Pointers, Arrays, Memory

Address vs value

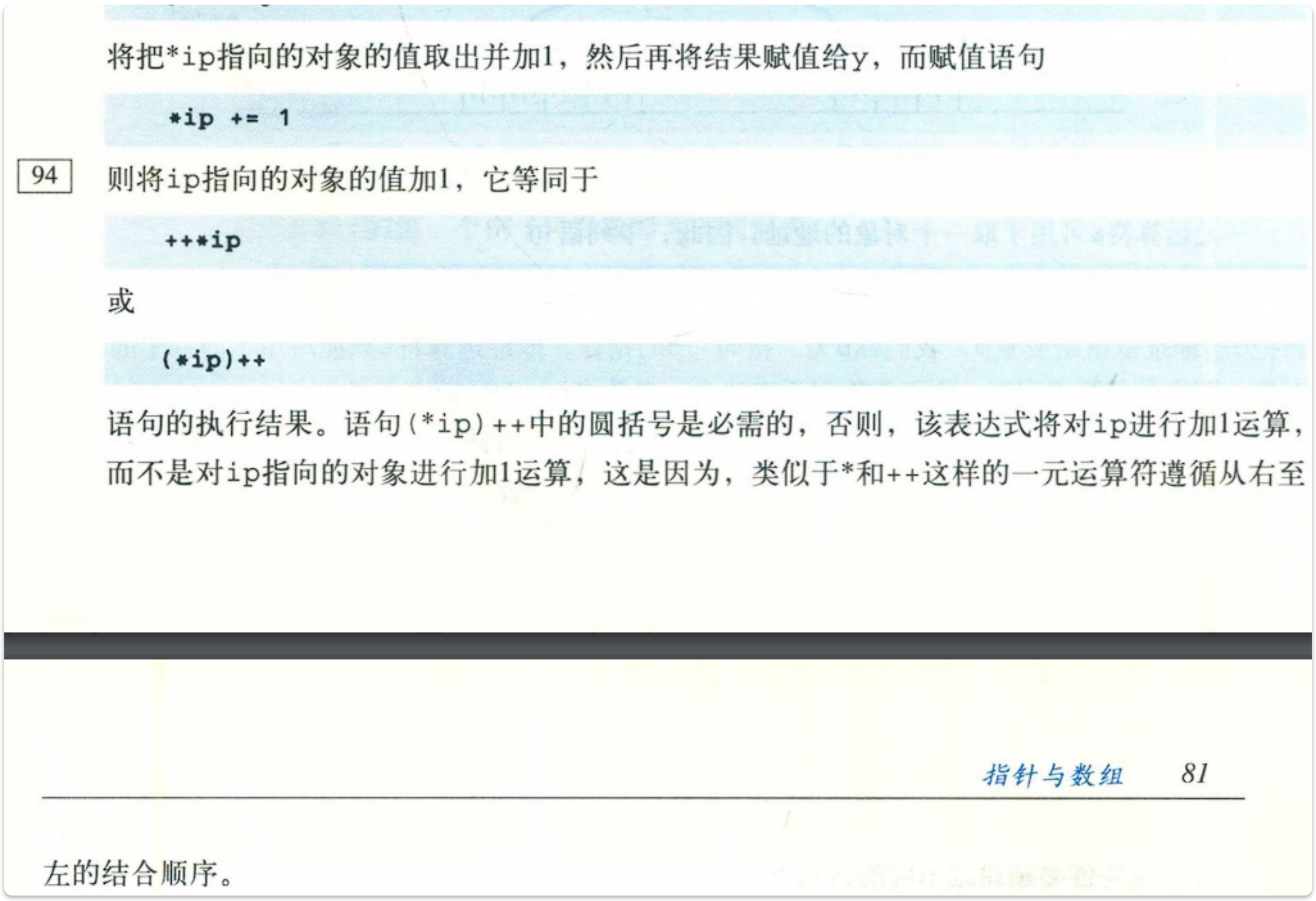
$void*$ is a type that can point to anything.
word alignment.

