《C 和指针》

《C 专家编程》

《C 陷阱与缺陷》

《C 语言编程要点》

《编程精粹--Microsoft 编写优质无错 C 程序秘诀》

总　结

说明：总结的知识点主要源于上面的 4 本书，《编程精粹--Microsoft 编写优质无错 C 程序秘诀》这本书未做总结，该书有清晰版的 pdf 格式的电子版。

指针和数组相关概念
****
字符与字符串的区别
指针与数组 1
指针与数组 2
指针和数组的相同与不同
用 malloc 为字符串分配存储空间时的注意事项
作为常数的数组声明(c 缺陷与陷阱 3.3 节.在其它部分有包含该节的知识点，了解 or 略过)
字符串常量
用字符串常量初始化指针和数组
二维数组下标操作的相关概念
指向一维、二维数组的指针
array_name 和&array_name 的异同
数组作为函数的参数时，不能通过 sizeof 运算符得到该数组的大小
用 strlen()求字符串的长度
‘char **’ 和 ‘const char **’的兼容性问题
空指针相关的问题
NULL 和 NUL 的区别
未初始化的指针和 NULL 指针的区别
理解函数的声明
函数参数的传值调用
函数指针
作为函数参数的多维数组
强制类型转换相关概念
可变参数相关问题
malloc()、calloc()、realloc()
在程序退出 main()函数之后，还有可能执行一部分代码吗?
总线错误和段错误相关概念

数字和字符串之间转换相关的函数
****
怎样判断一个字符是数字、字母或其它类别的符号?
怎样将数字转换为字符串?
怎样将字符串转换为数字?

字符串以及内存操作相关函数
****
字符串拷贝和内存拷贝函数:
strcpy
strncpy
memcpy
memmove
memccpy
bcopy

字符串和内存数据比较函数:
strcmp
strcasecmp
strncasecmp
memcmp
strcoll
bcmp

连接字符串的函数:
strcat
strncat

查找字符/字符串的函数:
strstr
strchr
strrchr
memchr

其它相关的函数:
index
rindex
strlen
strdup
memset
bzero
strspn
strcspn
strpbrk
strtok

数据结构及算法相关函数
qsort（）
bsearch()
lsearch（线性搜索）
lfind（线性搜索）
srand（设置随机数种子）
rand（产生随机数）

OTHER
****
什么是标准预定义宏?
断言 assert(表达式) 相关概念
连接运算符“##”和字符串化运算符"#"有什么作用?
注释掉一段代码的方法
Typedef 相关概念
= 不同于 ==
词法分析中的“贪心法”
运算符的优先级问题

变量的存储类型及初始化相关概念
左值和右值相关的概念
变量的值和类型相关的概念
怎样删去字符串尾部的空格?
怎样删去字符串头部的空格?
怎样打印字符串的一部分?
结构的自引用
结构的存储分配
边界计算与不对称边界
整数溢出
返回整数的 getchar 函数
更新顺序文件
随机数的相关概念
用递归和迭代两种办法解 fibonacci

字符与字符串的区别(c 缺陷与陷阱 1.5 节)

C 语言中的单引号和双引号含义迥异，在某些情况下如果把两者弄混，编译器并不会检测报错，从而在运行时产生难以预料的结果。

用单引号引起的一个字符实际上代表一个整数，整数值对应于该字符在编译器采用的字符集中的序列值。因此，对于采用 ASCII 字符集的编译器而言，'a'的含义与 0141（八进制）或者 97（十进制）严格一致。

用双引号引起的字符串，代表的却是一个指向无名数组起始字符的指针，该数组被双引号之间的字符以及一个额外的二进制值为零的字符'\0'初始化。

下面的这个语句：

```
printf ("Hello world\n");
```

与

```
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ',
                'w', 'o', 'r', 'l', 'd', '\n', 0};
printf (hello);
```

是等效的。

整型数（一般为 16 位或 32 位）的存储空间可以容纳多个字符（一般为 8 位），因此有的 C 编译器允许在一个字符常量（以及字符串常量）中包括多个字符。也就是说，用'yes'代替"yes"不会被该编译器检测到。后者（即"yes"）的含义是“依次包含'y'、'e'、's'以及空字符'\0'的 4 个连续内存单元的首地址”。前者（即'yes'）的含义并没有准确地进行定义，但大多数 C 编译器理解为，“一个整数值，由'y'、'e'、's'所代表的整数值按照特定编译器实现中定义的方式组合得到”。因此，这两者如果在数值上有什么相似之处，也完全是一种巧合而已。

> 译注：在 Borland C++ v5.5 和 LCC v3.6 中采取的做法是，忽略多余的字符，最后的整数值即第一个字符的整数值；而在 Visual C++ 6.0 和 GCC v2.95 中采取的做法是，依次用后一个字符覆盖前一个字符，最后得到的整数值即最后一个字符的整数值。

```
#include <stdio.h>
int main()
{
    char ch = 'abcdefghijklmnopqrstuvwxyz';
    char str[] = "abcdefghijklmnopqrstuvwxyz";
    printf("-----%c-----\n%s\n",ch, str );
    return 0;
}
```

编译该程序可以通过，但是会产生警告；输出结过为：

```
-----z-----
abcdefghijklmnopqrstuvwxyz
```

// 在 Dev-C++ 4.9.9.2 编译环境中可以通过，但是在 VC6.0 中通不过

指针与数组 1（c 缺陷与陷阱 3.1 节）

c 语言中的数组值得注意的地方有以下两点：

1、*c 语言中只有一维数组*，而且数组的大小必须在编译期间就作为一个常数确定下来(C99 标准允许变长数组，GCC 编译器中实现了变长数组)。然而，c 语言中数组的元素可以是任何类型的对象，当然也可以是另外一个数组。这样，要仿真出一个多维数组就不是一件难事。

2、对于一个数组，我们只能够做两件事：确定该数组的大小，以及获得指向该数组下标为 0 的元素的指针。其他有关数组的操作，哪怕它们乍看上去是以数组下标进行运算的，实际上都是通过指针进行的。换句话说，任何一个数组下标运算都等同于一个对应的指针运算，因此我们完全可以依据指针行为定义数组下标的行为。

现在考虑下面的例子：

```
int i;
int *p;
int calendar[12][31];
```

上面声明的 calendar 是一个数组，**该数组拥有 12 个数组类型的元素，其中的每个元素都是一个拥有 31 个整型元素的数组。**因此，sizeof(calendar)的值是：31×12×sizeof(int)。

考虑一下，calendar[4]的含义是什么？因为 calender 是一个有着 12 个数组类型元素的数组，它的每个数组类型元素又是一个有着 31 个整型元素的数组，所以 calendar[4]是 calendar 数组的第 5 个元素，是 calendar 数组中 12 个有着 31 个整型元素的数组之一。因此，calendar[4]的行为也表现为一个有着 31 个整型元素的数组的行为。例如，sizeof(calendar[4])的结果是：31×sizeof(int)。

又如，p = calendar[4]；这个语句使指针 p 指向了数组 calendar[4]中下标为 0 的元素。因为 calendar[4]是一个数组，我们可以通过下标的形式来指定这个数组中的元素：i = calendar[4][7]，这个语句也可以写成下面这样而表达的意思保持不变：**i = *(calendar[4] + 7)，还可以进一步写成：i = *(*(calendar + 4) + 7)。**

下面我们再看：p = calendar；这个语句是非法的，因为 calendar 是一个二维数组，即“数组的数组”，在此处的上下文中使用 calendar 名称会将其转换为一个指向数组的指针。而 p 是一个指向整型变量的指针，两个指针的类型不一样，所以是非法的。显然，我们需要一种声明指向数组的指针的方法。

```
int calendar[12][31];
int (*monthp)[31];
monthp = calendar;
```

int (*monthp)[31] 语句声明的 *monthp 是一个拥有 31 个整型元素的数组，因此，monthp 就是一个指向这样的数组的指针。monthp 指向数组 calendar 的第一个元素。

HERE

指针与数组 2（c 和指针. P141. ）

- 1、数组的名的值是一个指针常量，不能试图将一个地址赋值给数组名；
- 2、当数组名作为 sizeof 操作符的操作数时，sizeof(arrayname)返回的是整个数组的长度，而不是指向数组的指针的长度；
- 3、当数组名作为单目操作符&的操作数，取一个数组名的地址所产生的是一个指向数组的指针，而不是一个指向某个指针常量值的指针。
- 4、指针和数组并不总是相等的。为了说明这个概念，请考虑下面这两个声明：

```
int a[5];
int *b;
```

a 和 b 能够互换吗？它们都具有指针值，它们都可以进行间接访问和下标操作。但是，它们还是有很大的区别的：声明一个数组时，编译器将根据声明所指定的元素数量为数组保留内存空间，然后再创建数组名，它的值是一个常量，指向这段空间的起始位置。声明一个指针变量时，编译器只为指针本身保留内存空间，它并不为任何整型值分配内存空间。而且，指针变量并未被初始化为指向任何现有的内存空间，如果它是一个自动变量，它甚至根本不会被初始化。把这两个声明用图的方法表示，可以发现它们之间存在显著的不同：

因此，上述声明后，表达式*a 是完全合法的，但表达式*b 却是非法的。*b 将访问内存中某个不确定的位置，或者导致程序终止。另一方面，表达式 b++可以通过编译，但是 a++却不能，因为 a 的值是一个常量。

```
#include<stdio.h>

int main()
{
    //注意 sizeof(num)的长度应该为 10*4=40
    int num[] = {0,1,2,3,4,5,6,7,8,9};
    printf(" sizeof(num) = %d\n", sizeof(num) );

    //注意 sizeof(str)的长度应该为 11,包括字符串后面的'\0'
    char str[] = "0123456789";
    printf(" sizeof(str) = %d\n", sizeof(str) );

    //注意 sizeof(str1)的长度应该为 10,不包括字符串后面的'\0'，但是，最好将字符串的最后一个字符设定为空
```

```
    char str1[] = {'0','1','2','3','4','5','6','7','8','9'};
    printf(" sizeof(str1) = %d\n", sizeof(str1) );

    //&num 的类型为'int (*)[10]'，表示的是一个指向长度为 10 的整形数组的指针
    int (*ptoint)[10] = &num;
    printf(" sizeof(ptoint) = %d, (*ptoint)[9] = %d\n", sizeof(ptoint), (*ptoint)[9] );

    //&str 的类型为'char (*)[11]'，表示的是一个指向长度为 11 的字符数组的指针，注意 str 数组的长度是 11，而不是 10
    char (*ptostr)[11] = &str;
    printf(" sizeof(ptostr) = %d, (*ptostr)[9] = %c\n", sizeof(ptostr), (*ptostr)[9] );

    //由于 p 指向的是数组 num[5]，所以对下标取负值后，不会超出数组的正常取值范围
    //该例子也说明了为什么下标检查在 C 语言中是一项困难的任务：下标引用可以作用于任意的指针，而不仅仅是数组名
    //作用于指针的下标引用的有效性即依赖于该指针当时恰好指向什么内容，也依赖于下标的值
    int *p = num + 5;
    printf(" p[-1] = %d, p[0] = %d, p[1] = %d \n", p[-1],p[0],p[1] );

    //下面的表达式中，'num[5]和 5[num]'的值是一样的，把它们转换成对等的间接访问表达式，它们都等同于*(num + 2)
    //'5[num]'这个古怪的表达式之所以可行，缘于 C 实现下标的方法。对编译器来说，这两种形式并无差别
    //但是，决不应该编写形如'5[num]'的表达式，因为它会大大的影响程序的可读性
    printf(" num[5] = %d, 5[num] = %d \n", num[5], 5[num] );
   getchar();
    return 0;
}
```

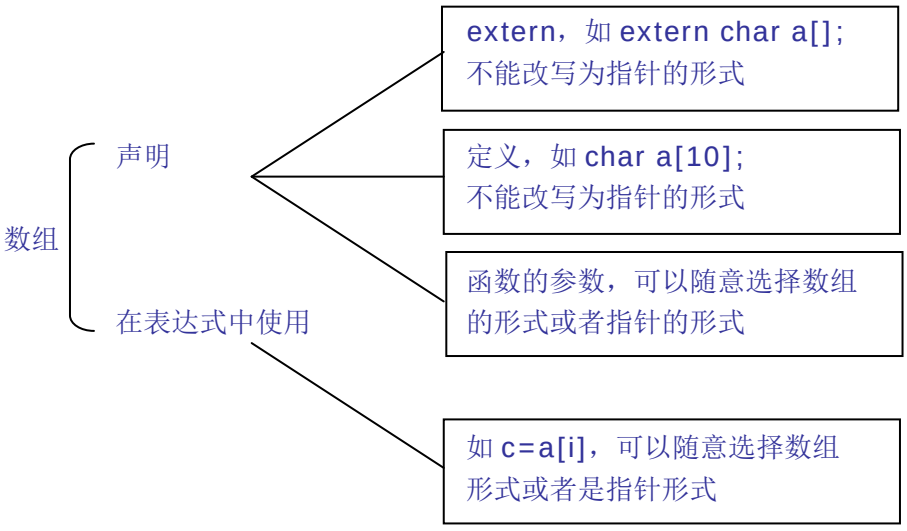
输出结果为：

```
sizeof(num) = 40
sizeof(str) = 11
sizeof(str1) = 10
sizeof(ptoint) = 4, (*ptoint)[9] = 9
sizeof(ptostr) = 4, (*ptostr)[9] = 9
p[-1] = 4, p[0] = 5, p[1] = 6
num[5] = 5, 5[num] = 5
```

指针和数组的相同与不同(c 专家编程. P199.)

在实际应用中，数组和指针可以互换的情形要比两者不可互换的情形更为常见。让我们分别考虑“声明”和“使用”这两种情况。声明本身还可以进一步分为 3 种情况：

- 外部数组的声明；
- 数组的定义（定义是声明的一种特殊情况，它分配内存空间，并可能提供一个初始值）；
- 函数参数的声明；



也既是：**作为函数参数时、在语句或表达式中使用数组时，我们可以采用数组或者指针的任何一种形式，除此之外的其他情况下，指针和数组不要互换**。下面就数组和指针相同的情况做详细的说明：

- **规则 1、表达式中的数组名被编译器当作一个指向该数组第一个元素的指针。**

  假如我们声明：

  ```
  int a[10];
  int *p = a;
  ```

  就可以通过一下任何一种方式来访问 a[i]：

  p[i]　　*(p + i)　　*(a + i)　　•••••

  事实上，可以采用的方法很多。对数组的引用如 a[i] 在编译时总是被编译器改写成*(a+i)的形式，C 语言标准要求编译器必须具备这个概念性的行为。

编译器自动把下标值的步长调整到数组元素的大小。如果整型数的长度是 4 个字节，那么 a[i+1]和 a[i]在内存中的距离就是 4。对起始地址执行加法操作之前，编译器会负责计算每次增加的步长。这就是为什么指针总是有类型限制，每个指针只能指向一种类型的原因所在，因为编译器需要知道对指针进行解除引用操作时应该取几个字节，以及每个下标的步长应取几个字节。

- **规则 2、下标总是和指针的偏移量相同。**

  把数组下标作为指针加偏移量是 c 语言从 BCPL(C 语言的祖先)继承过来的技巧。在人们的常规思维中，在运行时增加对 c 语言下标的范围检查是不切实际的。因为取下标操作只是表示将要访问该数组，但并不保证一定要访问。而且程序员完全可以使用指针来访问数组，从而绕过下标操作符。在这种情况下，数组下标范围检测并不能检测所有对数组的访问的情况。事实上，下标范围检测被认为不值得加入到 c 语言当中。

  还有一个说法是，在编写数组算法时，使用指针比使用数组更有效率。这个颇为人们所接收的说法在通常情况下是错误的。使用现代的产品质量优化的编译器，一维数组和指针引用所产生的代码并不具有显著的差别。不管怎样，数组下标是定义在指针的基础上，所以优化器常常可以把它转化为更有效率的指针表达式，并生成相同的机器指令。

- **规则 3、在函数参数的声明中，数组名被编译器当作指向该数组第一个元素的指针。**

  在函数形参定义这个特殊情况下，编译器必须把数组形式改写成指向数组第一个元素的指针形式。编译器只向函数传递数组的地址，而不是整个数组的拷贝。这种转换意味着在声明函数的时候，以下三种形式都是合法的(同时无论实参是数组还是真的指针也都是合法的)：

```
my_function( int *turnip ) { ••• }
my_function( int turnip[] ) { ••• }
my_function( int turnip[200] ) { ••• }
```

用 malloc 为字符串分配存储空间时的注意事项(c 缺陷与陷阱 3.2 节)

在C语言中，字符串常量代表了一块包括字符串中所有字符以及一个空字符（'\0'）的内存区域的地址。因为 C 语言要求字符串常量以空字符作为结束标志，对于其他字符串，C 程序员通常也沿用了这一惯例。

假定我们有两个这样的字符串 s 和 t，我们希望将这两个字符串连接成单个字符串 r 。要做到这一点，我们可以借助常用的库函数 strcpy 和 strcat。下面的方法似乎一目了然，可是却不能满足我们的目标：

```
char *r;
strcpy(r, s);
strcat(r, t);
```

之所以不行的原因在于不能确定 r 指向何处。我们还应该看到，不仅要让 r 指向一个地址，而且 r 所指向的地址处还应该有内存空间可供容纳字符串，这个内存空间应该是以某种方式已经被分配了的。

我们再试一次，记住给 r 分配一定的内存空间：

```
char r[100];
strcpy(r, s);
strcat(r, t);
```

只要 s 和 t 指向的字符串并不是太大，那么现在我们所用的方法就能够正常工作。不幸的是，C 语言强制要求我们必须声明数组大小为一个常量，因此我们不够确保 r 足够大。然而，大多数 C 语言实现为我们提供了一个库函数 malloc，该函数接受一个整数，然后分配能够容纳同样数目的字符的一块内存。大多数 C 语言实现还提供了一个库函数 strlen，该函数返回一个字符串中所包括的字符数。有了这两个库函数，似乎我们就能够像下面这样操作了：

```
char *r, *malloc( );
r = malloc(strlen(s) + strlen(t));
strcpy(r, s);
strcat(r, t);
```

这个例子还是错的，原因归纳起来有三个。第一个原因，malloc 函数有可能无法提供请求的内存，这种情况下 malloc 函数会通过返回一个空指针来作为“内存分配失败”事件的信号。

第二个原因，给 r 分配的内存在使用完之后应该及时释放，这一点务必要记住。因为在前面的程序例子中 r 是作为一个局部变量声明的，因此当离开 r 作用域时，r 自动被释放了。修订后的程序显式地给 r 分配了内存，为此就必须显式地释放内存。

第三个原因，也是最重要的原因，就是前面的例程在调用 malloc 函数时并未分配足够的内存。我们再回忆一下字符串以空字符作为结束标志的惯例。库函数 strlen 返回参数中字符串所包括的字符数目，而作为结束标志的空字符并未计算在内。因此，如果 strlen(s)的值是 n，那么字符串实际需要 n+1 个字符的空间。所以，

我们必须为 r 多分配一个字符的空间。做到了这些，并且注意检查了函数 malloc 是否调用成功，我们就得到正确的结果：

```
char *r, *malloc( );
r = malloc(strlen(s) + strlen(t) + 1);
if(!r) {
        complain();
       exit(1);
}
strcpy(r, s);
strcat(r, t);

/* 一段时间之后再使用 */
free(r);
```

作为常数的数组声明(c 缺陷与陷阱 3.3 节. 在其它部分有包含该节的知识点，了解 or 略过)

在 C 语言中，我们没有办法可以将一个数组作为函数参数直接传递。如果我们使用数组名作为参数，那么数组名会立刻被转换为指向该数组第 1 个元素的指针。例如，下面的语句：

```
char hello[] = "hello";
```

声明了 hello 是一个字符数组。如果将该数组作为参数传递给一个函数，

```
printf("%s\n", hello);
```

实际上与将该数组第 1 个元素的地址作为参数传递给函数的作用完全等效，即：

```
printf("%s\n", &hello[0]);
```

因此，将数组作为函数参数毫无意义。所以，C 语言中会自动地将作为参数的数组声明转换为相应的指针声明。也就是说，像这样的写法：

```
int strlen(char s[])
{
```

```
    /* 具体内容 */
    }
```

与下面的写法完全相同：

```
int strlen(char* s)
{
        /* 具体内容 */
}
```

C 程序员经常错误地假设，在其他情形下也会有这种自动地转换。本书 4.5 节详细地讨论了一个具体的例子，程序员经常在此处遇到麻烦：

```
extern char *hello;
```

这个语句与下面的语句有着天渊之别：

```
extern char hello[];
```

如果一个指针参数并不实际代表一个数组，即使从技术上而言是正确的，采用数组形式的记法经常会起到误导作用。如果一个指针参数代表一个数组，情况又是如何呢？一个常见的例子就是函数 main 的第二个参数：

```
main(int argc, char* argv[])
{
/* 具体内容 */
}
```

这种写法与下面的写法完全等价：

```
main(int argc, char** argv)
{
/* 具体内容 */
}
```

需要注意的是，前一种写法强调的重点在于 argv 是一个指向某数组的起始元素的指针，该数组的元素为字符指针类型。因为这两种写法是等价的，所以读者可以任选一种最能清楚反映自己意图的写法。

字符串常量(c 和指针. P269.)

