeof(3+4.0)       is [8]
sizeof(++i+i++)     is [4]
sizeof(sizeof(++i+i++)) is [4]
i                  is [0]
sizeof(add())       is [4]
i                  is [0]
sizeof(arr)         is [10]
*(arr+1) is [5], *(&arr+1) is [-1073744166]
sizeof(cdc_t)       is [16]
sizeof(empty_t)     is [0/1] /* gcc/g++下结果分别为 0/1 */
sizeof(ca_t)        is [1]

```

以下为在 Linux 下使用 -malign-double 选项及在 Xscale 下运行与上面不同的结果, 相同的没有摘录。

```

<yug>[/mnt/hgfs/share]%gcc -malign-double sizeof.c
<yug>[/mnt/hgfs/share]%a.out
*(arr+1) is [5], *(&arr+1) is [-1073744166]
sizeof(cdc_t)       is [24]

```



```

/* run in Xscale */
sizeof(double)          is [8]
sizeof(long double)     is [8]
*(arr+1) is [5], *(&arr+1) is [23623934]
sizeof(cdc_t)           is [16]

```

## Details

## Primitive Data Type's sizeof

### Char

ANSI C 正式规定字符类型为 1 字节。包括 char、unsigned char 或 signed char。

```

char c = 'A';
printf("sizeof( c ) = %d\n", sizeof( c )); // sizeof( c ) = 1 in all compiler use the type char

```

不同编译器对此处理可能也会稍有差别。如，你可以用下面的 sizeof 的特性来区别你所用的编译器是 C 编译器还是 C++ 编译器。

```

printf("sizeof( 'A' ) = %d\n", sizeof( 'A' )); // sizeof( 'A' ) = 4 in C compiler
                                                // sizeof( 'A' ) = 1 in C++ compiler

```

当然你完全可以用宏 \_\_cplusplus 来达到同样的目的。

### Other primitive data type

int、unsigned int、short int、unsigned short、long int、unsigned long、float、double、long double 类型的 sizeof 在 ANSI C 中没有具体规定。一般的，在 32 位编译环境中，sizeof(int) 的取值为 4。

Intel 由于是从 16 位体系结构扩展成 32 位的，所以其用术语“字 (word)”表示 16 位数据类型。因此称 32 位数为“双字 (double words)”，称 64 位数为“四字 (quad words)”。

C declaration	Intel Data Type	GAS suffix	Size (Bytes)
char	Byte	b	1
short	Word	w	2
int	Double Word	l	4
unsigned	Double Word	l	4
long int	Double Word	l	4
unsigned long	Double Word	l	4
char *	Double Word	l	4
float	Single Precision	s	4
double	Double Precision	l	8
long double	Extended Precision	t	10/12

Figure1: Sizes of standard data types



其中，GCC 用数据类型 `long double` 来表示扩展精度的浮点值。为了提高存储系统的性能，它将这样的浮点数存储成 12 字节数。虽然 ANSI C 标准包括 `long double` 数据类型，但是对大多数编译器和机器组合来说，它的实现和普通 `double` 的 8 字节格式是一样的。对 GCC 和 IA32 的组合来说，支持扩展精度是很少见的。

Microsoft Windows 对对齐的要求更严格，任何 `k` 字节(基本)对象的地址都必须是 `k` 的倍数。特别地，它要求一个 `double` 数的地址应该是 8 的倍数。这种要求提高了存储器性能，代价是浪费了一些空间。Linux 中的设计策略是除了 `char, short` 类型，其它类型(如 `int, int*, float, double` 等)的地址必须是 4 的倍数。但使用 `-malign-double` 选项会使 GCC 为 `double` 类型数据使用 8 字节对齐。这会提高存储器性能，但是在与用 4 字节对齐方式下编译的库代码链接时，会导致不兼容。

## Point variable's sizeof

指针用来记录了另一个对象的地址。既然是来存放地址的，那么它当然等于计算机内部地址总线的宽度。所以在 32 位计算机中，一个指针变量的返回值必定是 4（注意结果是以字节为单位），可以预计，在将来的 64 位系统中指针变量的 `sizeof` 结果为 8。