Pointer Arithmetic

```
char *c;
char **d;
(c+5)-> c + sizeof(char)*5 ->c+5
(d+7)-> d + sizeof(char*)*7 ->d+28
```

- The actual value used by the compiler is the size what the pointer are pointing to.

Conclusion on Pointers



将把*ip指向的对象的值取出并加1，然后再将结果赋值给y，而赋值语句

```
*ip += 1
```

94 则将ip指向的对象的值加1，它等同于

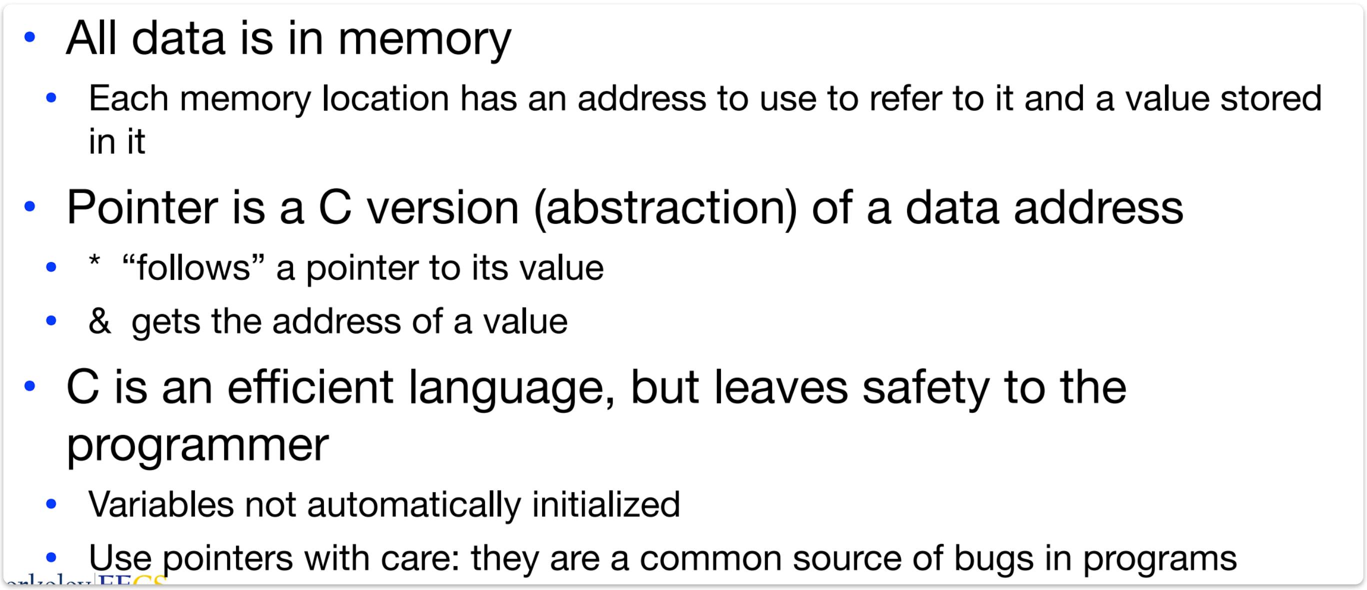
```
++*ip
```

或

```
(*ip)++
```

语句的执行结果。语句(*ip)++中的圆括号是必需的，否则，该表达式将对ip进行加1运算，而不是对ip指向的对象进行加1运算，这是因为，类似于*和++这样的一元运算符遵循从右至

左的结合顺序。



- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - * “follows” a pointer to its value
  - & gets the address of a value
- C is an efficient language, but leaves safety to the programmer
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs

Struct

It's an instruction to C on how to arrange a bunch of bytes in a bucket.
Provides enough space and aligns the data with padding.

```
struct foo{
int a;
char b;
struct foo *c;
}
```

So the real memory layout will be:
4 bytes for a,
1 byte for b,
3 bytes empty,
4 bytes for c.