当一个字符串常量出现在表达式中时，它的值是指针常量。编译器把该字符串的一份拷贝存储在内存的某个位置，并存储一个指向第一个字符的指针。我们可以对字符串常量进行下标引用、间接访问以及指针运算。

“xyz”+1

字符串常量实际上是个指针，这个表达式计算“指针值加上 1”的值。它的结果也是个指针，指向字符串中的第二个字符 y

* “xyz”

对一个指针执行间接访问操作时，其结果就是指针所指向的内容。字符串常量的类型是“指向字符的指针”，所以这个间接访问的结果就是它所指向的字符：x。注意表达式的结果并不是整个字符串，而只是它的第一个字符。

“xyz”[2]

同样可以推断出上面这个表达式的值就是字符 z。

```c
#include<stdio.h>

//接受一个无符号整型值，把它转换成字符，并打印出来
//如果是打印 16 进值的数，可以用这种方法：putchar( "0123456789ABCDEF"[ value % 16 ] )
void binary_to_ascii( unsigned long value )
{
    unsigned long quotient;

     quotient = value / 10;
     if( quotient != 0 )
          binary_to_ascii( quotient );

     putchar( "0123456789"[ value % 10 ] );
}

int main()
{

    //字符串常量实际上是个指针，这个表达式计算"指针值加上 1"的值。它的结果也是个指针，
    //指向字符串中的第二个字符：y
    printf( "%s\n", "xyz"+1 );

    //对一个指针执行间接访问操作时，其结果就是指针所指向的内容。
    //字符串常量的类型是"指向字符的指针"，所以这个间接访问的结果就是它所指向的字符：x
    printf( "%c\n", *"abcdefg" );

    //同样可以推断出上面这个表达式的值就是字符 z
    printf( "%c\n", "abcdefg"[3] );

    binary_to_ascii( 1234567 );

    getchar();
    return 0;
}
```

用字符串常量初始化指针和数组（c 专家编程. P87.）

定义指针时，编译器并不为指针所指的对象分配空间，它只是分配指针本身的空间，除非在定义时同时赋给指针一个字符串常量进行初始化。例如，下面的定义创建一个字符串常量（为其分配内存）：

char *p = “breadfruit”;

注意只有对字符串常量才是如此。不能指望为浮点数之类的变量分配空间，如：

float *pip = 3.14;　　/*错误，无法通过编译*/

在 ANSI C 中，初始化指针时所创建的字符串常量被定义为只读。如果试图通过指针修改这个字符串值，程序会出现未定义的行为。在有些编译器中，字符串常量被存放在只允许读取的文本段中，以防止它被修改。

数组也可以用字符串常量进行初始化：

char a[] = “gooseberry”;

与指针相反，由字符串常量初始化的数组是可以修改的。比如下面的语句：

strncpy(a, “black”, 5);

将数组的值修改为 “blackberry”。

```c
#include<stdio.h>
#include<string.h>

int main(void)
{
    char *p = "this is a example";
      //char *pi = 3.14; //这样定义是错误的，无法通过编译
      //p[0] = 'T'; //修改该字符串常量时，编译是没问题，但是运行时会出现异常

      char a[] = "gooseberry";
    strncpy( a, "black", 5 );

      printf("%s\n", p );
      printf("%s\n", a );

      return 0;
}
```

二维数组下标操作的相关概念(c 和指针. P156.)

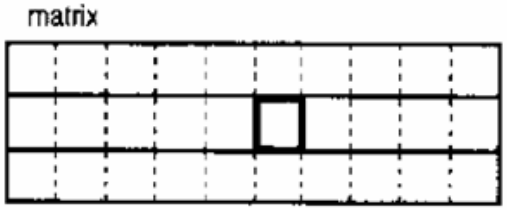
如果要标识一个多维数组的某个元素，必须按照与数组声明时相同的顺序为每一维都提供一个下标，而且每个下标都单独位于一对方括号内。在下面的声明中：

```
int matrix[3][10];
```

表达式

```
matrix[1][5]
```

访问下面这个元素：



但是，下标引用实际上只是间接访问表达式的一种伪装形式，即使在多维数组中也是如此。考虑下面这个表达式：

```
matrix
```

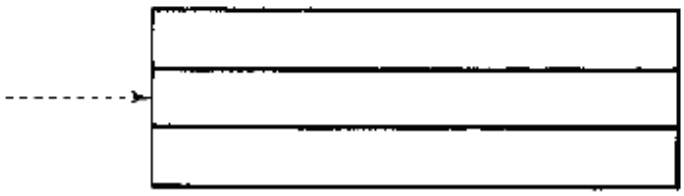
它的类型是“指向包含 10 个整型元素的数组的指针”，它的值是：

它指向包含 10 个整型元素的第 1 个子数组。

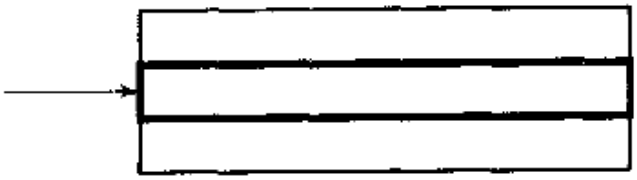
表达式

```
matrix + 1
```

也是一个“指向包含 10 个整型元素的数组的指针”，但它指向 matrix 的另一行：



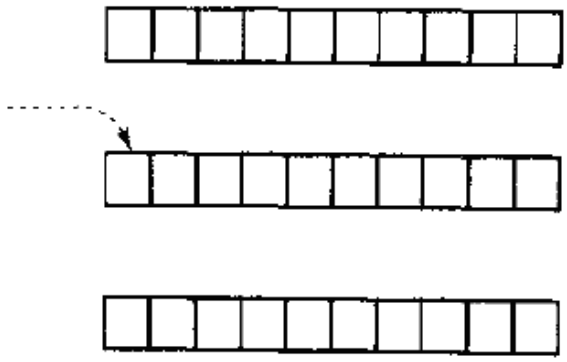
为什么？因为 1 这个值根据包含 10 个整型元素的数组的长度进行调整，所以它指向 matrix 的下一行。如果对其执行间接访问操作，就如下图随箭头选择中间这个子数组：



所以表达式

```
*( matrix + 1 )
```

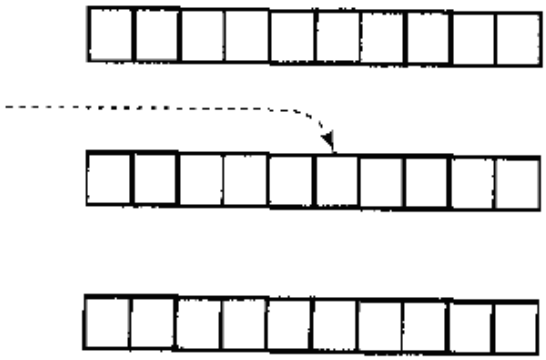
事实上标识了一个包含 10 个整型元素的子数组。数组名的值是个常量指针，它指向数组的第 1 个元素，在这个表达式中也是如此。它的类型是“指向整型的指针”，我们现在可以在下一维的上下文环境中显示它的值：



现在请拿稳你的帽子，猜猜下面这个表达式的结果是什么？

```
*( matrix + 1 ) + 5
```

前一个表达式是个指向整型值的指针，所以 5 这个值根据整型的长度进行调整。整个表达式的结果是一个指针，它指向的位置比原先那个表达式所指向的位置向后移动了 5 个整型元素。



对其执行间接访问操作：

```
*( *( matrix + 1 ) + 5 )
```

它所访问的正是图中的那个整型元素。如果它作为右值使用，你就取得存储于那个位置的值。如果它作为左值使用，这个位置将存储一个新值。

这个看上去吓人的表达式实际上正是我们的老朋友——下标。我们可以把子表达式*(*matrix + 1*)改写为 matrix[1]。把这个下标表达式代入原先的表达式，我们将得到：

```
*( matrix[1] + 5 )
```

这个表达式是完全合法的。matrix[1]选定一个子数组，所以它的类型是一个指向整型的指针。我们对这个指针加上5，然后执行间接访问操作。

但是，我们可以再次用下标代替间接访问，所以这个表达式还可以写成：

```
matrix[1][5]
```

这样，即使对于多维数组，下标仍然是另一种形式的间接访问表达式。

这个练习的要点在于它说明了多维数组中的下标引用是如何工作的，以及它们是如何依赖于指向数组的指针这个概念。下标是从左向右进行计算的，数组名是一个指向第1维第1个元素的指针，所以第1个下标值根据该元素的长度进行调整。它的结果是一个指向那一维中所需元素的指针。间接访问操作随后选择那个特定的元素。由于该元素本身是个数组，所以这个表达式的类型是一个指向下一维第1个元素的指针。下一个下标值根据这个长度进行调整，这个过程重复进行，直到所有的下标均计算完毕。

指向一维、二维数组的指针(c 和指针. P158.)

下面这些声明合法吗?

```
int vector[10], *vp = vector;
int matrix[3][10], *mp = matrix;
```

第 1 个声明是合法的。它为一个整型数组分配内存，并把 vp 声明为一个指向整型的指针，并把它初始化为指向 vector 数组的第 1 个元素。vector 和 vp 具有相同的类型：指向整型的指针。但是，第 2 个声明是非法的。它正确地创建了 matrix 数组，并把 mp 声明为一个指向整型的指针。但是，mp 的初始化是不正确的，因为 matrix 并不是一个指向整型的指针，而是一个指向整型数组的指针。我们应该怎样声明一个指向整型数组的指针的呢?

```
int (*p)[10];
```

这个声明比我们以前见过的所有声明更为复杂，但它事实上并不是很难。你只要假定它是一个表达式并对它求值。下标引用的优先级高于间接访问，但由于括号的存在，首先执行的还是间接访问。所以，p 是个指针，但它指向什么呢?

接下来执行的是下标引用，所以 p 指向某种类型的数组。这个声明表达式中并没有更多的操作符，所以数组的每个元素都是整数。

声明并没有直接告诉你 p 是什么，但推断它的类型并不困难——当我们对它执行间接访问操作时，我们得到的是个数组，对该数组进行下标引用操作得到的是一个整型值。所以 p 是一个指向整型数组的指针。

在声明中加上初始化后是下面这个样子:

```
int        (*p)[10] = matrix;
```

它使 p 指向 matrix 的第 1 行。

p 是一个指向拥有 10 个整型元素的数组的指针。当你把 p 与一个整数相加时，该整数值首先根据 10 个整型值的长度进行调整，然后再执行加法。所以我们可以使用这个指针一行一行地在 matrix 中移动。

如果你需要一个指针逐个访问整型元素而不是逐行在数组中移动，你应该怎么办呢?下面两个声明都创建了一个简单的整型指针，并以两种不同的方式进行初始化，指向 matrix 的第 1 个整型元素。

```
int        *pi = &matrix[0][0];
int        *pi   matrix[0];
```

增加这个指针的值使它指向下一个整型元素。

警告:

如果你打算在指针上执行任何指针运算，应该避免这种类型的声明:

```
int        (*p)[] = matrix;
```

p 仍然是一个指向整型数组的指针，但数组的长度却不见了。当某个整数与这种类型的指针执行指针运算时，它的值将根据空数组的长度进行调整(也就是说，与零相乘)，这很可能不是你所设想的。有些编译器可以捕捉到这类错误，但有些编译器却不能。

array_name 和&array_name 的异同

前者是指向数组中第一个元素的指针，后者是指向整个数组的指针。

```
char a[MAX];          /*array of MAX characters*/
char *p = a;          /*p 为指向数组的指针*/
char *pa = &a;        /*该语句是不正确的，pa 的类型为'char *'，而&a 的类型为'char (*)[MAX]’ */
char (*pb)[MAX] = &a;         /*该语句是正确的，pb 的类型为'char (*)[MAX]'*/
```

```
#include<stdio.h>

void main()
{
    char a[5] = {'a','b','c','d','\0'};
    char *p = a;

    //运行下面这句后， vc6.0 提示的错误为：cannot convert from 'char (*)[5]' to 'char *'，&a 的类型应该是指向一个数组的指针
    //char *pa = &a;
    //所以，应该定义一个指向相同类型和大小的数组的指针来获得“&a”的值
    char (*point_to_str)[5];
    point_to_str = &a;

    printf("%d\n%d\n",&p, &point_to_str);
    printf("%s\n%s\n", p,  point_to_str);
}

运行结果为：
   1245044
   1245040
   abcd
   abcd
```

数组作为函数的参数时，不能通过 sizeof 运算符得到该数组的大小

不可以。当把数组作为函数的参数时，你无法在程序运行时通过数组参数本身告诉函数该数组的大小，因为函数的数组参数相当于指向该数组第一个元素的指针。这意味着把数组传递给函数的效率非常高，也意味着程序员必须通过某种机制告诉函数数组参数的大小。为了告诉函数数组参数的大小，人们通常采用以下两种方法：

第一种方法是将数组和表示数组大小的值一起传递给函数，例如 memcpy()函数就是这样做的：

```
memcpy( dest, source, length );
```

第二种方法是引入某种规则来结束一个数组，例如在 C 语言中字符串总是以 ASCII 字符 NUL('\0')结束，而一个指针数组总是以空指针结束。请看下述函数，它的参数是一个以空指针结束的字符指针数组，这个空指针告诉该函数什么时候停止工作：

```
void printMany( char *strings[] )
{
   int i = 0;
   while( strings[i] != NULL )
   {
      puts(strings[i++]);
   }
}
```

C 程序员经常用指针来代替数组下标，因此大多数 C 程序员通常会将上述函数编写得更隐蔽一些：

```
void printMany( char *strings[] )
{
   while( *strings )
   {
      puts(*strings++);
   }
}
```

尽管你不能改变一个数组名的值，但是 strings 是一个数组参数，相当于一个指针，因此可以对它进行自增运算，并且可以在调用 puts()函数时对 strings 进行自增运算。

用 strlen()求字符串的长度(c 和指针. P159.)

库函数 strlen 的原型为：size_t strlen(char const *string);
strlen 返回一个类型为 size_t 的值。这个类型是在头文件 stddef.h 中定义的，它是一个无符号整型类型。
在表达式中使用无符号数可能导致不可预期的结果。例如，下面两个表达式看起来是相等的：

if(strlen(str1) >= strlen(str2))...
if(strlen(str1) - strlen(str2) >= 0)...

但事实上它们是不相等的，第 1 条语句会按照预想的那样工作，但第 2 条语句的结果将永远是真的。strlen 的结果是无符号数，所以操作符 >= 左边的表达式也将是无符号数，而无符号数决不可能是负的。表达式中如果同时包含了无符号数和有符号数，可能会产生奇怪的结果。和上面的一对语句一样，下面两条语句并不相等，原因相同。

if(strlen(str1) >= 10)...
if(strlen(str1) - 10 >= 0)...

如果将 strlen 的结果值强制转换成 int，就可以消除这个问题。类似的，**sizeof()的返回类型也是 size_t**，和 strlen()一样，也存在类似的现象（sizeof()是一个运算符，不是函数）。

对无符号类型的建议：
尽量不要在代码里面使用无符号类型，以免增加不必要的复杂性。尤其是，不要仅仅因为无符号数不存在负值而用它来表示数量。
尽量使用像 int 这样的有符号类型，这样在涉及升级混合类型的复杂细节时，不必担心边界情况。
对于返回类型为无符号的函数(strlen()、sizeof())，最好将结果转换成整型（ (int)strlen(...)、(int)sizeof(...) ），这样可以避免出现比较微妙的 bug（在 java 里面，就没有无符号数）。

```
#include<stdio.h>
#include<stddef.h>
#include<string.h>

int main()
{
    char str1[] = "0123456789";
    char str2[] = "abcdefghijk";

    // sizeof()的返回类型也是无符号类型，无符号类型的运算结果也被转换成无符号类型，不可能为负
    // (int)sizeof( str1 ) -(int) sizeof( str2 ) > 0 这个表达式将得到预期结果
    if( sizeof( str1 ) - sizeof( str2 ) > 0 )
    {
        printf(" 'sizeof( str1 ) - sizeof( str2 )'的计算结果是无符号型的，不可能为负\n");
    }

    if( strlen(str1) >= strlen(str2) )
    {
        printf(" strlen 的返回值为无符号整型类型，把两个无符号整型类型做比较，会得到预期的结果\n");
    }

    if( strlen(str1) - strlen(str2) >= 0 )
    {
        printf(" strlen(str1) = %d\n",strlen(str1) );
        printf(" strlen(str2) = %d\n",strlen(str2) );
        printf(" ( strlen(str1) - strlen(str2) >= 0 ) 表达式的值为： %d\n", strlen(str1) - strlen(str2) >= 0);
        printf(" 'strlen(str1) - strlen(str2)'的结果是无符号类型，无符号数不可能是负值，所以该条件永远成立\n");
    }
    //注意：sizeof()和 strlen()两个函数取得的值是不相等的
    //sizeof()求得的长度包括字符串末尾的那个空字符'\0'
    //strlen()求得的长度不包括字符串末尾的空字符
    printf(" sizeof(str1) = %d\n strlen(str1) = %d\n", sizeof(str1), strlen(str1) );

    getchar();
    return 0;
}
```

‘char **’ 和 ‘const char **’的兼容性问题(c 专家编程. P19.)