```
char    *pc = "abc";
int     *pi;
string  *ps;
char    **ppc = &pc;
void ( *pf ) ();    // 函数指针
```

```
sizeof( pc );        // 结果为 4
sizeof( pi );        // 结果为 4
sizeof( ps );        // 结果为 4
sizeof( ppc );       // 结果为 4
sizeof( pf );        // 结果为 4
```

## Array's sizeof

数组的 `sizeof` 值等于数组所占用的内存字节数，而对作为参数的数组名做 `sizeof`，却相当对指针的 `sizeof` 而不是整个数组的大小。

```
char    a1[] = "abc";
int     a2[3];

sizeof( a1 );        // 结果为 4，若对此字符串取 strlen 该是多少？
sizeof( a2 );        // 结果为 3*4=12（依赖于 int）

void foo3(char a3[3])
{
    int c3 = sizeof( a3 ); // c3 =4
}
void foo4(char a4[])
{
    int c4 = sizeof( a4 ); // c4 =4
}
```



数组看起来非常简单，但由于其与指针的复杂关系，还是需要深入理解才不会出错。下面顺便说几句题外话，讲一下数组名及指针的关系。pc-lint 的 545 错误号对此做出了详细描述。

```
int a[10], (*p)[10];
p = a;    //一般应告警
p = &a;   //不会告警
```

此处告警如前面例子中的 [Warning](#) 一样，分析一下为什么。数组指针 `p` 的含义是：首先 `p` 是一个指针。其次，它指向的是一块 10 个 `int` 类型数据大小的空间的一个指针。数组名 `a` 代表的是该数组的首地址，相当于 `&a[0]`，仅仅相当于一个指向一个 `int` 类型空间的指针。所以将其赋值给 `p`，编译器通常会告警类型不匹配。那么再看 `&a` 的含义，`&a` 代表的是一个有 10 个 `int` 型数据空间的地址，可见是和指针 `p` 的含义相同。所以 `p = &a;` 不会产生告警。此时也应该清楚 `a+1` 与 `&a+1` 可是大不相同的。

指针和数组在大多数情况下可以互换使用，但在细微之处还有很多不同。在其他文中再进一步说明，此处不再赘述。

## Struct's sizeof

首次考虑 Struct's sizeof 一定会让人感到费解的。主要是因为结构的 sizeof 涉及到字节对齐的一些规则。要充分理解结构的 sizeof，最好首先了解一下结构在内存中分配的对齐要求。

- 1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；
- 2) 结构体每个成员相对于结构体首地址的偏移量 (offset) 都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节 (internal adding)；
- 3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在最末一个成员之后加上填充字节 (trailing padding)。

如前面例子所示 `struct cdc_t`，`sizeof(cdc_t)=16`，而不是简单地 `sizeof(char) + sizeof(double) + sizeof(char)=10`。并且对于使用了 `-malign-double` 选项的 GCC 编译器，`sizeof(cdc_t)=24`。

对于上面的结构对齐原则，除了用于理解结构对齐，在实际中我们也可以灵活地加以应用。如下面代码：

```
#define list_entry(ptr, type, member) \
    ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
page = list_entry(entry, struct page, list);
```

扩展后，我们便可以得到如下代码：