Unions

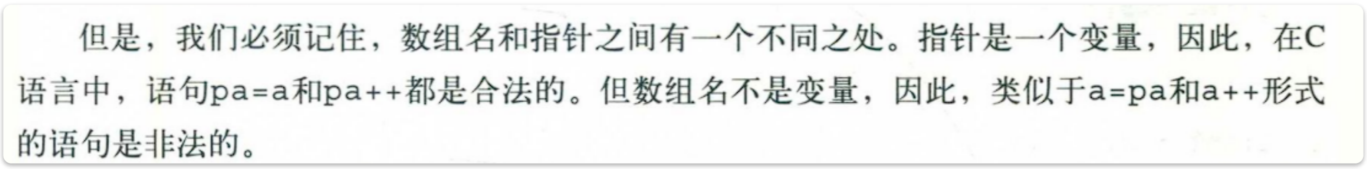
```
union foo{
int a;
char b;
union foo *c
}
```

Provides enough space for the **largest** element.

C Arrays

```
int ar[2];
```

The number of elements is static in the declaration, you can't do `int ar[x]` where x is a variable


但是，我们必须记住，数组名和指针之间有一个不同之处。指针是一个变量，因此，在C语言中，语句pa=a和pa++都是合法的。但数组名不是变量，因此，类似于a=pa和a++形式的语句是非法的。

- Can use pointer variable to access arrays.
  An array is passed into a function as a pointer.

C Strings

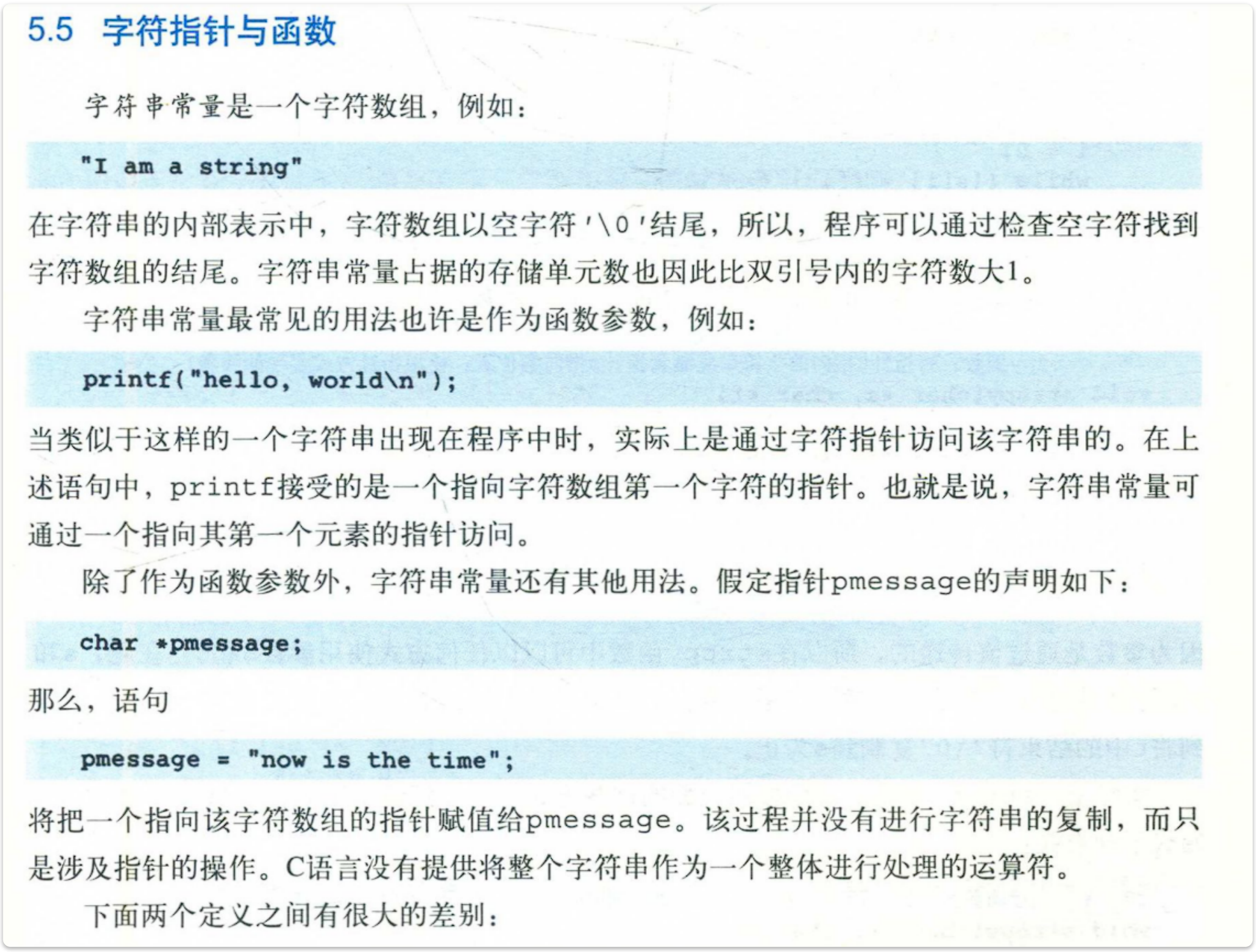
This can be modified.

```c
char string[] = "abc";
```