有时候必须非常专注的阅读 ANSI C 标准才能找到某个问题的答案。一位销售工程师把下面的代码作为测试例子发给 SUN 的编译器小组。

```
#include<stdio.h>

void foo( const char **P )
{}

int main( int argc, char **argv )
{
    foo( argv );
    return 0;
}
```

在 VC6.0 下编译这段代码，编译器会发出警告：

```
cannot convert parameter 1 from 'char ** ' to 'const char ** '
```

提交代码的工程师想知道为什么会产生类似的警告，他认为，实参 char *s 与形参 const char *p 应该是相容的，标准库中所有的字符串处理函数都是这样的。那么，为什么实参 char **argv 与形参 const char **P 实际上不能相容呢？答案是肯定的，它们并不相容。现在我们回顾一下标准中有关简单赋值的部分，它位于 ANSI C 第 6.3.16.1 节，描述了下列约束条件：

> 要使上述赋值形式合法，必须满足下列条件之一：
> 两个操作数都是指向有限定符或无限定符的相容类型的指针，左边指针所指向的类型必须具有右边指针所指向类型的全部限定符。

正是这个条件，使得函数调用中实参 char*能够与形参 const char*匹配。它之所以合法，是因为在下面的代码中：

```
char *cp;
const char *cpp;
cpp = cp;
```

左操作数是一个指向有 const 限定符的 char 的指针；

右操作数是一个指向没有限定符的 char 的指针；

char 类型和 char 类型是相容的，左操作数所指向的类型具有右操作数所指向类型的限定符(无),再加上自身的限定符 const (注意反过来不能赋值)。

标准第 6.3.16.1 节没有说明 char **实参与 const char **形参是否相容。标准 6.1.2.5 节中讲述实例的部分声称：const float * 类型并不是一个有限定符的类型，它的类型是“指向一个具有 const 限定符的 float 类型的指针”，也就是说 const 限定符是修饰指针所指向的类型，而不是指针。类似地，**const char **也是一个没有限定符的指针类型，它的类型是“指向有 const 限定符的 char 类型的指针的指针”。 由于 char ** 和 const char ** 都是没有限定符的指针类型，但它们所指向的类型不一样（前者指向 char *，后者指向 const char *），因此它们是不相容的**。因此类型为 char **的实参和类型为 const char **的形参是不相容的，编译器会产生一条诊断信息。

备注：解释的有些牵强，目前记住结果就可以了

空指针相关的问题(c 缺陷与陷阱 3.5 节)

除了一个重要的例外情况，在C语言中将一个整数转换为一个指针，最后得到的结果都取决于具体的C编译器实现。这个特殊情况就是常数0，编译器保证由0转换而来的指针不等于任何有效的指针。出于代码文档化的考虑，常数0这个值经常用一个符号来代替：

```
#define NULL 0
```

当然无论是直接用常数0，还是用符号NULL，效果都是相同的。需要记住的重要一点是，当常数0被转换为指针使用时，这个指针绝对不能被解除引用(dereference)。换句话说，当我们将0赋值给一个指针变量时，绝对不能企图使用该指针所指向的内存中存储的内容。下面的写法是完全合法的：

```
if (p == (char *) 0) ...
```

但是如果要写成这样：

```
if (strcmp(p, (char *) 0) == 0) ...
```

就是非法的了，原因在于库函数strcmp的实现中会包括查看它的指针参数所指向内存中的内容的操作。

如果p是一个空指针，即使

```
printf(p);
```

和

```
printf("%s", p);
```

的行为也是未定义的。而且，与此类似的语句在不同的计算机上会有不同的效果。

```c
#include<stdio.h>
#include<string.h>

int main()
{
    char *p = NULL;
    if( p == (char *)0 )
    {
        printf( "p is a null point\n" );
    }
    else
    {
        printf( "p is not a null point\n" );
    }

    //该语句不会引起编译错误，但是运行时会出现异常
    if( strcmp( p, (char *)0 ) == 0 )
    {
        printf("can't dereference p\n");
    }

    //该语句不会引起编译错误，但是运行时会出现异常
    printf("%d", *p );

    getchar();
    return 0;
```

```
}
```

NULL 和 NUL 的区别

NULL 是在<stddef.h>头文件中专门为空指针定义的一个宏。NUL 是 ASCII 字符集中第一个字符的名称，它对应于一个零值。C 语言中没有 NUL 这样的预定义宏。注意：在 ASCII 字符集中，数字 0 对应于十进制值 48，不要把数字 0 和'\0'(NUL)的值混同起来。

NULL 可以被定义为(void *)0，而 NUL 可以被定义为'\0'。NULL 和 NUL 都可以被简单地定义为 0，这时它们是等价的，可以互换使用，但这是一种不可取的方式。为了使程序读起来更清晰，维护起来更容易，你在程序中应该明确地将 NULL 定义为指针类型，而将 NUL 定义为字符类型。

对指针进行解引用操作可以获得它的值。从定义来看，NULL 指针并未指向任何东西。因此，对一个 NULL 指针进行解引用操作是非法的。在对指针进行解引用操作之前，必须确保它并非 NULL 指针。

未初始化的指针和 NULL 指针的区别(c 和指针. P95.)

未初始化的指针

下面这个代码段说明了一个极为常见的错误：

```
int     *a;
...
*a = 12;
```

这个声明创建了一个名叫 a 的指针变量，后面那条赋值语句把 12 存储在 a 所指向的内存位置。

警告：

但是究竟 a 指向哪里呢？我们声明了这个变量，但从未对它进行初始化，所以我们没有办法预测 12 这个值将存储于什么地方。从这一点看，指针变量和其他变量并无区别。如果变量是静态的，它会被初始化为 0；但如果变量是自动的，它根本不会被初始化。无论是哪种情况，声明一个指向整型的指针都不会“创建”用于存储整型值的内存空间。

所以，如果程序执行这个赋值操作，会发生什么情况呢？如果你运气好，a 的初始值会是个非法地址，这样赋值语句将会出错，从而终止程序。在 UNIX 系统上，这个错误被称为“段违例(segmentation violation)”或“内存错误(memory fault)”。它提示程序试图访问一个并未分配给程序的内存位置。在一台运行 Windows 的 PC 上，对未初始化或非法指针进行间接的访问操作是一般保护性异常(General Protection Exception)的根源之一。

对于那些要求整数必须存储于特定边界的机器而言，如果这种类型的数据在内存中的存储地址处在错误的边界上，那么对这个地址进行访问时将会产生一个错误。这种错误在 UNIX 系统中被称为“总线错误(bus error)”。

一个更为严重的情况是：这个指针偶尔可能包含了一个合法的地址。接下来的事很简单：位于那个位置的值被修改，虽然你并无意去修改它。像这种类型的错误非常难以捕捉，因为引发错误的代码可能与原先用于操作那个值的代码完全不相干。所以，在你对指针进行间接访问之前，必须非常小心，确保它们已被初始化！

NULL 指针

标准定义了 NULL 指针，它作为一个特殊的指针变量，表示不指向任何东西。要使一个指针变量为 NULL，你可以给它赋一个零值。为了测试一个指针变量是否为 NULL，你可以将它与零值进行比较。之所以选择零这个值是因为一种源代码约定。就机器内部而言，NULL 指针的实际值可能与此不同。在这种情况下，编译器将负责零值和内部值之间的翻译转换。

NULL 指针的概念是非常有用的，因为它给了你一种方法，表示某个特定的指针目前并未指向任何东西。例如，一个用于在某个数组中查找某个特定值的函数可能返回一个指向查找到的数组元素的指针。如果该数组不包含指定条件的值，函数就返回一个 NULL 指针。这个技巧允许返回值传达两个不同片段的信息。首先，有没有找到元素？其次，如果找到，它是哪个元素？

对指针进行解引用操作可以获得它所指向的值。但从定义上看，NULL 指针并未指向任何东西。因此，对一个 NULL 指针进行解引用操作是非法的。在对指针进行解引用操作之前，你首先必须确保它并非 NULL 指针。

警告：

如果对一个 NULL 指针进行间接访问会发生什么情况呢？它的结果因编译器而异。在有些机器上，它会访问内存位置零。编译器能够确保内存位置零没有存储任何变量，但机器并未妨碍你访问或修改这个位置。这种行为是非常不幸的，因为程序包含了一个错误，但机器却隐匿了它的症状，这样就使这个错误更加难以寻找。

在其他机器上，对 NULL 指针进行间接访问将引发一个错误，并终止程序。宣布这个错误比隐藏这个错误要好得多，因为程序员能够更容易修正它。

提示：

如果所有的指针变量（而不仅仅是位于静态内存中的指针变量）能够被自动初始化为 NULL，那实在是件幸事，但事实并非如此。不论你的机器对解引用 NULL 指针这种行为作何反应，对所有的指针变量进行显式的初始化是种好做法。如果你已经知道指针将被初始化为什么地址，就把它初始化为该地址，否则就把它初始化为 NULL。风格良好的程序会在指针解引用之前对它进行检查，这种初始化策略可以节省大量的调试时间。

理解函数的声明(c 缺陷与陷阱 2.1 节)

有一次，一个程序员与我交谈一个问题。他当时正在编写一个独立运行于某种微处理器上的 C 程序。当计算机启动时，硬件将调用首地址为 0 位置的子例程。

为了模拟开机启动时的情形，我们必须设计出一个 C 语句，以显式调用该子例程。经过一段时间的思考，我们最后得到的语句如下：

```
(*(void(*)())0)();
```

像这样的表达式恐怕会令每个 C 程序员的内心都“不寒而栗”。然而，他们大可不必对此望而生畏，因为构造这类表达式其实只有一条简单的规则：按照使用的方式来声明。

任何 C 变量的声明都由两部分组成：类型以及一组类似表达式的声明符(declarator)。声明符从表面上看与表达式有些类似，对它求值应该返回一个声明中给定类型的结果。最简单的声明符就是单个变量，如：

```
float f, g;
```

这个声明的含义是：当对其求值时，表达式 f 和 g 的类型为浮点数类型(float)。因为声明符与表达式的相似，所以我们也可以在声明符中任意使用括号：

```
float ((f));
```

这个声明的含义是：当对其求值时，((f))的类型为浮点类型，由此可以推知，f 也是浮点类型。

同样的逻辑也适用于函数和指针类型的声明，例如：

```
float ff();
```

这个声明的含义是：表达式 ff()求值结果是一个浮点数，也就是说，ff 是一个返回值为浮点类型的函数。类似地，

```
float *pf;
```

这个声明的含义是*pf 是一个浮点数，也就是说，pf 是一个指向浮点数的指针。

以上这些形式在声明中还可以组合起来，就像在表达式中进行组合一样。因此，

```
float *g(), (*h)();
```

表示*g()与(*h)()是浮点表达式。因为()结合优先级高于*，*g()也就是*(g())：g 是一个函数，该函数的返回值类型为指向浮点数的指针。同理，可以得出 h 是一个函数指针，h 所指向函数的返回值为浮点类型。

一旦我们知道了如何声明一个给定类型的变量，那么该类型的类型转换符就很容易得到了：只需要把声明中的变量名和声明末尾的分号去掉，再将剩余的部分用一个括号整个“封装”起来即可。例如，因为下面的声明：

```
float (*h)();
```

表示 h 是一个指向返回值为浮点类型的函数的指针，因此，

```
(float (*)())
```

表示一个“指向返回值为浮点类型的函数的指针”的类型转换符。

拥有了这些预备知识，我们现在可以分两步来分析表达式 (*(void(*)())0)() 。

第一步，假定变量 fp 是一个函数指针，那么如何调用 fp 所指向的函数呢？调用方法如下：

```
(*fp)();
```

因为 fp 是一个函数指针，那么*fp 就是该指针所指向的函数，所以(*fp)()就是调用该函数的方式。ANSI C 标准允许程序员将上式简写为 fp()，但是一定要记住这种写法只是一种简写形式。

在表达式(*fp)()中，*fp 两侧的括号非常重要，因为函数运算符()的优先级高于单目运算符*。如果*fp 两侧没有括号，那么*fp()实际上与*(fp())的含义完全一致，ANSI C 把它作为*((*fp)())的简写形式。

现在，剩下的问题就只是找到一个恰当的表达式来替换 fp。我们将在分析的第二步来解决这个问题。如果 C 编译器能够理解我们大脑中对于类型的认识，那么我们可以这样写：

```
(*0)();
```

上式并不能生效，因为运算符*必须要一个指针来做操作数。而且，这个指针还应该是一个函数指针，这样经运算符*作用后的结果才能作为函数被调用。因此，在上式中必须对 0 作类型转换，转换后的类型可以大致描述为："指向返回值为 void 类型的函数的指针"。

如果 fp 是一个指向返回值为 void 类型的函数的指针，那么(*fp)()的值为 void，fp 的声明如下：

```
void (*fp)();
```

因此，我们可以用下式来完成调用存储位置为 0 的子例程：

```
void (*fp)();
(*fp)();
```

> 译注：此处作者假设 fp 默认初始化为 0，这种写法不宜提倡。

这种写法的代价是多声明了一个"哑"变量。

但是，我们一旦知道如何声明一个变量，也就自然知道如何对一个常数进行类型转换，将其转型为该变量的类型：只需要在变量声明中将变量名去掉即可。因此，将常数 0 转型为"指向返回值为 void 的函数的指针"类型，可以这样写：

```
(void (*)())0
```

因此，我们可以用(void (*)())0 来替换 fp，从而得到：

```
(*(void (*)())0)();
```

末尾的分号使得表达式成为一个语句。

在我当初解决这个问题的时候，C 语言中还没有 typedef 声明。尽管不用 typedef 来解决这个问题对剖析本例的细节而言是一个很好的方式，但无疑使用 typedef 能够使表述更加清晰：

```
typedef void (*funcptr)();
(*(funcptr)0)();
```

这个棘手的例子并不是孤立的，还有一些 C 程序员经常遇到的问题，实际上和这个例子是同一个类型的。

函数参数的传值调用(c 和指针. P122.)

C 函数的所有参数均以“传值调用”方式进行传递，这意味着函数将获得参数值的一份拷贝。这样，函数可以放心修改这个拷贝值，而不必担心会修改调用程序实际传递给它的参数。这个行为与 Modula 和 Pascal 中的值参数（不是 var 参数）相同。

C 的规则很简单：所有参数都是传值调用。但是，如果被传递的参数是一个数组名，并且在函数中使用下标引用该数组的参数，那么在函数中对数组元素进行修改实际上修改的是调用程序中的数组元素。函数将访问调用程序的数组元素，数组并不会被复制。这个行为被称为“传址调用”，也就是许多其他语言所实现的 var 参数。

数组参数的这种行为似乎与传值调用规则相悖。但是，此处其实并无矛盾之处——数组名的值实际上是一个指针，传递给函数的就是这个指针的一份拷贝。下标引用实际上是间接访问的另一种形式，它可以对指针执行间接访问操作，访问指针指向的内存位置。参数（指针）实际上是一份拷贝，但在这份拷贝上执行间接访问操作所访问的是原先的数组。此处只要记住两个规则：

1. 传递给函数的标量参数是传值调用的。
2. 传递给函数的数组参数在行为上就像它们是通过传址调用的那样。

这里有一个有趣的问题。如果你想把一个数组名参数传递给函数，正确的函数形参应该是怎样的？它是应该声明为一个指针还是一个数组？

正如你所看到的那样，调用函数时实际传递的是一个指针，所以函数的形参实际上是个指针。但为了使程序员新手更容易上手一些，编译器也接受数组形式的函数形参。因此，下面两个函数原型是相等的：

```
int  strlen( char *string );
int  strlen( char string[] );
```

这个相等性暗示指针和数组名实际上是相等的。但千万不要被它糊弄了！这两个声明确实相等，但只是在**当前这个上下文环境中**。如果它们出现在别处，就可能完全不同，就像前面讨论的那样。但对于数组形参，你可以使用任何一种形式的声明。

你可以使用任何一种声明，但哪个“更加准确”呢？答案是指针，因为实参实际上是个指针，而不是数组。同样，表达式 sizeof string 的值是指向字符的指针的长度，而不是数组的长度。

现在你应该清楚为什么函数原型中的一维数组形参无需写明它的元素数目，因为函数并不为数组参数分配内存空间。形参只是一个指针，它指向的是已经在其他地方分配好内存的空间。这个事实解释了为什么数组形参可以与任何长度的数组匹配——它实际传递的只是指向数组第 1 个元素的指针。另一方面，这种实现方法使函数无法知道数组的长度。如果函数需要知道数组的长度，它必须作为一个显式的参数传递给函数。

```
#include<stdio.h>

char ga[] = "abcdefghijklm";

void my_array_func( char ca[10] )
{
    //&ca 相当于一个指向字符数组的指针的地址
    char **pp = &ca;
    printf(" &ca = %#x \n",&ca);
    printf(" ca = %#x \n",ca);
    printf(" &(ca[0]) = %#x \n",&(ca[0]));
    printf(" &(ca[1]) = %#x \n",&(ca[1]));
    printf(" sizeof(ca) = %d \n\n", sizeof(ca));
}
```

```
void my_pointer_func( char *pa )
{
    //&pa 相当于一个指向字符数组的指针的地址
    char **pp = &pa;
    printf(" &pa = %#x \n",&pa);
    printf(" pa = %#x \n",pa);
    printf(" &(pa[0]) = %#x \n",&(pa[0]));
    printf(" &(pa[1]) = %#x \n",&(pa[1]));
    printf(" sizeof(pa) = %d \n\n", sizeof(pa));
}

int main()
{
    //&ga 相当于一个指向字符数组的指针的地址
    char (*pp)[14] = &ga;
    printf(" &ga = %#x \n",&ga);
    printf(" ga = %#x \n",ga);
    printf(" &(ga[0]) = %#x \n",&(ga[0]));
    printf(" &(ga[1]) = %#x \n",&(ga[1]));
    printf(" sizeof(ga) = %d \n\n", sizeof(ga));
    my_array_func( ga );
    my_pointer_func( ga );
    getchar();
    return 0;
}
//摘自《c 专家编程》p216 页，做了部分修改
```

运行结果为：

```
&ga = 0x402000
ga = 0x402000
&(ga[0]) = 0x402000
&(ga[1]) = 0x402001
sizeof(ga) = 14

&ca = 0x22ff50
ca = 0x402000
&(ca[0]) = 0x402000
&(ca[1]) = 0x402001
sizeof(ca) = 4

&pa = 0x22ff50
pa = 0x402000
&(pa[0]) = 0x402000
&(pa[1]) = 0x402001
sizeof(pa) = 4
```

从结果可以看出，数组参数的地址和数组参数的第一个元素的地址是不一样的，比如&ca 和 ca 的值。

函数指针(c 和指针. P260.)

你不会每天都使用函数指针。但是，它们确有用武之地，最常见的两个用途是转换表(jump table)和作为参数传递给另一个函数。本节，我们将探索这两方面的一些技巧。但是，首先容我指出一个常见的错误，这是非常重要的。

警告：

简单声明一个函数指针并不意味着它马上就可以使用。和其他指针一样，对函数指针执行间接访问之前必须把它初始化为指向某个函数。下面的代码段说明了一种初始化函数指针的方法。

```
int f( int );
int (*pf)( int ) = &f;//这两种初始化形式都是正确的
int (*pf)( int ) = f;
```

第 2 个声明创建了函数指针 pf，并把它初始化为指向函数 f。函数指针的初始化也可以通过一条赋值语句来完成。在函数指针的初始化之前具有 f 的原型是很重要的，否则编译器就无法检查 f 的类型是否与 pf 所指向的类型一致。

初始化表达式中的&操作符是可选的，因为函数名被使用时总是由编译器把它转换为函数指针，&操作符只是显式地说明了编译器将隐式执行的任务。

在函数指针被声明并且初始化之后，我们就可以使用三种方式调用函数：

```
int ans;

ans = f(25);
ans = (*pf)(25);
ans = pf(25);
```

第 1 条语句简单地使用名字调用函数 f，但它的执行过程可能和你想象的不太一样。函数名 f 首先被转换为一个函数指针，该指针指定函数在内存中的位置。然后，函数调用操作符调用该函数，执行开始于这个地址的代码。

第 2 条语句对 pf 执行间接访问操作，它把函数指针转换为一个函数名。这个转换并不是真正需要的，因为编译器在执行函数调用操作符之前又会把它转换回去。不过，这条语句的效果和第 1 条语句是完全一样的。

第 3 条语句和前两条语句的效果是一样的。间接访问操作并非必需，因为编译器需要的是一个函数指针。这个例子显示了函数指针通常是如何使用的。

作为函数参数的多维数组(c 和指针. P159.)