```
page = ((struct page *)((char *)(entry)-(unsigned long)(&((struct page *)0)->list)))
```

这里的 `entry` 是一个 `page` 结构内部的成分 `list` 的在使用的地址，当我们需要那个 `page` 结构本身的地址，所以要 `entry` 减去一个位移量，即成分 `list` 在 `page` 结构内部的位移量，才能达到要求。那么，这位移量到底是多少呢？`&((struct page *)0)->list` 就表示当结构 `page` 正好在地址 0 上时其成分 `list` 的地址，也就是我们上面规则 2) 中的位移量。这样，我们就得到了我们想要的结果。

提到对齐，就不得不说一说 GCC 保留字 `attribute` 与 `align` 及 `pack`。关于这个话题，会在其他文进一步阐述。此处至少应该知道在考虑结构的 sizeof 时，应该考虑到结构的 `align` 或 `pack`。



## Union's sizeof

结构体在内存组织上是顺序式的，联合体则是重叠式，各成员共享一段内存，联合体首地址必须与联合中最大基本成员对齐，也就是说联合的首地址必须能被其最大的基本类型数据大小整除。所以整个联合体的 sizeof 也就是每个成员 sizeof 的最大值。结构体的成员也可以是复合类型，这里，复合类型成员是被作为整体考虑的。

所以，下面例子中，U 的 sizeof 值等于 sizeof(s)。

```
union U
{
    int      i;
    char     c;
    cdc_t    s;
};
```

## Bit-fields' sizeof

位域成员不能单独被取 sizeof 值，我们这里要讨论的是含有位域的结构体的 sizeof，只是考虑到其特殊性而将其专门列了出来。

C99 规定 int、unsigned int 和 bool 可以作为位域类型，但编译器几乎都对此作了扩展，允许其它类型类型的存在。

位域的使用规则：

- ① 通常，位域成员的类型须指定为 unsigned int 型
- ② 一个位域必须存储在同一存储单元中，不能跨两个单元
- ③ 位域的长度不能大于存储单元的长度，也不能定义位域数组
- ④ 长度为 0 的位域，其作用是使下一个位域从下一个存储单元开始存放
- ⑤ 可以定义无名位域
- ⑥ 位域可以用整型格式符输出
- ⑦ 位域可以在数值表达式中引用，它会被系统自动地转换成整形数

使用位域的主要目的是压缩存储，其大致规则为：

- 1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的 sizeof 大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；
- 2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的 sizeof 大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；
- 3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6 采取不压缩方式，Dev-C++ 采取压缩方式；
- 4) 如果位域字段之间穿插着非位域字段，则不进行压缩；

下面来看看例子：

```
struct BF1
{
```



```
char    f1 : 3;
char    f2 : 4;
char    f3 : 5;
};
```

位域类型为 char，第 1 个字节仅能容纳下 f1 和 f2，所以 f2 被压缩到第 1 个字节中，而 f3 只能从下一个字节开始。因此 sizeof(BF1)的结果为 2。

```
struct BF2
{
    char    f1 : 3;
    short   f2 : 4;
    char    f3 : 5;
};
```

由于相邻位域类型不同，在 VC6 中其 sizeof 为 6，在 Dev-C++中为 2。

```
struct BF3
{
    char    f1 : 3;
    char    f2;
    char    f3 : 5;
};
```

非位域字段穿插在其中，不会产生压缩，在 VC6 和 Dev-C++中得到的大小均为 3。

## Attention

使用 sizeof 运算符，我们须注意如下事项：

- ① sizeof 运算不会存在结果为 0 的运算，即使是一个空的类
- ② sizeof 不能用在下列操作数上：
  - 返回值为 void 的函数
  - void 类型
  - 位域
  - 动态数组
  - 不完整类型
  - 未定义类
- ③ 当 sizeof 应用于一个对象的引用时，结果相当于 sizeof 作用于对象本身
- ④ 如果结构最末存在没有大小的数组时，结构的 sizeof 相当于并不包含数组的结构的大小

## Application

通常，sizeof 常用在下列情形中：

- ① 表示一个对象的大小时，使用 sizeof 而不是固定数值，是编写在不同机器类型上可移植代码的一个良好习惯。
- ② 用来计算数组成员个数：sizeof array / sizeof array[0]





## Reference

sizeof, 终极无惑 Englep @ 2004-07-30 23:48  
The C Book <http://publications.gbdirect.co.uk/>  
Sizeof Operator <http://docs.hp.com/>

欲练神功