This can't. 字符串常量

```c
char *string = "abc";
```

Ends with a `\0`.



5.5 字符指针与函数

字符串常量是一个字符数组，例如：

```
"I am a string"
```

在字符串的内部表示中，字符数组以空字符'\0'结尾，所以，程序可以通过检查空字符找到字符数组的结尾。字符串常量占据的存储单元数也因此比双引号内的字符数大1。

字符串常量最常见的用法也许是作为函数参数，例如：

```
printf("hello, world\n");
```

当类似于这样的一个字符串出现在程序中时，实际上是通过字符指针访问该字符串的。在上述语句中，printf接受的是一个指向字符数组第一个字符的指针。也就是说，字符串常量可通过一个指向其第一个元素的指针访问。

除了作为函数参数外，字符串常量还有其他用法。假定指针pmessage的声明如下：

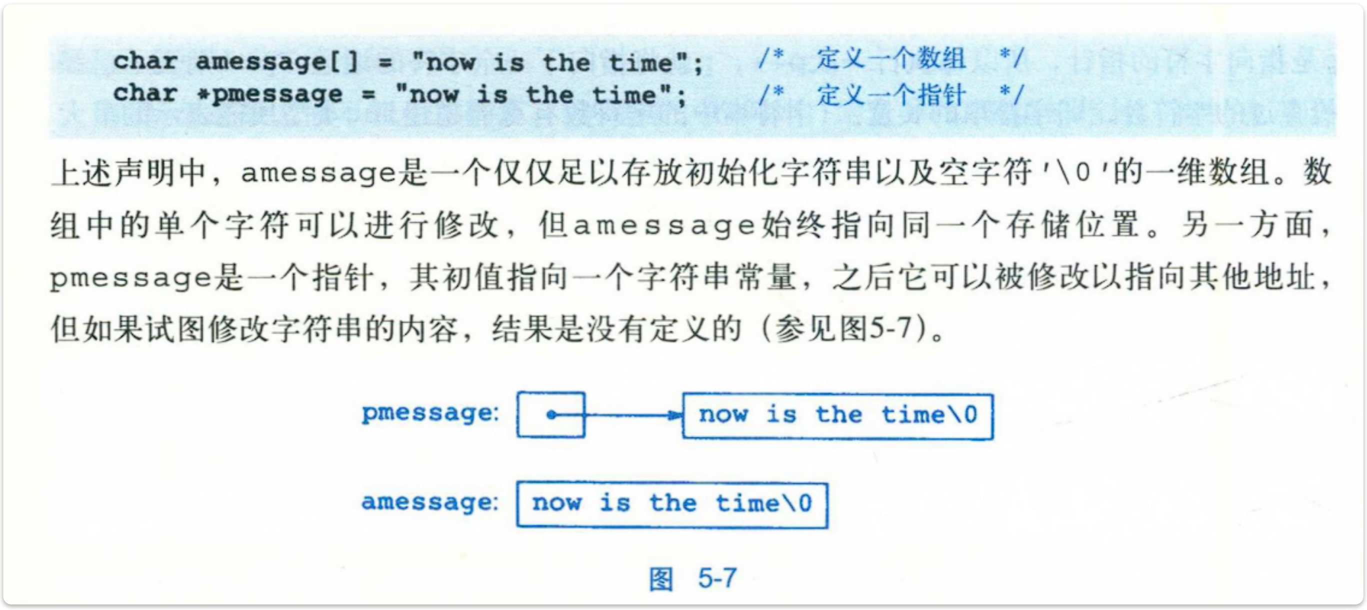
```
char *pmessage;
```

那么，语句

```