作为函数参数的多维数组名的传递方式和一维数组名相同——实际传递的是个指向数组第 1 个元素的指针。但是，两者之间的区别在于，多维数组的每个元素本身是另外一个数组，编译器需要知道它的维数，以便为函数形参的下标表达式进行求值。这里有两个例子，说明了它们之间的区别：

```
int vector[10];
...
func1(vector);
```

参数 vector 的类型是指向整型的指针，所以 func1 的原型可以是下面两种中的任何一种：

```
void func1( int *vec );
void func1( int vec[] );
```

作用于 vec 上面的指针运算把整型的长度作为它的调整因子。

现在让我们来观察一个矩阵：

```
int matrix[3][10];
...
func2(matrix);
```

这里，参数 matrix 的类型是指向包含 10 个整型元素的数组的指针。func2 的原型应该是怎样的呢？你可以使用下面两种形式中的任何一种：

```
void func2( int (*mat)[10] );
void func2( int mat[][10] );
```

在这个函数中，mat 的第 1 个下标根据包含 10 个元素的整型数组的长度进行调整，接着第 2 个下标根据整型的长度进行调整，这和原先的 matrix 数组一样。

这里的关键在于编译器必须知道第 2 个及以后各维的长度才能对各下标进行求值，因此在原型中必须声明这些维的长度。第 1 维的长度并不需要，因为在计算下标值时用不到它。

在编写一维数组形参的函数原型时，你既可以把它写成数组的形式，也可以把它写成指针的形式。但是，对于多维数组，只有第 1 维可以进行如此选择。尤其是，把 func2 写成下面这样的原型是不正确的：

```
void func2( int **mat );
```

这个例子把 mat 声明为一个指向整型指针的指针，它和指向整型数组的指针并不是一回事。

```
#include<stdio.h>

int func1( int *vec )
{
     printf("一维数组做为参数：通过'int func1( int *vec )'的形式调用函数\n");
    return 0;
}

int func2( int vec[] )
{
     printf("一维数组做为参数：通过'int func2( int vec[] )'的形式调用函数\n");
    return 0;
}

void func3( int mat[3][10] )
{
    printf("二维数组做为参数：通过'void func3( int mat[3][10] )'的形式调用函数\n");
     return ;
}
```

```
void func4( int mat[][10] )
{
    printf("二维数组做为参数：通过'void func4( int mat[][10] )'的形式调用函数\n");
     return ;
}

void func5( int (*mat)[10] )
{
    printf("二维数组做为参数：通过'void func5( int (*mat)[10] )'的形式调用函数\n");
     return ;
}

void func6( int **p )
{
    printf("二维数组做为参数：通过'void func6( int **p )'的形式调用函数\n");
     return ;
}


int main()
{
    int vector[10] = {1,2,3,4,5,6,7,8,9,10};
     func1(vector);
     func2(vector);

     int matrix[3][10] = {
                                {1,2,3,4,5,6,7,8,9,10},
                                {1,2,3,4,5,6,7,8,9,10},
                                {1,2,3,4,5,6,7,8,9,10}
                            };

     func3( matrix );
     func4( matrix );
     func5( matrix );
     //将 matrix 参数传递给函数'void func6( int **p )'会出现异常
     //func6( matrix );

    getchar();
    return 0;
}
```

只有当二维数组是一个指向字符串的指针数组时，函数的声明才可以采取 char **my_array 的形式。这是因为字符串和指针都有一个显示的越界值(分别是 NUL 和 NULL)，可以作为结束标记。至于其它的类型，并没有一种类似的通用且可靠的值，所以并没有一种内资的方法知道何时到达数组某一维的结束位置。即使是指向字符串的指针数组，通常也需要一个计数参数 argc，以纪录字符串的数量。

```
#include<stdio.h>
#include<stdlib.h>

void my_func( char **p )
{
    while( *p != NULL )
          printf( "%s\n", *p++ );
}

int main()
{
    char *p[] = { "one", "two", "three", "four", "five", NULL };

    my_func( p );

    return 0;
}
```

强制类型转换相关概念(c 专家编程. P187.)

强制类型转换(cast)这个术语从 C 语言一诞生就开始使用，即用于类型转换，也用于消除类型歧义。可以很容易地把某种类型的数据强制转换为基本类型的数据：在括号里写上新类型的名称，然后把它们放在需要转化类型的表达式的前面。在强制转换一个更为复杂的类型时，可以采取如下的方法：

1、 一个对象的声明，它的类型就是想要转换的结果类型；

2、 删去标识符以及任何类似 extern 之类的存储限定符，并把剩余的内容放在一对括号里面；

3、 把第二步产生的内容放在需要进行类型转换的对象的左边。

作为一个例子，程序员经常发现他们需要强制类型转换以便使用 qsort()库函数。这个库函数接收 4 个参数，其中一个是指向比较函数的指针，qsort()函数的声明如下：

void qsort(void *buf，size_t num，size_t size，int(*comp)(const void *ele1，const void *ele2));

当调用 qsort 函数时，可以向它传递一个你所喜欢的比较函数。你的比较函数将接收实际的数据类型而不是 void*参数，就像下面这样：

```
int intcompare( const int *i, const int *j )
{
    return *i - *j;
}
```

这个函数并不与 qsort()的 comp()参数完全匹配，所以要进行强制类型转换。假定有一个数组 a，它具有 10 个元素，需要对它们进行排序，根据上面列出的 3 个步骤，可以发现对 qsort()的调用将会是下面这个样子：

```
qsort( a, 10, sizeof(a[0]), (int(*)(const void *, const void *))intcompare );
```

```
#include<stdio.h>
#include<stdlib.h>

//对整型数进行排序
int intcompare( const int *i, const int *j )
{
    return *i - *j;
}

//对 double 类型的数进行排序
int doublecompare( const double * i, const double * j )
{
    if( *i > *j )
        return 1;
     else if( *i < *j )
        return -1;
     else
        return 0;
}

int main()
{
    int base[]={ 3, 102, 5, -2, 98, 52, 18, 56, 38, 70 };
    int i, len = sizeof(base)/sizeof(base[0]);
    for( i = 0; i < len; i++ )
    {
       printf("%d, ", base[i]);
    }
    printf("\n");
    qsort( base, len, sizeof(base[0]), (int(*)(const void *, const void *))intcompare );
    for( i=0; i < len; i++)
    {
       printf("%d, ", base[i]);
    }

    printf("\n\n");
    double base1[]={1.2, 2.6, 3.4, 8.6, 14.3, 0.2, 7.9, 1.6, 9.2, 10.8, 5.55 };
    int len1 = sizeof(base1)/sizeof(base1[0]);
    for( i = 0; i < len1; i++ )
    {
       printf("%5.2f, ", base1[i]);
    }
    printf("\n");
```

```
    qsort( base1, len1, sizeof(base1[0]), (int(*)(const void *, const void *))doublecompare );
    for( i=0; i < len1; i++)
    {
       printf("%5.2f, ", base1[i]);
    }

    return 0;
}
```

可变参数相关问题(c 和指针. P135.)

在函数的原型中，列出了函数期望接受的参数，但原型只能显示固定数目的参数。让一个函数在不同的时候接受不同数目的参数是不是可以呢？答案是肯定的，但存在一些限制。考虑一个计算一系列值的平均值的函数。如果这些值存储于数组中，这个任务就太简单了，所以为了让问题变得更有趣一些，我们假定它们并不存储于数组中。程序 7.9a 试图完成这个任务。

这个函数存在几个问题。首先，它不对参数的数量进行测试，无法检测到参数过多这种情况。不过这个问题很好解决，简单加上测试就是了。其次，函数无法处理超过 5 个的值。要解决这个问题，你只有在已经很臃肿的代码中再增加一些类似的代码。

但是，当你试图用下面这种形式调用这个函数时，还存在一个更为严重的问题：

```
avg1 = average( 3, x, y, z );
```

这里只有 4 个参数，但函数具有 6 个形参。标准是这样定义这种情况的：这种行为的后果是未定义的。这样，第 1 个参数可能会与 n_values 对应，也可能与形参 v2 对应。你当然可以测试一下你的编译器是如何处理这种情况的，但这个程序显然是不可移植的。我们需要的是一种机制，它能够以一种良好定义的方法访问数量未定的参数列表。

```
/*
** 计算指定数目的值的平均值（差的方案）。
*/

float
average( int n_values, int v1, int v2, int v3, int v4, int v5 )
{
        float  sum = v1;

        if( n_values >= 2 )
            sum += v2;
        if( n_values >= 3 )
            sum += v3;
        if( n_values >= 4 )
            sum += v4;
        if( n_values >= 5 )
            sum += v5;
        return sum / n_values;
}
```

程序 7.9a 计算标量参数的平均值：差的版本 average1.c

7.6.1 stdarg 宏

可变参数列表是通过宏来实现的，这些宏定义于 stdarg.h 头文件，它是标准库的一部分。这个头文件声明了一个类型 va_list 和三个宏——va_start、va_arg 和 va_end[1]。我们可以声明一个类型为 va_list 的变量，与这几个宏配合使用，访问参数的值。

程序 7.9b 使用这一个宏正确地完成了程序 7.9a 试图完成的任务。注意参数列表中的省略号：它提示此处可能传递数量和类型未确定的参数。在编写这个函数的原型时，也要使用相同的记法。

函数声明了一个名叫 var_arg 的变量，它用于访问参数列表的未确定部分。这个变量通过调用 va_start 来初始化。它的第 1 个参数是 va_list 变量的名字，第 2 个参数是省略号前最后一个有名字的参数。初始化过程把 var_arg 变量设置为指向可变参数部分的第 1 个参数。

为了访问参数，需要使用 va_arg，这个宏接受两个参数：va_list 变量和参数列表中下一个参数的类型。在这个例子中，所有的可变参数都是整型，在有些函数中，你可能要通过前面获得的数据来判断下一个参数的类型[2]。va_arg 返回这个参数的值，并使 var_arg 指向下一个可变参数。

最后，当访问完毕最后一个可变参数之后，我们需要调用 va_end。

7.6.2 可变参数的限制

注意，可变参数必须从头到尾按照顺序逐个访问。如果你在访问了几个可变参数后想半途中止，这是可以的。但是，如果你想一开始就访问参数列表中间的参数，那是不行的。另外，由于参数列表中的可变参数部分并没有原型，所以，所有作为可变参数传递给函数的值都将执行缺省参数类型提升。

```
/*
** 计算指定数量的值的平均值。
*/

#include <stdarg.h>

float
average( int n_values, ... )
{
        va_list  var_arg;
        int  count;
        float sum = 0;

        /*
        ** 准备访问可变参数。
        */
        va_start( var_arg, n_values );

        /*
        ** 添加取自可变参数列表的值。
        */
        for( count = 0; count < n_values; count += 1 ){
```

```
            sum += va_arg( var_arg, int );
        }

        /*
        ** 完成处理可变参数。
        */
        va_end( var_arg );

        return sum / n_values;
    }
```

程序 7.9b 计算标量参数的平均值：正确版本 average2.c

你可能同时注意到参数列表中至少要有一个命名参数。如果连一个命名参数也没有，你就无法使用 va_start。这个参数提供了一种方法，用于查找参数列表的可变部分。

对于这些宏，存在两个基本的限制。一个值的类型无法简单地通过检查它的位模式来判断，这两个限制就是这个事实的直接结果。

1. 这些宏无法判断实际存在的参数的数量。
2. 这些宏无法判断每个参数的类型。

要回答这两个问题，就必须使用命名参数。在程序 7.9b 中，命名参数指定了实际传递的参数数量，不过它们的类型被假定为整型。printf 函数中的命名参数是格式字符串，它不仅指定了参数的数量，而且指定了参数的类型。

警告：

如果你在 va_arg 中指定了错误的类型，那么其结果是不可预测的。这个错误是很容易发生的，因为 va_arg 无法正确识别作用于可变参数之上的缺省参数类型提升。char、short 和 float 类型的值实际上将作为 int 或 double 类型的值传递给函数。所以你在 va_arg 中使用后面这些类型时应该小心。

```
#include<stdio.h>
#include<stdarg.h>

float average( int n_values, ... )
{
    va_list var_arg;
    int count;
    float sum = 0;

    /*准备访问可变参数*/
    va_start( var_arg, n_values );

    /*添加取自可变参数的值*/
    for( count = 0; count < n_values; count++ )
    {
        sum = sum + va_arg( var_arg, int );
    }

    /*完成处理可变参数*/
    va_end( var_arg );

    return sum / n_values;
}

int main()
{

    int num[] = {1,2,3,4,5,6,7,8,9,10};
    float result1=0, result2=0;
```

```
    result1 = average(3, num[0], num[1], num[2] );
    result2 = average(6, num[0], num[1], num[2], num[3], num[4], num[5] );

    printf("average(3, num[0], num[1], num[2] ) = %f\n", result1 );
    printf("average(6, num[0], num[1], num[2], num[3], num[4], num[5] ) = %f\n", result2);
  getchar();
    return 0;
}
```

malloc()、calloc()、realloc()

函数 malloc()和 calloc()都可以用来分配动态内存空间，但两者稍有区别。malloc()函数有一个参数，即要分配的内存空间的大小：

void *malloc(size_t size);

calloc()函数有两个参数，分别为元素的数目和每个元素的大小，两个参数的乘积就是要分配的空间的大小：

void *calloc(size_t numElements，size_t sizeOfElement);

如果调用成功，函数 malloc()和 calloc()都将返回所分配的内存空间的首地址。

malloc()函数和 calloc()函数的主要区别是前者不能初始化所分配的内存空间，而后者能。如果由 malloc()函数分配的内存空间原来没有被使用过，则其中的每一位可能都是 0；反之，如果这部分内存空间曾经被分配、释放和重新分配，则其中可能遗留各种各样的数据。也就是说，使用 malloc()函数的程序开始时(内存空间还没有被重新分配)能正常运行，但经过一段时间后(内存空间已被重新分配)可能会出现问题。

calloc()函数会将所分配的内存空间中的每一位都初始化为零，也就是说，如果你是为字符类型或整数类型的元素分配内存，那么这些元素将保证会被初始化为零；如果你是为指针类型的元素分配内存，那么这些元素通常(但无法保证)会被初始化为空指针；如果你是为实数类型的元素分配内存，那么这些元素可能(只在某些计算机中)会被初始化为浮点型的零。

malloc()函数和 calloc()函数的另一点区别是 calloc()函数会返回一个由某种对象组成的数组，但 malloc()函数只返回一个对象。为了明确是为一个数组分配内存空间，有些程序员会选用 calloc()函数。但是，除了是否初始化所分配的内存空间这一点之外，绝大多数程序员认为以下两种函数调用方式没有区别：

calloc(numElements，sizeOfElement)；

malloc(numElements * sizeOfElement)；

需要解释的一点是，理论上(按照 ANSIC 标准)指针的算术运算只能在一个指定的数组中进行，但是在实践中，即使 C 编译程序或翻译器遵循这种规定，许多 C 程序还是冲破了这种限制。因此，尽管 malloc()函数并不能返回一个数组，它所分配的内存空间仍然能供一个数组使用(对 realloc()函数来说同样如此，尽管它也不能返回一个数组)。总之，当你在 calloc()函数和 malloc()函数之间作选择时，你只需考虑是否要初始化所分配的内存空间，而不用考虑函数是否能返回一个数组。

realloc 函数用于修改一个原先已经分配的内存块的大小。使用这个函数，你可以使一块内存扩大或者缩小。如果它用于扩大一个内存块，那么这块内存原先的内容依然保留，新增加的内存添加到原先内存块的后面，新内存并未以任何方法进行初始化。如果它用于缩小一个内存块，该内存块尾部的部分内存便被拿掉，剩余部分内存的原先内容依然保留。如果原先的内存块无法改变大小，realloc 将分配另外一块正确大小的内存，并把原先那块内存的内容复制到新的块上，因此，在使用 realloc 之后，就不能再使用指向旧内存的指针，而是应该改用 realloc 所返回的新指针。

常见的动态内存错误(c 和指针. P223.)

在使用动态内存分配的程序中，常常会出现许多错误。这些错误包括对 NULL 指针进行解除引用操作、对分配的内存进行操作时越过边界、释放并非动态分配的内存、试图释放一块动态分配的内存的一部分以及一块动态内存被释放之后还继续使用它。以下是一些需要注意的事项：

1、在请求动态内存分配时，要检查所请求的内存是否成功分配。

2、操作内存时，不要超过动态分配的内存的边界。对分配的内存之外的区域进行访问可能会破坏别的数据，产生一些莫名其妙的很难发现的 bug。

3、传递给 free 的指针必须是一个从 malloc、calloc、realloc 函数返回的指针。

4、动态分配的内存必须整块一起释放，不允许释放一块动态分配的内存的一部分(realloc 函数可以缩小一块动态分配的内存，有效地释放它尾部的部分内存)。

在程序退出 main()函数之后，还有可能执行一部分代码吗?

可以，但这要借助 C 库函数 atexit()。利用 atexit()函数可以在程序终止前完成一些“清理”工作——如果将指向一组函数的指针传递给 atexit()函数，那么在程序退出 main()函数后(此时程序还未终止)就能自动调用这组函数。在使用 atexit()函数时你要注意这样两点：

第一：由 atexit()函数指定的要在程序终止前执行的函数要用关键字 void 说明，并且不能带参数；

第二：由 atexit()函数指定的函数在入栈时的顺序和调用 atexit()函数的顺序相反，即它们在执行时遵循后进先出(LIFO)的原则。

```
#include<stdlib.h>
#include<stdio.h>

void my_exit1(void)
{
    printf("my_exit1() function !\n");
}

void my_exit2(void)
{
    printf("my_exit2() function !\n");
}

void   main()
{
    atexit ( my_exit1 );
    atexit ( my_exit2 );
    printf("now, eixt this program...\n");
}
输出结果为：
now, eixt this program...
my_exit2() function !
my_exit1() function !
```

总线错误和段错误相关概念(c 专家编程. P157.)

在 UNIX 上编程时，经常会遇到如下两个常见的运行时错误：
bus error　（总线错误）
segmentation fault　（段错误）

总线错误

总线错误几乎都是由于未对齐的读或写造成的。它之所以称为总线错误，是因为出现未对齐的内存访问请求时，被堵塞的组件就是地址总线。对齐的意思就是数据项只能存储在地址是数据项大小的整数倍的内存位置上。在现代的计算机架构中，尤其是 RISC 架构，都需要数据对齐，因为与任意的对齐有关的额外逻辑会使整个内存系统更大且更慢。通过迫使每个内存访问局限在一个 cache 行或一个单独的页面内，可以极大地简化如 cache 控制器或内存管理单元这样的硬件。

我们表达“数据项不能跨越页面或 cache 边界”规则的方法多少有些间接，因为我们用地址对齐这个术语来陈述这个问题，而不是直截了当说是禁止内存跨页访问，但它们说的是同一回事。例如，访问一个 8 字节的 double 数据时，地址只允许是 8 的整数倍。所以一个 double 数据可以存储于地址 24、8008、32768，但不能存储于地址 1006，页和 cache 的大小是经过精心设计的，这样只要遵守对齐规则就可以保证一个原子数据项不会跨越一个页或 cache 块的边界。

段错误

段错误通常是由于解除引用一个未初始化或非法值的指针引起的。以发生频率为序，最终可能导致段错误的常见编程错误是：

1、坏指针错误：在指针赋值之前就用它来引用内存；或者向库函数传递一个坏指针(如果调试器显示系统程序中出现了段错误，很可能并不是系统程序引起的段错误，问题可能就出现在自己的代码中)；或者指针被释放后还继续访问它的内容。

2、改写错误：越过数组边界写入数据，在动态分配的内存空间以外写入数据，或改写一些堆管理数据结构(在动态分配的内存之前的区域写入数据就很容易发生这种情况)。

3、指针释放引起的错误：释放同一块内存两次，或释放一块未曾使用 malloc 分类的内存，或释放一个无效的指针。一个极为常见的与释放内存有关的错误就是在 for(p=start;p;p=p->next) 这样的循环中迭代一个链表，并在循环体内使用 free(p) 这样的语句。这样，在下一次循环迭代时，程序就会对已经释放的指针进行解除引用操作，从而导致不可预料的结果。

怎样判断一个字符是数字、字母或其它类别的符号?