pmessage = "now is the time";
```

将把一个指向该字符数组的指针赋值给pmessage。该过程并没有进行字符串的复制，而只是涉及指针的操作。C语言没有提供将整个字符串作为一个整体进行处理的运算符。

下面两个定义之间有很大的差别：



```
char amessage[] = "now is the time";    /*  定义一个数组  */
char *pmessage = "now is the time";     /*  定义一个指针  */
```

上述声明中，amessage是一个仅仅足以存放初始化字符串以及空字符'\0'的一维数组。数组中的单个字符可以进行修改，但amessage始终指向同一个存储位置。另一方面，pmessage是一个指针，其初值指向一个字符串常量，之后它可以被修改以指向其他地址，但如果试图修改字符串的内容，结果是没有定义的（参见图5-7）。

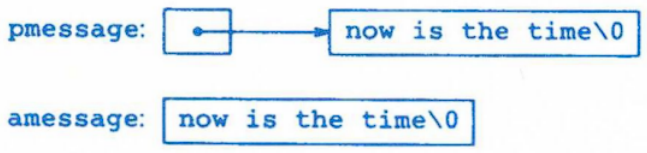
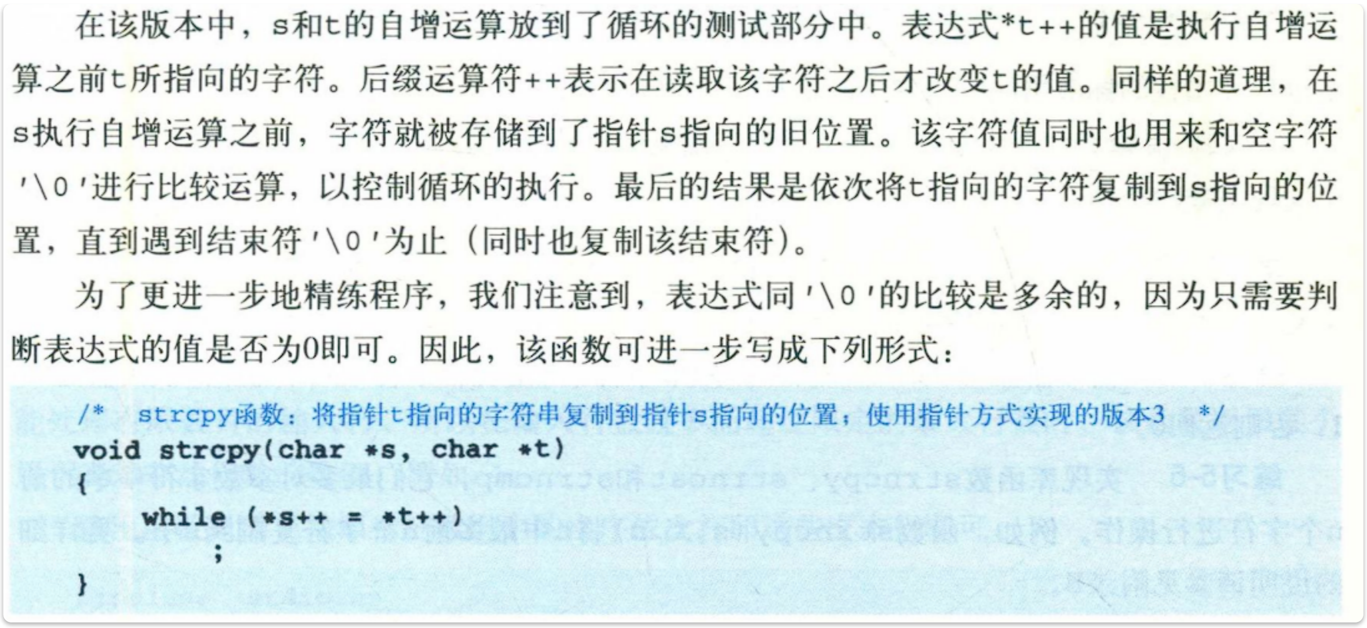


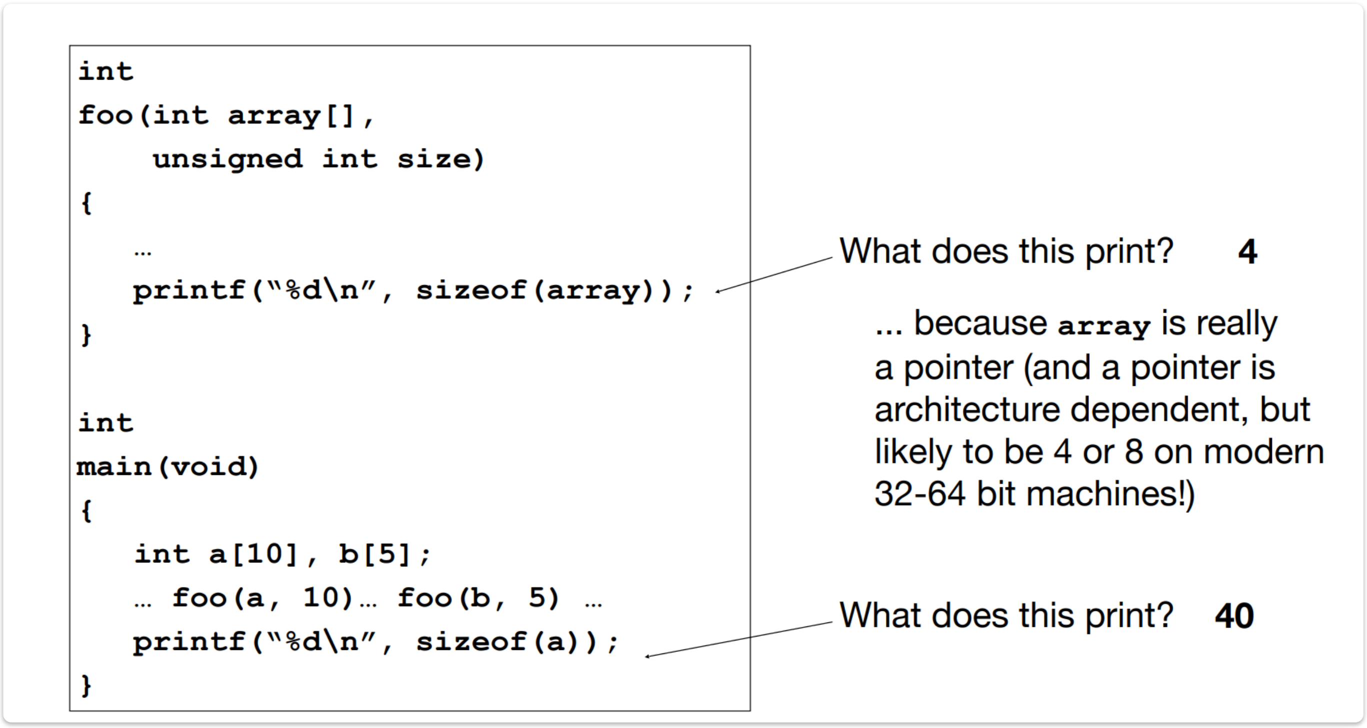
图 5-7



在该版本中，s和t的自增运算放到了循环的测试部分中。表达式*t++的值是执行自增运算之前t所指向的字符。后缀运算符++表示在读取该字符之后才改变t的值。同样的道理，在s执行自增运算之前，字符就被存储到了指针s指向的旧位置。该字符值同时也用来和空字符'\0'进行比较运算，以控制循环的执行。最后的结果是依次将t指向的字符复制到s指向的位置，直到遇到结束符'\0'为止（同时也复制该结束符）。