在头文件 ctype.h 中定义了一批函数，它们可用来判断一个字符属于哪一类别。下面列出了这些函数：

函数	字符类别	返回非零值的字符
isdigit()	十进制数	0--9
isxdigit()	十六进制数	0--9，a—f，或 A--F
isalnum()	字母数字符号	0--9，a--Z，或 A--Z
isalpha()	字母	a—z 或 A--Z
islower()	小写字母	a—z
isupper()	大写字母	A--Z
isspace()	空白符	空格符，水平制表符，垂直制表符，换行符，换页符，或回车符
isgraph()	非空白字符	任何打印出来不是空白的字符(ASCII 码从 21 到 7E)
isprint()	可打印字符	所有非空白字符，加上空格符
ispunct()	标点符	除字母数字符号以外的所有非空白字符
iscntrl()	控制字符	除可打印字符外的所有字符(ASCII 码从 00 到 1F，加上 7F)

与前文提到过的使用标准库函数的好处相似，调用上述这些宏而不是自己编写测试字符类别的程序也有三点好处。**首先**，这些宏运算速度快，因为它们的实现方式通常都是利用位屏蔽技术来检查一个表，所以即使是进行一项相当复杂的检查，也比真正去比较字符的值要快得多。**其次**，这些宏都是正确的。如果你自己编写一个测试程序，你很容易犯逻辑上或输入上的错误，例如引入了一个错误的字符(或漏掉了一个正确的字符)。**第三**，这些宏是可移植的。信不信由你，并非所有的人都使用同样的含 PC 扩充字符的 ASCII 字符集。也许今天你还不太在意，但是，当你发现你的下一台计算机使用的是 Unicode 字符集而不是 ASCII 字符集，你就会庆幸自己原来没有按照字符集中的字符值来编写程序。

其他字符转换函数：

isascii（测试字符是否为 ASCII 码字符）

```
int isascii(int c);
```

检查参数 c 是否为 ASCII 码字符，也就是判断 c 的范围是否在 0 到 127 之间。若参数 c 为 ASCII 码字符，则返回 TRUE，否则返回 NULL(0)。

toascii（将整型数转换成合法的 ASCII 码字符）
int toascii(int c);
toascii()会将参数 c 转换成 7 位的 unsigned char 值，第八位则会被清除，此字符即会被转成 ASCII 码字符。将转换成功的 ASCII 码字符值返回。

tolower（将大写字母转换成小写字母）
int tolower(int c);
若参数 c 为大写字母则将该对应的小写字母返回。返回转换后的小写字母，若不须转换则将参数 c 值返回。

toupper（将小写字母转换成大写字母）
int toupper(int c);
若参数 c 为小写字母则将该对映的大写字母返回。 返回转换后的大写字母，若不须转换则将参数 c 值返回。

以 isalmun 为例，说明这些函数的用法[剩余的其他函数跟它类似]：

int isalnum (int c)

检查参数 c 是否为英文字母或阿拉伯数字，若参数 c 为字母或数字，则返回 TRUE，否则返回 NULL。此为宏定义，非真正函数。

```
#include<stdio.h>
#include <ctype.h>
int main()
{
    char str[]="123c@#FDsP[e?";

    for( int i = 0; str[i] != 0; i++ )
    {
        if( isalnum( str[i] ) )
            printf("%c is an alphanumeric character\n", str[i] );
    }
    return 0;
}
```

怎样将数字转换为字符串?

C 语言提供了几个标准库函数，可以将任意类型(整型、长整型、浮点型等)的数字转换为字符串。以下是用 itoa() 函数将整数转换为字符串的一个例子：

```
# include <stdio.h>
# include <stdlib.h>

int main ()
{
    int num = 435443435;
    char str[30];
    itoa( num, str, 10 );
    printf("The number 'num' is %d and the string 'str' is %s. \n" , num, str );
    getchar();
    return 0;
}
```

itoa()函数有 3 个参数：第一个参数是要转换的数字，第二个参数是要写入转换结果的目标字符串，第三个参数是转移数字时所用的基数。在上例中，转换基数为 10。

函数名	作　用
itoa()	将整型值转换为字符串
ltoa()	将长整型值转换为字符串
ultoa()	将无符号长整型值转换为字符串

请注意，上述函数与 ANSI 标准是不兼容的。能将整数转换为字符串而且与 ANSI 标准兼容的方法是使用 sprintf() 函数， 请看下例：

```
#include<stdio.h>
#include <stdlib.h>

int main ()
{
    int num = 123456;
    char str[50];
    sprintf(str, "%d" , num);
    printf ("The number 'num' is %d and the string 'str' is %s. \n" ,  num, str);

    getchar();
    return 0;
}
```

gcvt（将浮点型数转换为字符串，取四舍五入）

```
char *gcvt( double number, size_t ndigits, char *buf );
```

gcvt()用来将参数 number 转换成 ASCII 码字符串，参数 ndigits 表示显示的位数。gcvt()与 ecvt()和 fcvt()不同的地方在于，gcvt()所转换后的字符串包含小数点或正负符号。若转换成功，转换后的字符串会放在参数 buf 指针所指的空间。该函数返回一字符串指针，此地址即为 buf 指针。

ecvt() ：将双精度浮点型值转换为字符串，转换结果中不包含十进制小数点
fcvt()： 以指定位数为转换精度，其余同 ecvt()

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double a = 123.45546;
    double b = -1234.56;
    char ptr[20] = {0};

    gcvt( a, 7, ptr );
    printf( "a value = %s\n",ptr );
    char *p = gcvt( b, 5, ptr );
    printf( "b value = %s\n",p );
}
```

怎样将字符串转换为数字?

下列函数可以将字符串转换为数字：

函数名	作　用
atof()	将字符串转换为双精度浮点型值
atoi()	将字符串转换为整型值
atol()	将字符串转换为长整型值
strtod()	将字符串转换为双精度浮点型值，并报告不能被转换的所有剩余数字
strtol()	将字符串转换为长整值，并报告不能被转换的所有剩余数字
strtoul()	将字符串转换为无符号长整型值，并报告不能被转换的所有剩余数字

atof（将字符串转换成浮点型数）
double atof(const char *nptr);
atof()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。参数 nptr 字符串可包含正负号、小数点或 E(e)来表示指数部分，如 123.456 或 123e-2。该函数返回转换后的浮点型数。atof()与使用 strtod(nptr,(char**)NULL)结果相同。

atoi（将字符串转换成整型数）
int atoi(const char *nptr);
atoi()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。该函数返回转换后的整型数。atoi()与使用 strtol(nptr，(char**)NULL，10)；结果相同。

atol（将字符串转换成长整型数）
long atol(const char *nptr);
atol()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。该函数返回转换后的长整型数。atol()与使用 strtol(nptr,(char**)NULL,10)；结果相同。

```
# include <stdio.h>
# include <stdlib.h>

int main ()
{
    char *a0 ="-1000";
    char *b0 ="456";
    int c0 = atoi( a0 )+atoi( b0 );
    printf( "atoi:   c0 = %d\n", c0 );

    char *a1 = "-100.23";
    char *b1 = "200e-2";
    float c1 = atof( a1 ) + atof( b1 );
    printf( "atof:   c1 = %f\n", c1 );

    char *a2 = "4563453";
    char *b2 = "4563431237";
    long c2 = atol( a2 ) + atol( b2 );
    printf( "atol:   c2 = %ld\n", c2 );
}
```

strtod（将字符串转换成浮点数）

double strtod(const char *nptr, char **endptr);

strtod()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，到出现非数字或字符串结束时('\0')才结束转换，并将结果返回。若 endptr 不为 NULL，则会将遇到不合条件而终止的 nptr 中的字符指针由 endptr 传回。参数 nptr 字符串可包含正负号、小数点或 E(e)来表示指数部分。如 123.456 或 123e-2。该函数返回转换后的浮点型数。

strtol（将字符串转换成长整型数）

long int strtol(const char *nptr, char **endptr, int base);

strtol()会将参数 nptr 字符串根据参数 base 来转换成长整型数。参数 base 范围从 2 至 36，或 0。参数 base 代表采用的进制方式，如 base 值为 10 则采用 10 进制，若 base 值为 16 则采用 16 进制等。当 base 值为 0 时则是采用 10 进制做转换，但遇到如'0x'前置字符则会使用 16 进制做转换。一开始 strtol()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数 endptr 不为 NULL，则会将遇到不合条件而终止的 nptr 中的字符指针由 endptr 返回。该函数返回转换后的长整型数，否则返回 ERANGE 并将错误代码存入 errno 中(ERANGE 指定的转换字符串超出合法范围)

strtoul（将字符串转换成无符号长整型数）

unsigned long int strtoul(const char *nptr,char **endptr,int base);

strtoul()会将参数 nptr 字符串根据参数 base 来转换成无符号的长整型数。参数 base 范围从 2 至 36，或 0。参数 base 代表采用的进制方式，如 base 值为 10 则采用 10 进制，若 base 值为 16 则采用 16 进制数等。当 base 值为 0 时则是采用 10 进制做转换，但遇到如'0x'前置字符则会使用 16 进制做转换。一开始 strtoul()会扫描参数 nptr 字符串，跳过前面的空格字符串，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数 endptr 不为 NULL，则会将遇到不合条件而终止的 nptr 中的字符指针由 endptr 返回。该函数返回转换后的长整型数，否则返回 ERANGE 并将错误代码存入 errno 中(ERANGE 指定的转换字符串超出合法范围)

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
    char a[]="1000000000abcde";
    char b[]="1000000000abcde";
    char c[]="ffffOPQRST";
    char **p;

    printf("a = %d, p = %s\n",strtol(a, p, 10), *p );
    printf("b = %d, p = %s\n",strtol(b, p,  2), *p );
    printf("c = %d, p = %s\n",strtol(c, p, 16), *p );
}
```

输出结果为：

```
a = 1000000000, p = abcde
b = 512, p = abcde
c = 65535, p = OPQRST
```

字符串拷贝和内存拷贝函数：

strcpy（拷贝字符串）

定义函数：char *strcpy(char *dest, const char *src);

strcpy()函数只能拷贝字符串。strcpy()函数将源字符串 src 的每个字节拷贝到目的字符串 dest 中，src 字符串末尾的'\0'也被拷贝过去。strcpy()函数返回参数 dest 的起始地址。如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况（程序员必须保证目标字符数组的空间足够容纳需要复制的字符串。如果 **src** 字符串比 **dest** 字符串长，多余的字符仍将被复制，它们将覆盖原先存储于 **dest** 数组后面的内存空间的值），在编写程序时请特别留意，或者用 strncpy()来取代。如果参数 src 和 dst 在内存中出现重叠，其结果是未定义的。

strncpy（拷贝字符串）

定义函数：char *strncpy(char *dest, const char *src, size_t n);

strncpy()会将参数 src 字符串拷贝前 n 个字符至参数 dest 所指的地址。函数返回参数 dest 的字符串起始地址。注意 n 的取值范围，不要超过 src 和 dest 的长度。

```
#include<string.h>
#include<stdio.h>

int main()
{
    char a1[30]="string(1)";
    char b1[]="STRING(2)";
    printf("before strcpy()  : %s\n", a1 );
    printf("after  strcpy()  : %s\n", strcpy( a1, b1 ) );

    char a2[30]="string(1)";
    char b2[]="STRING(2)";
    printf("before strncpy() : %s\n", a2 );
    printf("after  strncpy() : %s\n", strncpy( a2, b2, 6 ) );
}
```

memcpy（拷贝内存内容）

定义函数：void * memcpy(void * dest, const void *src, size_t n);

memcpy()用来拷贝 src 所指的内存内容前 n 个字节到 dest 所指的内存地址上。与 strcpy()不同的是，memcpy()会完整的复制 n 个字节，不会因为遇到字符串结束'\0'而结束。memcpy()函数可以拷贝任意类型的数据。memcpy()函数返回指向 dest 的指针。指针 src 和 dest 所指的内存区域不可重叠。在拷贝字符串时，通常都使用 strcpy()函数；在拷贝其它数据(例如结构)时，通常都使用 memcpy()函数。

memmove（拷贝内存内容）

定义函数：void *memmove(void *dest, const void *src, size_t n);

memmove()与 memcpy()一样都是用来拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址上。不同的是，当 src 和 dest 所指的内存区域重叠时，memmove()仍然可以正确的处理，不过执行效率上会比使用 memcpy()略慢些。该函数返回指向 dest 的指针。

```
#include<string.h>

int main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcpy() :%s\n", a );
    printf("after strcpy() :%s\n", strcpy( a, b ) );

    a[30]="string(1)";
    b[]="string(2)";
    printf("before strncpy() : %s\n", a );
    printf("after strncpy() : %s\n", strncpy( a, b, 6 ) );
}
```

memccpy（拷贝内存内容）

定义函数：void * memccpy(void *dest, const void *src, int c, size_t n);

memccpy()用来拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址上。与 memcpy()不同的是，memccpy()会在复制时检查参数 c 是否出现，若是则返回 dest 中值为 c 的下一个字节地址。该函数返回指向 dest 中值为 c 的下一个字节指针。返回值为 NULL 表示在 src 所指内存前 n 个字节中没有值为 c 的字节。

```
#include<string.h>
#include<stdio.h>

int main()
{
    char a[]="string(a)";
    char b[]="string(b)";
    char *p;

    p = ( char * )memccpy( a, b, 'k', sizeof( b ) );

    if( p == NULL )
    {
        //注意 p 为 NULL 的情况，这时不能读取 p 所指的地方的内容
        printf("the return pointer of mymccpy is null !\n");
    }
    else
    {
        printf("memccpy(): %s, *p = %c\n", a, *p );
    }
}
```

bcopy（拷贝内存内容）

定义函数：void bcopy (const void *src,void *dest ,int n);

bcopy()与 memcpy()一样都是用来拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址，不过参数 src 与 dest 在传给函数时是相反的位置。建议使用 memcpy()取代 。

字符串和内存数据比较函数：

strcmp（比较字符串）

int strcmp(const char *s1, const char *s2);

strcmp()用来比较参数 s1 和 s2 字符串。字符串大小的比较是以 ASCII 码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将 s1 第一个字符值减去 s2 第一个字符值，若差值为 0 则再继续比较下个字符，若差值不为 0 则将差值返回。例如字符串"Ac"和"ba"比较则会返回字符"A"(65)和'b'(98)的差值(－33)。 若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值。

strcoll（采用目前区域的字符排列次序来比较字符串）

int strcoll(const char *s1, const char *s2);

strcoll()会依环境变量 LC_COLLATE 所指定的字符排列次序来比较 s1 和 s2 字符串。若参数 s1 和 s2 字符串相同则返回 0，s1 若大于 s2 则返回大于 0 的值，s1 若小于 s2 则返回小于 0 的值。若 LC_COLLATE 为"POSIX"或"C"，则 strcoll()与 strcmp()作用完全相同。

strcasecmp（忽略大小写比较字符串）

int strcasecmp (const char *s1, const char *s2);

strcasecmp()用来比较参数 s1 和 s2 字符串，比较时会自动忽略大小写的差异。若参数 s1 和 s2 字符串相同则返回 0，s1 长度大于 s2 长度则返回大于 0 的值，s1 长度若小于 s2 长度则返回小于 0 的值。

strncasecmp（忽略大小写比较字符串）

int strncasecmp(const char *s1, const char *s2, size_t n);

strncasecmp()用来比较参数 s1 和 s2 字符串前 n 个字符，比较时会自动忽略大小写的差异。若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值，s1 若小于 s2 则返回小于 0 的值。

strcmpi()或 stricmp()：对两个字符串进行大小写不敏感的比较。

strncmp()：对两个字符串的一部分进行大小写敏感的比较。

strnicmp()：对两个字符串的一部分进行大小写不敏感的比较。

```
#include <stdio.h>
#include <string.h>

int main()
{
    char *a="aBcDeF";
    char *b="AbCdEf";
    char *c="aacdef";
    char *d="ABCDEF";

    printf("strcmpi(a,b) : %d\n",strcmpi(a,b));
    printf("stricmp(a,b) : %d\n",stricmp(a,b));
    printf("strcmp(a,d) : %d\n",strcmp(a,d));

    printf("strncmp(a,b,3) : %d\n",strncmp(a,b,3));
    printf("strnicmp(a,b,3) : %d\n",strnicmp(a,b,3));

    getchar();
return 0;
}
```

memcmp（比较内存内容）

int memcmp(const void *s1, const void *s2, size_t n);

memcmp()用来比较 s1 和 s2 所指的内存区间前 n 个字符。字符串大小的比较是以 ASCII 码表上的顺序来决定。memcmp()首先将 s1 第一个字符值减去 s2 第一个字符的值，若差为 0 则再继续比较下个字符，若差值不为 0 则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(－33)。若参数 s1 和

s2 所指的内存内容都完全相同则返回 0 值。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值。

bcmp（比较内存内容）

int bcmp (const void *s1,const void * s2,int n);

bcmp()用来比较 s1 和 s2 所指的内存区间前 n 个字节，若参数 n 为 0，则返回 0。若参数 s1 和 s2 所指的内存内容都完全相同则返回 0 值，否则返回非零值。 建议使用 memcmp()取代。

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *a ="aBcDeF";
    char *b="AbCdEf";
    char *c="aacdef";
    char *d="aBcDeF";
    printf("memcmp(a,b):%d\n",memcmp((void*)a,(void*) b,6));
    printf("memcmp(a,c):%d\n",memcmp((void*)a,(void*) c,6));
    printf("memcmp(a,d):%d\n",memcmp((void*)a,(void*) d,6));
}
```

连接字符串的函数：

strcat（连接两字符串）

```
char *strcat (char *dest, const char *src );
```

strcat()会将参数 src 字符串拷贝到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要拷贝的字符串。该函数返回参数 dest 的字符串起始地址(strcat 函数要求 dest 参数原先已经包含了一个字符串，该字符串可以是空字符串。它找到这个字符串的末尾，并把 src 字符串的一份拷贝添加到这个位置。如果 src 和 dest 的位置发生重叠，其结果是未定义的)。

strncat（连接两字符串）

```
char * strncat( char *dest, const char *src, size_t n );
```

strncat()会将参数 src 字符串拷贝 n 个字符到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要拷贝的字符串。该函数返回参数 dest 的字符串起始地址。

```
#include<stdio.h>
#include<string.h>
int main()
{
    //strcat
    char a[30]="string1ABCDE";
    char b[]="STRING2";
    printf( "before strcat() : %s\n",a );
    printf( "after strcat()   : %s\n",strcat( a, b ) );

    //strncat
    char a1[30]="string1";
    char b1[]="STRING2";
    printf( "before strnact(): %s\n", a1 );
    printf( "after strncat() : %s\n", strncat( a1, b1, 6 ) );

    getchar();
    return 0;
}
```

查找字符/字符串的函数：

strstr（在一字符串中查找指定的字符串）

char *strstr(const char *s1, const char *s2);

strstr()会从字符串 s1 中搜寻字符串 s2，并将第一次出现的地址返回。返回指定字符串第一次出现的地址，否则返回 NULL(如果 s1 或者 s2 为空，在 vc 中编译没有问题，当时运行时会出现异常)。

```
#include<stdio.h>
#include<string.h>

//标准库中不存在 strrstr 函数，下面是一个自己编写的查找字符串 s1 中最后一次出现的字符串 s2 的位置的函数
char * my_strrstr( char const * s1, char const * s2 )
{
      char *last;
      char *current;

      //把指针初始化为我们已经找到的前一次匹配位置
      last = NULL;

      //只有在两个字符串都不为空时才进行查找,如果其中一个为空，返回 NULL
      if( s1 != NULL && s2 != NULL )
      {
            //查找 s2 在 s1 中第一次出现的位置
            current = strstr( s1, s2 );

            //我们每次找到字符串时，让指针指向它的起始位置，然后查找该字符串下一个匹配位置
            while( current != NULL )
            {
                  last = current;
                  current = strstr( last + 1, s2 );
            }
      }

      return last;
}

int main()
{
    char *str1="this is a example,example,example";
    char *str2="example";
    printf( "my_strrstr( str1, str2 )的返回结果为：%s\n", my_strrstr( str1, str2 ) );

    getchar();
      return 0;
}
```

strchr（查找字符串中第一个出现的指定字符）

char *strchr(const char *s, int c);

strchr()用来找出参数 s 字符串中第一个出现的参数 c 地址，然后将该字符出现的地址返回。如果找到指定的字符则返回该字符所在地址，否则返回 NULL。

strrchr（查找字符串中最后出现的指定字符）

char *strrchr(const char *s, int c);

strrchr()用来找出参数 s 字符串中最后一个出现的参数 c 地址，然后将该字符出现的地址返回。如果找到指定的字符则返回该字符所在地址，否则返回 NULL。

memchr（在某一内存范围中查找一特定字符）

void *memchr(const void *s, int c, size_t n);

memchr()从头开始搜寻 s 所指的内存内容前 n 个字节，直到发现第一个值为 c 的字节，则返回指向该字节的指针。如果找到指定的字节则返回该字节的指针，否则返回 NULL。

```
#include<stdio.h>
#include<string.h>
int main()
{
    char *s = "01234567890123456789012345678890";
    char *p;

    p = strchr(s,'9');
    printf(" strchr(): %s\n",p);

    p = strrchr(s,'9');
    printf("strrchr(): %s\n",p);

    p = strstr(s,"901");
    printf(" strstr(): %s\n",p);

    p = (char *)memchr( s, '9', 10 );
    printf(" memchr(): %s\n",p);
}
```

其它相关的函数：

index（查找字符串中第一个出现的指定字符）

char * index(const char *s, int c);

index()用来找出参数 s 字符串中第一个出现的参数 c 的地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。如果找到指定的字符则返回该字符所在地址，否则返回 NULL。[用 Dev-C++4.9.9.2 调试时，提示 index 为定义，在 linux 下确认一下]

rindex（查找字符串中最后一个出现的指定字符）

char * rindex(const char *s, int c);

rindex()用来找出参数 s 字符串中最后一个出现的参数 c 的地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。如果找到指定的字符则返回该字符所在的地址，否则返回 NULL。[用 Dev-C++4.9.9.2 调试时，提示 rindex 为定义，在 linux 下确认一下]

strlen（返回字符串长度）

size_t strlen(const char *s);

strlen()用来计算指定的字符串 s 的长度，不包括结束字符"\0"，该函数返回字符串 s 的字符数。

strdup（复制字符串）

char *strdup(const char *s);

strdup()会先用 maolloc()配置与参数 s 字符串相同的空间大小,然后将参数 s 字符串的内容复制到该内存地址，然后把该地址返回。该地址最后可以利用 free()来释放。函数返回一字符串指针，该指针指向复制后的新字符串地址，若返回 NULL 表示内存不足。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main()
{
    char a[] = "strdup";
    char *b;
    b = strdup( a );
    printf("b[]=\"%s\"\n", b );
    free(b);
return 0;
}
```

bzero（将一段内存内容全清为零）

void bzero(void *s,int n);

bzero()会将参数 s 所指的内存区域前 n 个字节，全部设为零值。相当于调用 memset((void*)s,0,size_tn);建议使用 memset 取代该函数。

memset（将一段内存空间填入某值）

void * memset (void *s ,int c, size_t n);

memset()会将参数 s 所指的内存区域前 n 个字节以参数 c 填入，然后返回指向 s 的指针。在编写程序时，若需要将某一数组作初始化，memset()会相当方便。附加说明：参数 c 虽声明为 int，但必须是 unsigned char，所以范围在 0 到 255 之间。

```c
#include<string.h>
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char s[30];
    memset (s,'A',sizeof(s));
    s[29]='\0';

    for( int i = 0; i < 30; i++ )
    {
        printf("%d ", s[i] );
    }
    printf("\n%s\n",s);

    memset (s,'\0',sizeof(s));
    for( int i = 0; i < 30; i++ )
    {
        printf("%d ", s[i] );
    }
    printf("\n\n%s\n",s);
    return 0;
}
```

输出结果为：

```
65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

strspn（返回字符串中连续含指定字符串内容的字符数）

```
size_t strspn( const char *s, const char *accept );
```

strspn()从参数 s 字符串的开头计算连续的字符，而这些字符都完全是 accept 所指字符串中的字符。简单的说，若 strspn()返回的数值为 n，则代表字符串 s 开头连续有 n 个字符都是属于字符串 accept 内的字符。该函数返回字符串 s 开头连续包含字符串 accept 内的字符数目。

strcspn（返回字符串中连续不含指定字符串内容的字符数）

```
size_t strcspn( const char *s, const char *reject );
```

strcspn()从参数 s 字符串的开头计算连续的字符，而这些字符都完全不在参数 reject 所指的字符串中。简单地说，若 strcspn()返回的数值为 n，则代表字符串 s 开头连续有 n 个字符都不含字符串 reject 内的字符。该函数返回字符串 s 开头连续不含字符串 reject 内的字符数目。

```c
#include<stdio.h>
#include<string.h>

int main()
{
    char *str="Linux was first developed for 386/486-based PCs.";
    char *t1="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    printf( "strspn:\n%d\n",strspn( str,t1 ) );

    printf( "strcspn: %d\n",strcspn( str, " " ) );
    printf( "strcspn: %d\n",strcspn( str, "/-" ) );
    printf( "strcspn: %d\n",strcspn( str, "1234567890" ) );
}
```

strpbrk（查找字符串中第一个出现的指定字符）

```
char *strpbrk( const char *s, const char *accept );
```

strpbrk()用来找出参数 s 字符串中最先出现存在于参数 accept 字符串中的任意字符。如果找到指定的字符则返回该字符所在地址，否则返回 NULL。

```
#include<stdio.h>
#include<string.h>

int main()
{
    char *str1=" 46545641321 this is a example";
char *str2="aeiouAEIOU";
// strpbrk( str1, str2 )的结果是返回 str1 中第一个 i 字符的地址；因为 i 是 str1 中第一个存在于 str2 中的字符
    printf( "strpbrk( str1, str2 )的返回结果为：%s\n", strpbrk( str1, str2 ) );

    getchar();
return 0;
}
```

strtok（分割字符串）

char * strtok(char *s, const char *delim);

strtok()用来将字符串分割成一个个片段。参数 s 指向欲分割的字符串，参数 delim 则为分割字符串，当 strtok()在参数 s 的字符串中发现参数 delim 的分割字符时则会将该字符改为\0 字符。在第一次调用时，strtok()必需给予参数 s 字符串，往后的调用则将参数 s 设置成 NULL。每次调用成功则返回下一个分割后的字符串指针。如果已无从分割则返回 NULL。

函数 strtok()具有“破坏性”，它会在原字符串中插入 NUL 字符(如果原字符串还要做其它的改变，应该拷贝原字符串，并将这份拷贝传递给函数 strtok())。函数 strtok()是不能“重新进入”的，你不能在一个信号处理函数中调用 strtok()函数，因为在下一次调用 strtok()函数时它总是会“记住”上一次被调用时的某些参数。strtok()函数是一个古怪的函数，但它在分解以逗号或空白符分界的数据时是非常有用的。

```
#include<stdio.h>
#include<string.h>

static char s0[] = "Now is the time for all good men . . . " ;
static char s1[] = "abcde--ABCDE--fghij--FGHIJ";
int main()
{
    char *p;
    p = strtok( s0, " ");
    while( p )
    {
        printf( "%s\n", p );
        p = strtok( NULL, " " );
    }

    p = strtok( s1, "--");
    while( p )
    {
        printf( "%s\n", p );
        p = strtok( NULL, "--" );
    }

    getchar();
    return 0;
}

输出结果为:
Now
is
the
time
for
all
good
men
.
.
```

```
.
abcde
ABCDE
fghij
FGHIJ
```

qsort（）：

C 标准库函数 qsort()排序算法使用起来比较方便，理由有以下三点：该函数是现成的；该函数是已通过调试的；该函数通常是已充分优化过的。qsort()函数通常使用快速排序法。以下是 qsort()函数的原型：

void qsort(void *buf，size_t num，size_t size，int(*comp)(const void *ele1，const void *ele2));

qsort()函数通过一个指针指向一个数组(buf)，该数组的元素为用户定义的数据，数组的元素个数为 num，每个元素的字节长度都为 size。数组元素的排序是通过调用指针 comp 所指向的一个函数来实现的，该函数对数组中由 ele1 和 ele2 所指向的两个元素进行比较，并根据前者是小于、等于或大于后者而返回一个小于、等于或大于 0 的值[buf 里面的元素必须是等长的，不能用 qsort 对变长字符串数组进行排序？]

```
#include<stdio.h>
#include<stdlib.h>

//对 double 类型的数组进行排序
int doublecompare( const double * i, const double * j )
{
    if( *i > *j )
        return 1;
    else if( *i < *j )
        return -1;
    else
        return 0;
}

int main()
{
    double base1[]={1.2, 2.6, 3.4, 8.6, 14.3, 0.2, 7.9, 1.6, 9.2, 10.8, 5.55 };
    int i, len1 = sizeof(base1)/sizeof(base1[0]);

    for( i = 0; i < len1; i++ )
    {
       printf("%5.2f, ", base1[i]);
    }

    printf("\n");

    qsort( base1, len1, sizeof(base1[0]), (int(*)(const void *, const void *))doublecompare );

    for( i=0; i < len1; i++)
    {
       printf("%5.2f, ", base1[i]);
    }
    return 0;
}
```

尽管有 qsort()函数，但程序员经常还要自己编写排序算法程序，其原因有这样几点：第一，在有些异常情况下，qsort()函数的运行速度很慢，而其它算法程序可能会快得多；第二，qsort()函数是为通用的目的编写的，这给它带来了一些不利之处，例如每次比较时都要通过用户提供的一个函数指针间接调用一个函数；第三，qsort()函数要求所有数据都在同一个数组中，而在实际应用中，数据的长度和性质千变万化，可能很难甚至无法满足这一要求；第四，qsort()函数通常不是一种稳定的排序方法。

bsearch()：

C 标准库函数 bsearch()是最方便的查找方法。bsearch()函数的原型如下：

void *bsearch(const void *key，const void *buf，size_t num，size_t size，

int (*comp) (const void *，const void *))；

bsearch()函数在一个数据已经排好序的数组中进行折半查找。折半查找也是一种分割处理式的算法。bsearch()函数的查找过程为：首先比较关键字与数组中间的元素，如果两者相等，则查找结束；如果前者比后者小，则要查找的数据必然在数组的前半部，此后只需在数组的前半部中继续进行折半查找，如果前者比后者大，则要查找的数据必然在数组的后半部，此后只需在数组的后半部中继续进行折半查找。找到关键字则返回该关键字的地址，否则返回 NULL**[为什么 bsearch 函数的常数里面还包含一个比较函数呢，在比较前，不是已经要求数组有序了吗？]**

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int cmp(const void *a,const void *b)
{
    return (strcmp((char *)a,(char *)b));
}

main()
{
    char data[][80]={ "linux", "freebsd", "solaris", "sunos", "windows", "ubuntu", "redhat" };
    char key[80], *base , *offset;
    int num, size;

    num = sizeof(data)/sizeof(data[0]);
    size = sizeof(data[0]);
    printf(" %d,  %d \n", num, size );

    while(1)
    {
        printf(">");
        fgets( key, sizeof(key), stdin );
        key[strlen(key)-1]='\0';

        if ( !strcmp( key, "exit" ) )
        {
            break;
        }

        base = data[0];
        qsort( base, num, size, cmp );
        offset = (char *) bsearch( key, base, num, size, cmp );

        if( offset == NULL )
        {
            printf("%s not found!\n",key);
        }
        else
        {
             printf("found: %s \n",offset);
        }
    }
    getchar();
}
```

lsearch（线性搜索）：

void *lsearch(const void * key, const void * base, size_t * nmemb, size_t size,

int (*compar) (const void *, const void *));

lsearch()利用线性搜索在数组中从头至尾一项项查找数据。参数 key 指向欲查找的关键数据，参数 base 指向要被搜索的数组开头地址，参数 nmemb 代表数组中的元素数量，每一元素的大小则由参数 size 决定，最后一项参数 compar 为一函数指针，这个函数用来判断两个元素是否相同，若传给 compar 的第一个参数所指的元素数据和第二个参数所指的元素数据相同时则返回 0，两个元素数据不相同则返回非 0 值。如果 lsearch（）找不到关键数据时会主动把该项数据加入数组里。

返回值: 找到关键数据则返回找到的元素的地址，如果在数组中找不到关键数据则将此关键数据加入数组，再把加入数组后的地址返回。

lfind（线性搜索）:

void *lfind (const void *key, const void *base, size_t *nmemb, size_t size,
int(* compar) (const void * ,const void *));

lfind()利用线性搜索在数组中从头至尾一项项查找数据。参数 key 指向欲查找的关键数据，参数 base 指向要被搜索的数组开头地址，参数 nmemb 代表数组中的元素数量，每一元素的大小则由参数 size 决定，最后一项参数 compar 为一函数指针，这个函数用来判断两个元素是否相同，若传给 compar 的异地个参数所指的元素数据和第二个参数所指的元素数据相同时则返回 0，两个元素数据不相同则返回非 0 值。Lfind()与 lsearch()不同点在于，当找不到关键数据时 lfind()仅会返回 NULL，而不会主动把该笔数据加入数组尾端。

srand（设置随机数种子）:

表头文件：#include<stdlib.h>
void srand(unsigned int seed);

srand()用来设置 rand()产生随机数时的随机数种子。参数 seed 必须是个整数，通常可以利用 geypid()或 time(0)的返回值来当做 seed。如果每次 seed 都设相同值，rand()所产生的随机数值每次就会一样。

rand（产生随机数）:

int rand(void);

rand()会返回一随机数值，范围在 0 至 RAND_MAX 间**(RAND_MAX 定义在 stdlib.h，其值为 2147483647)** 。在调用此函数产生随机数前，必须先利用 srand()设好随机数种子，如果未设随机数种子，rand()在调用时会自动设随机数种子为 1。关于随机数种子请参考 srand()。

```
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

int main()
{
    srand( (int) time(0) );
    for( int i = 0; i < 200; i++ )
    {
        if( i % 10 == 0 )
        {
            printf("\n");
        }
        /*产生[1, 10]之间的随机数*/
        printf("%d\t", rand() % 10 + 1 );
```

```
    }
    getchar();
    return 0;
}
```

什么是标准预定义宏?

ANSIC 标准定义了以下 6 种可供 C 语言使用的预定义宏：

宏 名	作 用
__LINE__	在源代码中插入当前源代码行号
__FILE__	在源代码中插入当前源代码文件名
__DATE__	在源代码中插入当前编译日期〔注意和当前系统日期区别开来〕
__TIME__	在源代码中插入当前编译时间〔注意和当前系统时间区别开来〕
__STDC__	当要求程序严格遵循 ANSIC 标准时该标识符被赋值为 1。

标识符__LINE__和__FILE__通常用来调试程序;标识符__DATE__和__TIME__通常用来在编译后的程序中加入一个时间标志，以区分程序的不同版本；当要求程序严格遵循 ANSIC 标准时，标识符__STDC__就会被赋值为 1；当用 C++编译程序编译时，标识符__cplusplus 就会被定义。

```
#include <stdio.h>

int main ()
{
    printf("该输出行在源程序中的位置：%d\n", __LINE__ );
    printf("该程序的文件名为：%s\n", __FILE__ );
    printf("当前日期为：%s\n", __DATE__ );
    printf("当前时间为：%s\n", __TIME__ );

    return 0;
}
```

断言 assert(表达式) 相关概念(c 和指针. P342.)

断言就是声明某种东西应该为真。ANSI C 实现了一个 **assert** 宏，它在调试程序时很有用。它的原型如下所示：
void assert(int expression);

当它被执行时，这个宏对表达式参数进行测试。如果它的值为假，它就向标准错误打印一条诊断信息并终止程序。这条信息的格式是由编译器定义的，但它将包含这个表达式所在的源文件的名字以及断言所在的行号。如果表达式为真，它不打印任何东西，程序继续执行。用这种方法使用断言使调试程序变得更容易，因为一旦出现错误，程序就会停止。而且，这条错误提示信息会准确地提示了症状出现的地点。

这个宏提供了一种方便的方法，对应该为真的东西进行检验。例如：如果一个函数必须使用一个不能为 NULL 的指针参数进行调用，那么函数可以用断言验证这个值：“assert(value != NULL);” 如果函数错误的接受了一个 NULL 参数，程序就会打印一条类似下面形式的信息：
Assertion failed: != NULL, file.c line 214

当程序被完整的测试完毕之后，你可以在编译时通过定义 NDEGUG 消除所有的断言。你可以使用-DNDEBUG 编译器命令行选项 或者 在源文件中头文件 assert.h 被包含之前增加下面这个定义：
#define NDEBUG
当 NDEBUG 被定义之后，预处理器将丢弃所有的断言，这样就消除了这方面的开销，而不必从源文件中把所有的断言实际删除。

```
#include<stdio.h>
#include<string.h>
//#define NDEBUG
#include<assert.h>


void my_strcopy( char * dest,  char * src )
{
    assert( src  != NULL );
    assert( dest != NULL );
    assert( strlen(src) >= strlen(dest) );

    while( (*dest++ = *src++) != '\0' );
}

int main()
{
    char str1[] = "0123456789";
    char str2[] = "abcdefghijk";

    printf("    原始字符串：\n%s\n%s\n",str1,str2);
    my_strcopy( str1, str2 );
    printf("拷贝后的字符串：\n%s\n%s\n",str1,str2);

    getchar();
    return 0;
}
```

连接运算符“##”和字符串化运算符"#"有什么作用?[仔细理解]

连接运算符“##”可以把两个独立的字符串连接成一个字符串。在 C 的宏中，经常用到“##”运算符，请看下例：

```
#include<stdio.h>
#define SORT(X)  sort_function ## X
void main(void)；
void main(void)
{
   char *array；
   int  elements，element_size；
   SORT(3) (array，elements，element_size)；
}
```

在该例中，宏 SORT 利用“##”运算符把字符串 sort_function 和经参数 x 传递过来的字符串连接起来，这意味着语句 SORT(3)(array，elemnts，element_size) 将被预处理程序转换为语句：

```
sort_function3(array，elements，element_size)；
```

从宏 SORT 的用法中你可以看出，如果在运行时才能确定要调用哪个函数，你可以利用“##”运算符动态地构造要调用的函数的名称。

```
#include<stdio.h>
#define FUNC(X)  func ## X

int func1(int a, int b)
{
    return 1;
}

int func2(int a, int b)
{
    return 2;
}

int main()
{

    int  a = 0, b = 0, result;
    result = FUNC(1)(a, b);
    printf("---%d---\n", result);
    result = FUNC(2)(a, b);
    printf("---%d---\n", result);

    getchar();
    return 0;
}
```

字符串化运算符"#"能将宏的参数转换为带双引号的字符串，请看下例：

```
define DEBUG_VALUE(v)  printf(#v"is equal to %d. \n", v)
```

你可以在程序中用 DEBUG_VALUE 宏检查变量的值，请看下例:

```
int x=20;
DEBUG_VALUE(x);
```

上述语句将在屏幕上打印"x is equal to 20"。这个例子说明，宏所使用的“#”运算符是一种非常方便的调试工具。

```
#include <stdio.h>

#define DEBUG_VALUE_INT(v)   printf(#v" is equal to %d.\n",v )
#define DEBUG_VALUE_STR(v)   printf(#v" is equal to %s.\n",v )

int main(int)
{
	int x = 0;
	char str[] = "asdfqweqwfdvasdf";

	DEBUG_VALUE_INT(x);
	DEBUG_VALUE_STR(str);
	getchar();
	return 0;
}
```

注释掉一段代码的方法

在有些语言中，注释有时用于把一段代码“注释掉”，也就是使这段代码在程序中不起作用，但并不将其真正从源文件中删除。在C语言中，这可不是个好主意，如果你试图在一段代码的首尾分别加上/*和*/符号来“注释掉”这段代码，你不一定能如愿。如果这段代码内部原先就有注释存在，这样做就会出问题。要从逻辑上删除一段C代码，更好的办法是使用#if指令。只要像下面这样使用：

```
#if 0
        statements
#endif
```

在#if和#endif之间的程序段就可以有效地从程序中去除，即使这段代码之间原先存在注释也无妨，所以这是一种更为安全的方法。预处理指令的作用远比你想象的要大

```
#include <stdio.h>
#include <stdlib.h>

#define DEL_CODE

int main()
{
    int i = 0;

    //删除一段代码的方法 1
   #ifndef DEL_CODE
   for( i = 0; i < 10; i++ )
    {
        printf("this is a example\n");
    }
   #endif

    //删除一段代码的方法 2
   #if 0
   for( i = 0; i < 10; i++ )
    {
        printf("this is anther example\n");
    }
   #endif

    return 0;
}
```

Typedef 相关概念

C 语言支持一种叫做 typedef 的机制，它允许你为各种数据类型定义新的名字。typedef 声明的写法和普通的声明基本相同，只是让 typedef 这个关键字出现在声明的前面。例如，下面这个声明：

```
char *ptr_to_char;
```