为了更进一步地精练程序，我们注意到，表达式同'\0'的比较是多余的，因为只需要判断表达式的值是否为0即可。因此，该函数可进一步写成下列形式：

```
/*  strcpy函数：将指针t指向的字符串复制到指针s指向的位置；使用指针方式实现的版本3  */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```



```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5)  …
   printf("%d\n", sizeof(a));
}
```

What does this print? **4**

... because `array` is really a pointer (and a pointer is architecture dependent, but likely to be 4 or 8 on modern 32-64 bit machines!)

What does this print? **40**

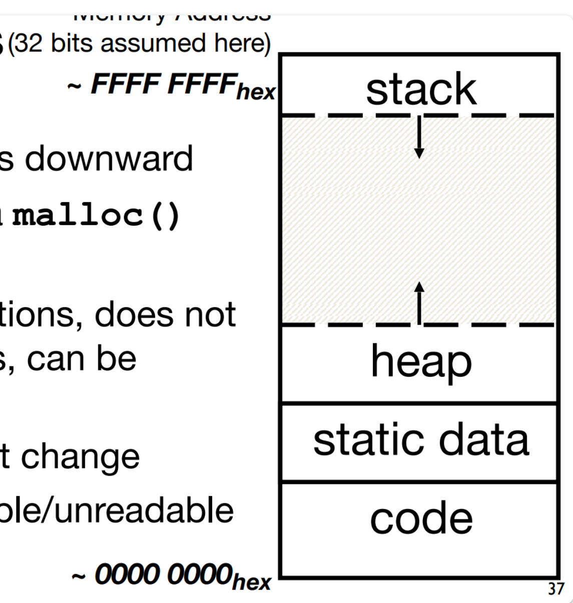
Endianness

The network byte order is big-endian.
Endian conversion functions in C:

```
ntohs()
htohs()
```

C Memory Management

- Program's address space contains 4 regions:
  - ***stack***: local variables inside functions, grows downward
  - ***heap***: space requested for dynamic data via **`malloc()`** resizes dynamically, grows upward
  - ***static data***: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - ***code***: loaded when program starts, does not change
  - 0x0000 0000 hunk is reserved and unwriteable/unreadable so you crash on null pointer access



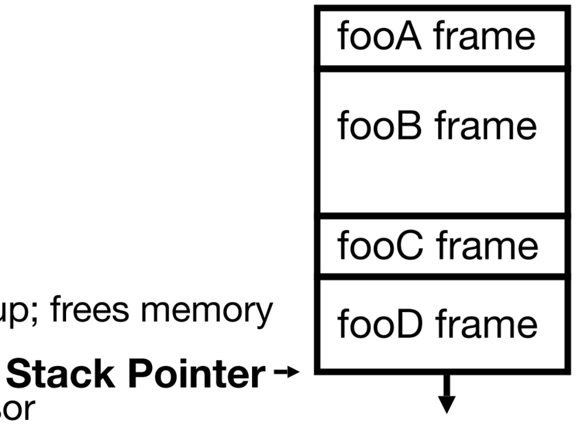
- If declared outside a function, allocated in "static" storage
- If declared inside function, allocated on the "stack" and freed when function returns
  - **`main()`** is treated like a function

```
int myGlobal;
main() {
   int myTemp;
}
```

- For both of these types of memory, the management is automatic:
  - You don't need to worry about deallocating when you are no longer using them
  - But a variable ***does not exist anymore*** once a function ends! Big difference from Java

- Every time a function is called, a new "stack frame" is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames uses contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack pointer moves up; frees memory for future stack frames
- We'll cover details later for RISC-V processor

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```



Managing the Heap

C supports functions for heap management:

- **`malloc()`** allocate a block of ***uninitialized*** memory
- **`calloc()`** allocate a block of ***zeroed*** memory
- **`free()`** free previously allocated block of memory
- **`realloc()`** change size of previously allocated block
  - careful – it might move!
    - And it ***will not update other pointers pointing to the same block of memory***