把变量 ptr_to_char 声明为一个指向字符的指针。但是，在添加了关键字 typedef 后，声明变为：

```
typedef char *ptr_to_char;
```

这个声明把标示符 ptr_to_char 作为指向字符的指针类型的新名字。你可以像使用任何预定义名字一样在声明中使用这个新名字，例如：

```
ptr_to_char a;  //声明 a 是一个指向字符的指针。
```

使用 typedef 声明类型可以减少使声明变得又臭又长的危险，尤其是那些复杂的声明。而且，如果你以后觉得应该修改程序使用的一些数据的类型时，修改一个 typedef 声明比修改程序中与这种类型有关的所有变量(和函数)的所有声明要容易得多。

一般情况下，typedef 用于简洁地表示指向其他东西的指针。典型的例子是 signal()原型的声明。Signal()是一种系统调用，用于通知运行时系统，当某种特定的软件中断发生时调用特定的程序。调用 signal()时，通过参数传递告诉它中断的类型以及用于处理中断的程序。在 ANSI C 标准中，signal 的声明如下：

```
void ( *signal( int sig, void( *func )( int ) ) )( int );
```

让我们分析一下这个函数：

```
void( *signal(  •••••  ) )( int );
```

signal 是一个函数指针，其所指向的函数接受一个 int 参数并返回 void。signal 其中的一个参数为：

```
void( *func )( int )
```

它表示一个函数指针，所指向的函数接受一个 int 参数并返回 void。现在，让我们看一下怎样用 typedef 来代表通用部分，从而简化该声明：

```
//ptr_to_func 是一个函数指针，该函数接受一个整型参数，返回值为 void
typedef void(*ptr_to_func)(int);
//signal 是一个函数，它接受两个参数，其中一个是 int，另一个是 ptr_to_func，返回值是 ptr_to_func
ptr_to_func signal( int, ptr_to_func  );
```

typedef 和 define 的一些区别：

1、可以用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这么做，如下所示：

```
#define peach int
Unsigned peach i;  /*没问题*/
typedef int banana;
unsigned bababa i;  /*错误！*/
```

2、在连续几个变量的声明中，用 typedef 定义的类型能够保证声明中所有的变量均为同一种类型，而用#define 定义的类型则无法保证，如下所示：

```
#define int_ptr int *
int_ptr chalk, cheese;
```

经过宏扩展，第二行变为 ：

```
int * chalk, cheese;
```

这使得 chalk 和 cheese 成为不同的类型，chalk 是一个指向 int 的指针，而 cheese 则是一个 int。

```
typedef char * char_ptr;
char_ptr bentley, rolls_royce;
```

bentley 和 rolls_royce 的类型是相同的，都是指向 char 的指针。

= 不同于 ==（c 缺陷与陷阱 1.1 节）

当形如 e1 = e2 这样的表达式出现在语句的条件判断部分时，有些编译器会给出警告消息。当确实需要对变量进行赋值并检查该变量是否为 0 时，为了避免来自编译器的警告，我们不应该简单的关闭警告选项，而应该显示的进行比较。也就是说，下面的例子：

```
if( x = y )
    foo();
```

应该改写为：

```
if( ( x = y ) != 0 )
{
    foo();
}
```

这种写法使得代码的意图一目了然

词法分析中的“贪心法”（c 缺陷与陷阱 1.3 节）

C 语言的某些符号，例如/ 、* 、和=，只有一个字符长，称为单字符符号。而 C 语言中的其他符号，例如/*和 == ，以及标识符，包括了多个字符，称为多字符符号。当 C 编译器读入一个字符'/'后又跟了一个字符'*'，那么编译器就必须做出判断：是将其作为两个分别的符号对待，还是合起来作为一个符号对待。C 语言对这个问题的解决方案可以归纳为一个很简单的规则：每一个符号应该包含尽可能多的字符。也就是说，编译器将程序分解成符号的方法是，从左到右一个字符一个字符地读入，如果该字符可能组成一个符号，那么再读入下一个字符，判断已经读入的两个字符组成的字符串是否可能是一个符号的组成部分；如果可能，继续读入下一个字符，重复上述判断，直到读入的字符组成的字符串已不再可能组成一个有意义的符号。这个处理策略有时被称为“贪心法”，或者，更口语化一点，称为“大嘴法”。Kernighan 与 Ritchie 对这个方法的表述如下，“如果（编译器的）输入流截止至某个字符之前都已经被分解为一个个符号，那么下一个符号将包括从该字符之后可能组成一个符号的最长字符串。”

需要注意的是，除了字符串与字符常量，符号的中间不能嵌有空白（空格符、制表符和换行符）。例如，==是单个符号，而== 则是两个符号，下面的表达式

```
a---b
```

与表达式

```
a -- - b
```

的含义相同，而与

```
a - -- b
```

的含义不同。同样地，如果/是为判断下一个符号而读入的第一个字符，而/之后紧接着*，那么无论上下文如何，这两个字符都将被当作一个符号/*，表示一段注释的开始。

根据代码中注释的意思，下面的语句的本意似乎是用 x 除以 p 所指向的值，把所得的商再赋给 y：

```
y = x/*p        /* p 指向除数*/;
```

而实际上，/*被编译器理解为一段注释的开始，编译器将不断地读入字符，直到*/出现为止。也就是说，该语句直接将 x 的值赋给 y，根本不会顾及到后面出现的 p。将上面的语句重写如下：

```
y = x / *p      /* p 指向除数 */;
```

或者更加清楚一点，写作：

```
y = x/(*p)      /* p 指向除数 */;
```

这样得到的实际效果才是语句注释所表示的原意。

```
#include <stdio.h>

int main()
{
    int a = 1, b = 1, result1 = 0, result2 = 0, result3 = 0;

    a = b = 1;
    /*相当于 a-- - b 的形式*/
    result1 = a---b;

    a = b = 1;
    /*result2 = a----b; 这种形式通不过编译*/
    result2 = a--- -b;

    a = b = 1;
    /*result3 = a+++++b;这种形式通不过编译*/
    result3 = a++ + ++b;

    printf(" a---b = %d\n a--- -b = %d\n a++ + ++b = %d\n",result1, result2, result3);
    getchar();
}
```

输出结果：

```
a---b = 0
a--- -b = 2
a++ + ++b = 3
```

运算符的优先级问题（c 缺陷与陷阱 2.2 节）

优先级		运算符	结合律
从高到低排列		() [] -> .	从左至右
		!(非) ~(按位求反) ++ -- (type) sizeof * &	从右至左
	算术运算符	* / %	从左至右
		+ -	从左至右
	移位运算符	<<(左移) >>(右移)	从左至右
	关系运算符	< <= > >=	从左至右
		== !=	从左至右
	按位运算符	&(按位与)	从左至右
		^(按位异或)	从左至右
		|(按位或)	从左至右
	逻辑运算符	&&	从左至右
		||	从左至右
	条件运算符	?:	
	赋值操作符	assignments	从右至左
	逗号运算符	,	从左至右

优先级最高的其实并不是真正意义上的运算符，包括：数组下标、函数调用操作符、各结构成员操作符。它们都是自左向右结合，因此 a.b.c 的含义是(a.b).c，而不是 a.(b.c) 。

单目运算符的优先级仅次于前述运算符。在所有真正意义上的运算符中，它们的优先级最高。由于函数调用比一元运算符绑定得更紧密，你必须写(*p)()来调用 p 指向的函数；*p()表示 p 是一个返回一个指针的函数。类型转换也是单目运算符，并且和其他单目运算符一样。单目运算符是右结合的，因此*p++表示*(p++)，即取指针 p 所指的对象，然后 p 递增 1；而不是(*p)++，即取指针所指的对象，然后将该对象递增 1 。

六个关系运算符并不具有相同的优先级：==和!=的优先级比其他关系运算符要低。因此，如果我们要比较 a 和 b 的相对大小顺序是否和 c 与 d 的相对大小顺序一样，就可以写成：a < b == c < d 。

条件运算符的优先级仅仅比赋值运算符和逗号运算符高。首先计算第一个表达式，如果该表达式不等于 0，则结果为第二个表达式的值，否则为第三个表达式的值。第二个和第三个操作数中仅有一个操作数会被计算。如果第二个和第三个操作数为算术类型，则要进行普通的算术类型转换，以使它们的类型相同，该类型也是结果的类型。如果它们都是 void 类型，或者是同一类型的结构或联合，或者是指向同一类型的对象的指针，则结果的类型与这两个操作数的类型相同。如果其中一个操作数为指针，而另外一个是常量 0，这 0 将被转换为指针类型，且结果为指针类型。如果一个操作数为指向 void 类型的指针，而另外一个操作数为指向其他类型的指针，则指向其他类型的指针将被转换为指向 void 的指针，这也是结果的类型。因为条件运算符的优先级仅仅比赋值运算符和逗号运算符高，这就允许我们在条件运算符的条件表达式中包括关系运算符的逻辑组合，例如：`z = a < b && b < c ? d : e` 。

在所有的运算符中，逗号运算符的优先级最低。由逗号分隔的两个表达式的求值顺序为从左到右，并且左表达式的值被丢弃，右操作数的类型和值就是结果的类型和值。

变量的存储类型及初始化相关概念(c 和指针. P43.)〔了解〕

变量的存储类型(storage class)是指存储变量值的内存类型。变量的存储类型决定变量何时创建、何时销毁以及它的值将保持多久。有三个地方可以用于存储变量：普通内存、运行时堆栈、硬件寄存器。在这三个地方存储的变量具有不同的特性。

变量的缺省存储类型取决于它的声明位置。凡是在任何代码块之外声明的变量总是存储于静态内存中，也就是不属于堆栈的内存，这类变量称为静态(static)变量。对于这类变量，你无法为它们指定其他存储类型。静态变量在程序运行之前创建，在程序的整个执行期间始终存在。它始终保持原先的值，除非给它赋一个不同的值或者程序结束。

在代码块内部声明的变量的缺省存储类型是自动的(automatic)，也就是说它存储于堆栈中，称为自动(auto)变量。有一个关键字 auto 就是用于修饰这种存储类型的，但它极少使用，因为代码块中的变量在缺省情况下就是自动变量。在程序执行到声明自动变量的代码块时，自动变量才被创建，当程序的执行流离开该代码块时，这些自动变量便自行销毁。如果该代码块被数次执行，例如一个函数被反复调用，这些自动变量每次都将重新创建。在代码块再次执行时，这些自动变量在堆栈中所占据的内存位置有可能和原先的位置相同，也可能不同。即使它们所占据的位置相同，你也不能保证这块内存同时不会有其他的用途。因此，我们可以说自动变量在代码块执行完毕后就消失。当代码块再次执行时，它们的值一般并不是上次执行时的值。

对于在代码块内部声明的变量，如果给它加上关键字 static，可以使它的存储类型从自动变为静态。具有静态存储类型的变量在整个程序执行过程中一直存在，而不仅仅在声明它的代码块的执行时存在。注意，修改变量的存储类型并不表示修改该变量的作用域，它仍然只能在该代码块内部按名字访问。函数的形式参数不能声明为静态，因为实参总是在堆栈中传递给函数，用于支持递归。

最后，关键字 register 可以用于自动变量的声明，提示它们应该存储于机器的硬件寄存器而不是内存中，这类变量称为寄存器变量。通常，寄存器变量比存储于内存的变量访问起来效率更高。但是，编译器并不一定要理睬 register 关键字，如果有太多的变量被声明为 register，它只选取前几个实际存储于寄存器中，其余的就按普通自动变量处理。如果一个编译器自己具有一套寄存器优化方

法，它也可能忽略 register 关键字，其依据是由编译器决定哪些变量存储于寄存器中比人脑的决定更为合理一些。

初始化

现在我们把话题返回到变量声明中变量的初始化问题。自动变量和静态变量的初始化存在一个重要的差别。在静态变量的初始化中，我们可以把可执行程序文件想要初始化的值放在当程序执行时变量将会使用的位置。当可执行文件载入到内存时，这个已经保存了正确初始值的位置将赋值给那个变量。完成这个任务并不需要额外的时间，也不需要额外的指令，变量将会得到正确的值。如果不显式地指定其初始值，静态变量将初始化为 0。

自动变量的初始化需要更多的开销，因为当程序链接时还无法判断自动变量的存储位置。事实上，函数的局部变量在函数的每次调用中可能占据不同的位置。基于这个理由，自动变量没有缺省的初始值，而显式的初始化将在代码块的起始处插入一条隐式的赋值语句。

这个技巧造成 4 种后果。首先，自动变量的初始化较之赋值语句效率并无提高。除了声明为 const 的变量之外，在声明变量的同时进行初始化和先声明后赋值只有风格之差，并无效率之别。其次，这条隐式的赋值语句使自动变量在程序执行到它们所声明的函数（或代码块）时，每次都将重新初始化。这个行为与静态变量大不相同，后者只是在程序开始执行前初始化一次。第 3 个后果则是个优点，由于初始化在运行时执行，你可以用任何表达式作为初始化值，例如：

```
int
func( int a )
{
        int     b = a + 3;
```

最后一个后果是，除非你对自动变量进行显式的初始化，否则当自动变量创建时，它们的值总是垃圾。

左值和右值相关的概念(c 和指针. P79.)〔了解〕

为了理解有些操作符存在的限制，你必须理解**左值**(L-value)和**右值**(R-value)之间的区别。这两个术语是多年前由编译器设计者所创造并沿用至今，尽管它们的定义并不与 C 语言严格吻合。

左值就是那些能够出现在赋值符号左边的东西，右值就是那些可以出现在赋值符号右边的东西。这里有个例子：

```
a = b + 25;
```

a 是个左值，因为它标识了一个可以存储结果值的地点，b + 25 是个右值，因为它指定了一个值。

它们可以互换吗？

```
b + 25 = a;
```

原先用作左值的 a 此时也可以当作右值，因为每个位置都包含一个值。然而，b + 25 不能作为左值，因为它并未标识一个特定的位置。因此，这条赋值语句是非法的。

注意当计算机计算 b + 25 时，它的结果必然保存于机器的某个地方。但是，程序员并没有办法预测该结果会存储在什么地方，也无法保证这个表达式的值下次还会存储于那个地方。其结果是，这个表达式不是一个左值。基于同样的理由，字面值常量也都不是左值。

听上去似乎是变量可以作为左值而表达式不能作为左值，但这个推断并不准确。在下面的赋值语句中，左值便是一个表达式。

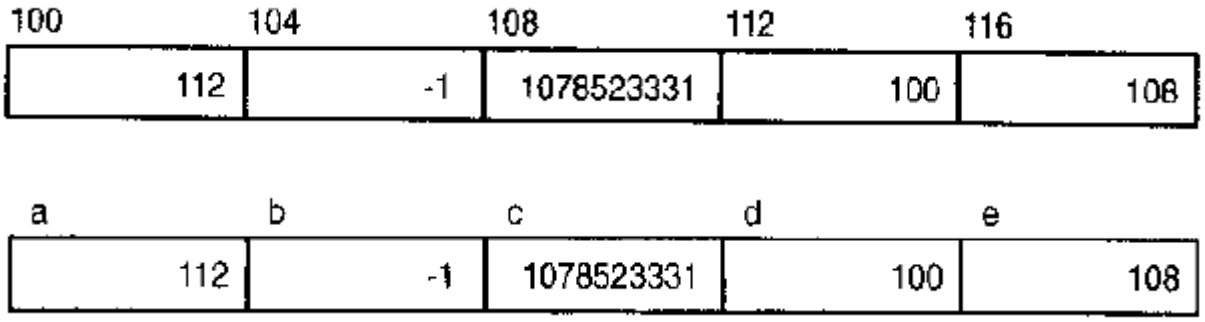
```
int     a[30];
...
a[ b + 10 ] = 0;
```

下标引用实际上是一个操作符，所以表达式的左边实际上是个表达式，但它却是一个合法的左值，因为它标识了一个特定的位置，我们以后可以在程序中引用它。这里有另外一个例子：

```
int     a, *pi;
...
pi = &a;
*pi = 20;
```

请看第 2 条赋值语句，它左边的那个值显然是一个表达式，但它却是一个合法的左值。为什么？指针 pi 的值是内存中某个特定位置的地址，*操作符使机器指向那个位置。当它作为左值使用时，这个表达式指定需要进行修改的位置。当它作为右值使用时，它就提取当前存储于这个位置的值。

变量的值和类型相关的概念(c 和指针. P92.)〔了解〕



现在让我们来看一下存储于这些位置的值。头两个位置所存储的是整数。第 3 个位置所存储的是一个非常大的整数，第 4、5 个位置所存储的也是整数。下面是这些变量的声明：

```
int     a = 112, b = -1;
float   c = 3.14;
int     *d = &a;
float   *e = &c;
```

在这些声明中，变量 a 和 b 确实用于存储整型值。但是，它声明 c 所存储的是浮点值。可是，在上图中 c 的值却是一个整数。那么到底它应该是哪个呢？整数还是浮点数？

答案是该变量包含了一序列内容为 0 或者 1 的位。它们可以被解释为整数，也可以被解释为浮点数，这取决于它们被使用的方式。如果使用的是整型算术指令，这个值就被解释为整数，如果使用的是浮点型指令，它就是个浮点数。

这个事实引出了一个重要的结论：**不能简单地通过检查一个值的位来判断它的类型。**为了判断值的类型（以及它的值），你必须观察程序中这个值的使用方式。考虑下面这个以二进制形式表示的32位值：

01100111011011000110111101100010

下面是这些位可能被解释的许多结果中的几种。这些值都是从一个基于Motorola 68000的处理器上得到的。如果换个系统，使用不同的数据格式和指令，对这些位的解释将又有所不同。

类　型	值
1个32位整数	1735159650
2个16位整数	26476和28514
4个字符	glob
浮点数	1.116533×10^{24}
机器指令	*bcg .+110* 和 *ble .+102*

这里，一个单一的值可以被解释为5种不同的类型。显然，值的类型并非值本身所固有的一种特性，而是取决于它的使用方式。因此，为了得到正确的答案，对值进行正确的使用是非常重要的。

当然，编译器会帮助我们避免这些错误。如果我们把c声明为float型变量，那么当程序访问它时，编译器就会产生浮点型指令。如果我们以某种对float类型而言不适当的方式访问该变量时，编译器就会发出错误或警告信息。现在看来非常明显，图中所标明的值是具有误导性质的，因为它显示了c的整型表示方式。事实上真正的浮点值是3.14。

怎样删去字符串尾部的空格?

C语言没有提供可删去字符串尾部空格的标准库函数，但是，编写这样的一个函数是很方便的。请看下例：

```
#include <stdio.h>
# include <string.h>

char * rtrim( char * );
int main()
{
    //char * trail_str = "0123456789          "; 把字符串定义成这种形式时，运行程序的时候会出现异常
    char trail_str[21] = "0123456789          ";

    printf( "Before calling rtrim(), trail_str is '%s'\n" , trail_str );
    printf( "and has a length of %d. \n" , strlen( trail_str ) );

    rtrim(trail_str);

    printf( "After calling rttim(), trail_ str is '%s'\n", trail_str );
    printf( "and has a length of %d. \n" , strlen( trail_str ) ) ;

    getchar();
    return 0;
}

/* The rtrim() function removes trailing spaces from a string. */
char * rtrim( char * str )
{
    int n = strlen(str) - 1;

      while( *( str + n ) == ' ' )
      {
            *( str + n-- ) = '\0';
      }

      return str;
}
```

在上例中，rtrim()是用户编写的一个函数，它可以删去字符串尾部的空格。函数 rtrim()从字符串中位于 null 字符前的那个字符开始往回检查每个字符，当遇到第一个不是空格的字符时，就将该字符后面的字符替换为 null 字符。因为在 C 语言中 null 字符是字符串的结束标志，所以函数 rtrim()的作用实际上就是删去字符串尾部的所有空格。

怎样删去字符串头部的空格?

C 语言没有提供可删去字符串头部空格的标准库函数，但是，编写这样的一个函数是很方便的。请看下例：

```
#include <stdio.h>
#include <string.h>

char *ltrim(char * );
char *rtrim(char * );
void main (void)
{
...
}

char * ltrim(char * str)
{
    strrev(str);     /* Call strrev to reverse the string. */
    rtrim(str);      /* Call rtrim to remvoe the "trailing" spaces. */
    strrev(str);     /* Restore the string's original order. */
    return str ;     /* Return a pointer to the string. */
}

char* rtrim(char* str)
{
...
}
```

在上例中，删去字符串头部空格的工作是由用户编写的ltrim()函数完成的，该函数调用了6.2的例子中的rtrim()函数和标准C库函数strrev()。ltrim()函数首先调用strrev()函数将字符串颠倒一次，然后调用rtrim()函数删去字符串尾部的空格，最后调用strrev()函数将字符串再颠倒一次，其结果实际上就是删去原字符串头部的空格。

怎样打印字符串的一部分?

使用printf()函数打印字符串的任意部分，请看下例：

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

int main()
{
    char * source_str = "THIS IS THE SOURCE STRING" ;

    /* Use printf() to print the first 11 characters of source_str. */
    printf("First 11 characters: ' %11.11s'\n" , source_str);

    /* Use printf() to print only the last 13 characters of source _str. */
    printf("Last 13 characters:'%13.13s'\n", source_str+(strlen(source_str)-13));
}
```

输出结果为：

```
First 11 characters: 'THIS IS THE'
Last 13 characters：'SOURCE STRING'
```

在上例中，第一次调用 printf()函数时，通过指定参数"%11.11s"，迫使 printf()函数只打印 11 个字符的长度，因为源字符串的长度大于 11 个字符，所以在打印时源字符串将被截掉一部分，只有头 11 个字符被打印出来。第二次调用 printf()函数时，它将源字符串的最后 13 个字符打印出来，其实现过程为：
(1)用 strlen()函数计算出 source_str 字符串的长度，即 strlen(source_str)。
(2)将 source_str 的长度减去 13(13 是将要打印的字符数)，得出 source_str 中剩余字符数。
(3)将 strlen(source_str)-13 和 source_str 的地址相加，得出指向 source_str 中倒数第 13 个字符的地址的指针；即 source_str+(strlen(source_str)-13)。这个指针就是 printf()函数的第二个参数。
(4)通过指定参数“%13. 13s”，迫使 printf()函数只打印 13 个字符的长度，其结果实际上就是打印源字符串的最后 13 个字符。

关于“S 格式符”的用法的简单说明：

(1) %ms ：输出的字符串占 m 列，如果字符串本身长度大于 m，则突破 m 的限制，将字符串全部输出；若串长度小于 m，则在左边补空格。
(2) %－ms：如果字符串长度小于 m，则在 m 列范围内，字符串向左靠，右补空格。
(3) %m.ns：输出占 m 列，但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列范围的右侧，左补空格。
(4) %－m.ns：其中的 m、n 的含义同上，n 个字符输出在 m 列范围的左侧，右补空格。如果 n>m，则 m 自动取 n 值，即保证 n 个字符正常输出。

结构的自引用(c 和指针. P198.)

在一个结构内部包含一个类型为该结构本身的成员是否合法呢？这里有一个例子，可以说明这个想法。

```
struct SELF_REF1{
        int a;
        struct SELF_REF1 b;
        int c;
}
```

这种类型的自引用是非法的，因为成员 b 是另一个完整的结构，其内部还将包含它自己的成员 b。这第 2 个成员又是另一个完整的结构，它还将包括它自己的成员 b。这样重复下去永无止境。这有点像永远不会终止的递归程序。但下面这个声明却是合法的，你能看出其中的区别吗？

```
struct SELF_REF2{
        int a;
        struct SELF_REF2 *b;
        int c;
}
```

这个声明和前面那个声明的区别在于 b 现在是一个指针而不是结构。编译器在结构的长度确定之前就已经知道指针的长度，所以这种类型的自引用是合法的。

如果你觉得一个结构内部包含一个指向该结构本身的指针有些奇怪，请记住它事实上所指向的是同一种类型的**不同**结构。更加高级的数据结构，如链表和树，都是用这种技巧实现的。每个结构指向链表的下一个元素或树的下一个分枝。

警告：

警惕下面这个陷阱：

```
typedef struct{
        int a;
        SELF_REF3 *b;
        int c;
} SELF_REF3;
```

这个声明的目的是为这个结构创建类型名 SELF_REF3。但是，它失败了。类型名直到声明的末尾才定义，所以在结构声明的内部它尚未定义。

解决方案是定义一个结构标签来声明 b，如下所示：

```
typedef struct SELF_REF3_TAG{
        int a;
        struct SELF_REF3_TAG *b;
        int c;
}SELF_REF3;
```

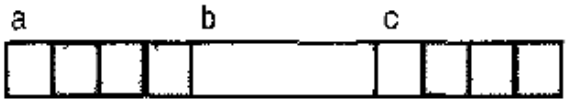
结构的存储分配(c 和指针. P206.)

结构在内存中是如何存储的呢？编译器按照成员列表的顺序一个接一个地给每个成员分配内存。只有当存储成员时需要满足正确的边界对齐要求时，成员之间才可能出现用于填充的额外内存空间。

为了说明这一点，考虑下面这个结构：

```
struct ALIGN{
        char a;
        int  b;
        char c;
};
```

如果某个机器的整型值长度为 4 个字节，并且它的起始存储位置必须能够被 4 整除，那么这个结构在内存中的存储将如下所示：



系统禁止编译器在一个结构的起始位置跳过几个字节来满足边界对齐要求，因此所有结构的起始存储位置必须是结构中边界要求最严格的数据类型所要求的位置。因此，成员 a（最左边的那个方框）必须存储于一个能够被 4 整除的地址。结构的下一个成员是一个整型值，所以它必须跳过 3 个字节（用灰色显示）到达合适的边界才能存储。在整型值之后是最后一个字符。

如果声明了相同类型的第 2 个变量，它的起始存储位置也必须满足 4 这个边界，所以第 1 个结构的后面还要再跳过 3 个字节才能存储第 2 个结构。因此，每个结构将占据 12 个字节的内存空间但实际只使用其中的 6 个，这个利用率可不是很出色。

你可以在声明中对结构的成员列表重新排列，让那些对边界要求最严格的成员首先出现，对边界要求最弱的成员最后出现。这种做法可以最大限度地减少因边界对齐而带来的空间损失。例如，下面这个结构

```
struct ALIGN1{
        int  b;
        char a;
        char c;
};
```

所包含的成员和前面那个结构一样，但它只占用 8 个字节的空间，节省了 33%。两个字符可以紧挨着存储，所以只有结构最后面需要跳过的两个字节才被浪费。

```
#include<stdio.h>
#include<stddef.h>

int main()
{
   struct ALIGN
    {
        char a;
        int  b;
        char c;
    };

    struct ALIGN1
    {
       int  b;
        char a;
        char c;
    };
```

```
    printf("      sizeof(struct ALIGN) = %d\n",sizeof(struct ALIGN) );
    printf(" offsetof(struct ALIGN, a) = %d\n", offsetof(struct ALIGN, a));
    printf(" offsetof(struct ALIGN, b) = %d\n", offsetof(struct ALIGN, b));
    printf(" offsetof(struct ALIGN, c) = %d\n", offsetof(struct ALIGN, c));

    printf("      sizeof(struct ALIGN1) = %d\n",sizeof(struct ALIGN1) );
    printf(" offsetof(struct ALIGN1, b) = %d\n", offsetof(struct ALIGN1, b));
    printf(" offsetof(struct ALIGN1, a) = %d\n", offsetof(struct ALIGN1, a));
    printf(" offsetof(struct ALIGN1, c) = %d\n", offsetof(struct ALIGN1, c));

   getchar();
    return 0;
}
```

运行结果为：

```
     sizeof(struct ALIGN) = 12
offsetof(struct ALIGN, a) = 0
offsetof(struct ALIGN, b) = 4
offsetof(struct ALIGN, c) = 8
     sizeof(struct ALIGN1) = 8
offsetof(struct ALIGN1, b) = 0
offsetof(struct ALIGN1, a) = 4
offsetof(struct ALIGN1, c) = 5
```

边界计算与不对称边界(C 缺陷与陷阱 3.6 节)

一个字符串中由下标为 16 到下标为 37 的字符元素所组成的子串，它的长度是多少呢？稍不留意，就会得到错误的结果 21。很自然的，人们会问这样一个问题：是否存在一些编程技巧，能够降低这类错误发生的可能性呢？

这个编程技巧不但存在，而且可以一言以蔽之：用第一个入界点和第一个出界点来表示一个数值的范围。具体而言，这个例子我们不说整数 x 满足边界条件 x>=16 且 x<=37，而是说整数满足边界条件 x>=16 且 x<38。注意，这里的下界是“入界点”，即包括在取值范围之中；而上界是“出界点”，即不包括在取值范围之中。这种不对称也许从数学上而言并不优美，但是它对程序设计的简化效果却足以令人吃惊：

1、取值范围的大小就是上界与下界的差。38-16 的值是 22，刚好是不对称边界 16 和 38 之间包括的元素数目。

2、如果取值范围为空，那么上界等于下界。

3、即使取值范围为空，上界也永远不可能小于下界。

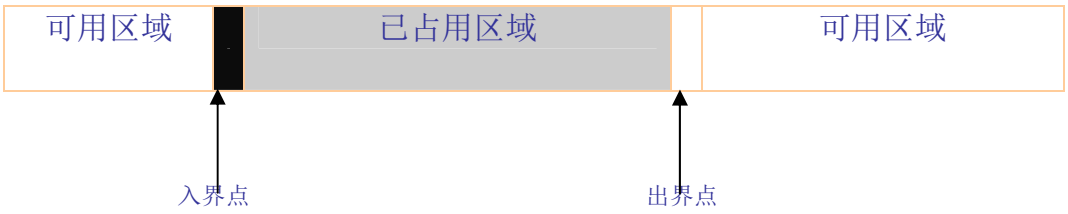
对于像 c 这样的数组下标从 0 开始的语言，不对称边界给程序设计带来的便利尤其明显：这种数组的上界（即第一个“出界点”）恰是数组元素的个数。因此，如果我们要在 c 语言中定义一个拥有 10 个元素的数组，那么 0 就是数组下标的第一个入界点，10 是数组下标中的第一个出界点。正因为如此，我们这样写：

```
int a[10], i;
for( i = 0; i < 10; i++ )
    a[i] = 0;
```

而不是写成下面这样：

```
int a[10], i;
for( i = 0; i <= 9; i++ )
    a[i] = 0;
```

另一种考虑不对称边界的方式是，把上界视作某序列中的第一个被占用的元素，而把下届视为序列中第一个被可以利用的元素，如下图所示：



整数溢出(C 缺陷与陷阱 3.9 节)

c 语言中存在两类整数算术运算，有符号运算与无符号运算。在无符号算术运算中，没有所谓的“溢出”一说：所有的无符号运算都是以 2 的 n 次方为模，这里的 n 是结果中的位数。如果算术运算符的一个操作数为有符号整数，另一个为无符号整数，那么有符号整数会被转换成无符号整数，“溢出”也不可能发生。但是，当两个操作数都是有符号数时，溢出就有可能发生，而且溢出的结果是未定义的。当一个运算的结果发生溢出时，做出任何假设都是不安全的。

假如 a 和 b 是两个非负整型变量，我们需要检查 a+b 是否会溢出，一种想当然的办法是这样：

```
if( a + b < 0 )
{
    complain();
}
```

这并不能正常运行。当 a+b 确实发生溢出时，所有关于结果如何的假设都不再可靠。例如，在某些机器上，加法运算将设置一个内部寄存器为四种状态之一：正、负、零、溢出。在这种机器上，c 编译器完全有理由这样来实现上面的例子，即 a 与 b 相加，然后检查内部寄存器的标志是否为“负”。当加法操作发生溢出时，这个内部寄存器的状态是溢出而不是负，那么 if 的语句的检查就会失败。

一种正确的方式是将 a 和 b 都强制转换为无符号整数：

```
if( ( unsigned )a + ( unsigned )b > INT_MAX )
{
    complain();
}
```

此处的 INT_MAX 是一个已定义的常量，代表可能的最大整数值。ANSI C 标准在<limits.h>中定义了 INT_MAX。不需要用到无符号算术运算的另外一种可行办法是：

```
if( a > INT_MAX - b )
{
    complain();
}
```

```
#include <limits.h>
#include <stdio.h>

int main()
{
    int a = 2147483640, b = 2147483640, c = 0, d = 0;

    if( ( unsigned )a + ( unsigned )b > INT_MAX )
    {
        printf( " 最大的整数是：%d, 最小的整数是：%d \n",INT_MAX,INT_MIN );
        printf( " INT_MAX 的值为 2147483647，INT_MIN 的值为-2147483648  \n" );
    }

    c = a + b;
    d = -a + -b;
    printf( "***%d***%d***\n", c, d );
    getchar();
    return 0;
}
```

返回整数的 getchar 函数(C 缺陷与陷阱 5.1 节)

我们首先看下面的这个例子：

```
#include <stdio.h>
int main()
{
      char c;     /* 改成：int c 就正确了 */

    while( ( c = getchar() ) != EOF )
    {
        putchar( c );
    }

    return 0;
}
```

getchar 函数在一般情况下返回的是标准输入文件中的下一个字符，当没有输入时返回 EOF（一个在头文件 stdio.h 中被定义的值，不同于任何一个字符）。这个程序乍一看似乎把标准输入复制到标准输出，实则不然。

原因在于**程序中的变量 c 被声明为 char 类型，而不是 int 类型。这意味着 c 无法容下所有可能的字符，特别是，可能无法容下 EOF**。因此，最终结果存在以下几种可能：

1、某些合法的输入在被截断后，使得 c 的取值和 EOF 相同；这时程序将在文件复制的中途终止；

2、c 根本不可能取到 EOF 这个值；这时程序陷入死循环；

3、程序表面上似乎能够正常工作，但完全是因为巧合。尽管函数 getchar 的返回结果在赋值给 char 类型的变量 c 时会发生“截断”操作，尽管 while 语句中比较运算的操作数不是函数 getchar 的返回值，然而令人惊讶地是许多编译器对上述表达式的实现并不正确。这些编译器确实对函数 getchar 的返回值做了截断处理，并把低端直接部分赋给了变量 c。但是，它们在比较表达式中并不是比较 c 与 EOF，而是比较 getchar 函数的返回值与 EOF，编译器如果采取这种做法，上面的例子程序看上去就能够“正常”运行了。

更新顺序文件（C 缺陷与陷阱 5.2 节）

许多系统中的标准输入/输出库都允许程序打开一个文件，同时进行写入和读出的操作：

```
FILE *fp;
fp = fopen( file, "r+" );
```

上面的代码打开了 file 指定的文件，对于存取权限的设定表明程序希望对这个文件进行输入和输出操作。编程者也许认为，程序一旦执行上述操作完毕，就可以自由的交错的进行读出和写入操作。遗憾的是，事情总难遂人愿，为了保持与过去不能同时进行读写操作的程序的向下兼容性，**一个输入操作不能随后直接紧跟一个输出操作，反之亦然。如果要同时进行输入和输出操作，必须在其中插入 fseek 函数的调用。**

下面的程序片段似乎更新了一个顺序文件中选定的记录：

```
FILE *fp;
struct record rec;
...
while( fread( (char *)&rec, sizeof(rec), 1, fp ) == 1 )
{
    /*对 rec 执行某些操作*/
    if( /*rec 必须被重新写入*/ )
    {
        fseek( fp, -(long)sizeof(rec), 1 );
        fwrite( (char *)&rec, sizeof(rec), 1, fp );
    }
}
```

这段代码乍看上去毫无问题，&rec 在传入 fread 和 fwrite 函数时被小心翼翼的转换为字符指针类型，sizeof(rec) 被转换为长整型。但是这段代码依然可能运行失败，而且出错的方式非常难于察觉 **[自注：fread 的定义为：size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)， fwrite 的定义类似， fseek 的定义为： int fseek(FILE *stream, long offset, int origin)]**

问题在于：如果一个记录需要重新被写入文件，也就是说，fwrite 函数得到执行，对这个文件执行的下一个操作将是循环开始的 fread 函数。因为在 fwrite 函数调用与 fread 函数调用之间缺少一个 fseek 函数调用，所以无法进行上述操作，解决的办法是把这段代码改写为：

```
FILE *fp;
struct record rec;
...
while( fread( (char *)&rec, sizeof(rec), 1, fp ) == 1 )
{
    /*对 rec 执行某些操作*/
    if( /*rec 必须被重新写入*/ )
    {
        fseek( fp, -(long)sizeof(rec), 1 );
        fwrite( (char *)&rec, sizeof(rec), 1, fp );
        fseek( fp, 0L, 1 );
    }
}
```

第二个 fseek 函数虽然看上去什么都没有做，但它改变了文件的状态，使得文件现在可以正常的进行读取了。

随机数的相关概念（c 和指针. P328. ）

有些程序每次执行时不应该产生相同的结果，如游戏和模拟，此时随机数就非常有用。下面两个函数合在一起使

用能够产生伪随机数。之所以如此称呼是因为它们通过计算产生随机数，因此有可能重复出现，所以并不是真正的随机数。

int rand(void);
void srand(unsigned int seed);

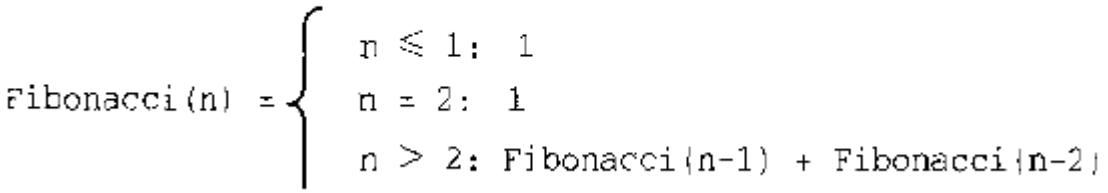
rand 返回一个范围在 0 和 RAND_MAX(至少为 32767)之间的伪随机数。当它被重复调用时，函数返回这个范围内的其他数。为了得到一个更小范围内的随机数，可以把这个函数的返回值根据所需范围的大小进行取模。为了避免程序每次运行时获得相同的随机数序列，我们可以调用 srand 函数。它用它的参数值对随机数发生器进行初始化。一个常用的技巧是使用每天的时间作为随机数产生器的种子。如下面的程序所示：

srand((unsigned int)time(0));

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define TRUE 1
#define FALSE 0

//使用随机数在牌桌上洗牌。第二个参数指定牌的张数。当这个函数第一次调用时，
//调用 srand 函数初始化随机数发生器
void shuffle( int *deck, int n_cards )
{
    int i;
     static int first_time = TRUE;

     //如果尚未进行初始化，用当天的当前时间作为随机数发生器
     if( first_time )
     {
          first_time = FALSE;
          srand( ( unsigned int )time( NULL ) );
     }

     //通过交换随机对的牌进行洗牌
     for( i = n_cards - 1; i > 0; i-- )
     {
          int where;
          int temp;

          where = rand() % i;

          temp = deck[where];
          deck[where] = deck[i];
          deck[i] = temp;
     }
}
.
.
.
.
.
```

用递归和迭代两种办法解 fibonacci

菲波那契数就是一个数列，数列中每个数的值就是它前面两个数的和。
这种关系常常用递归的形式进行描述：



Long

```
#include <stdio.h>

/*用递归的方法计算第 n 个非波那契数的值*/
long fibonacci1(int n)
{
    if( n <= 2 )
        return 1;
    else
        return fibonacci1(n-1)+fibonacci1(n-2);
}

/*用迭代的方法计算第 n 个非波那契数的值*/
long fibonacci2(int n)
{
     long result;
     long previous_result;
     long next_older_result;

     result = previous_result = 1;
     while( n > 2 )
     {
         --n;
         next_older_result = previous_result;
         previous_result = result;
         result = previous_result + next_older_result;
     }
     return result;
}


int main()
{
    printf("fibonacci1(10) = %ld\n", fibonacci1(10) );
    printf("fibonacci2(10) = %ld\n", fibonacci2(10) );

    getchar();
    return 0;
}
```