

# 《深入理解C指针》学习笔记

## 1 认识指针

### 1.1 指针和内存

	作用域	生命周期
全局内存	整个文件	应用程序的生命周期
静态内存	声明它的函数内部	应用程序的生命周期
自动内存（局部内存）	声明它的函数内部	限制在函数执行时间内
动态内存	由引用该内存的指针决定	直到内存释放

#### 1.1.2 声明指针

星号两边的空白符无关紧要，下面的声明都是等价的：

```
1  int* pi;
2  int * pi;
3  int *pi;
4  int*pi;
```

空白符的使用是个人喜好

#### 1.1.3 如何阅读声明

倒过来读：const int \*pci



- 1. pci是一个变量：const int \*pci;
- 2. pci是一个指针变量：const int \*pci;
- 3. pci是一个指向整数的指针变量：const int \*pci;

4. pci是一个指向整数常量的指针变量：`const int *pci;`

## 1.1.4 地址操作符

地址操作符`&`会返回操作数的地址。

尽快初始化指针是一个好习惯

## 1.1.5 打印指针的值

格式说明符	含义
%d	将值显示为十进制数
%x	将值显示为十六进制数
%o	将值显示为八进制数
%p	将值显示为实现专用的格式，通常是十六进制数

`%p` 和 `%x` 的不同之处在于：`%p`一般会把数字显示为十六进制的大写。

## 虚拟内存和指针

每个程序都假定自己能够访问机器的整个物理内存空间，实际上却不是。程序使用的地址是虚拟地址。操作系统会在需要时把虚拟地址映射为物理地址。应用程序的虚拟地址不会变，就是我们在查看指针内容时看到的地址。操作系统会帮我们将虚拟地址映射为真实地址。

## 1.1.8 null的概念

- null概念是通过null指针敞亮来支持的一种抽象。这个常量可能是也可能不是常量0。C程序员不需要关心实际的内部表示。
- NULL宏是强制类型转换为void指针的整数常量0。在很多库中定义如下：`#define NULL ((void *)0)`



- 如果编译器使用一个非零的位串来表示null，使用NULL或0是在语言层面表示null指针的符号，实际的null内部表示由实现定义。
- ASCII字符NUL定义为全0的字节。
- null字符串是空字符串，不包含任何字符。
- null语句就是只有一个分号的语句。

有趣的是，我们可以给指针赋0，但是不能赋任何别的整数值。

指针可以作为逻辑表达式的唯一操作数。

任何时候都不应该对null指针进行解引，因为它并不包含合法地址。执行这样的代码会导致程序终结。

## 1.用不用NULL

对于指针，使用NULL或0都可以，但NULL不能用于指针之外的上下文中。尤其是替代ASCII字符NUL是有问题的。

0的含义随着上下文变化而变化，有时可能是整数0，有时可能是null指针。

```
1  int num;
2  int *pi = 0; //这里的0表示null的指针NULL
3  pi = &num;
4  *pi = 0; //这里的0表示整数0
```

## 2.void指针

void指针是通用指针，用来存放任何数据类型的引用。

- void指针具有与char指针相同的形式和内存对齐方式；
- void指针和别的指针永远不会相等，不过，两个赋值为NULL的void指针是相等的。

void指针只能用做数据指针，而不能用做函数指针。

## 3.全局和静态指针



指针被声明为全局或静态，就会在程序启动时被初始化为NULL。

## 1.2 指针的长度和类型

### 1.2.1 内存模型

模型取决于操作系统和编译器，一种操作系统可能支持多种模型，这通常是用编译器选项来控制的。

### 1.2.2 指针相关的预定义类型

- `size_t`: 用于安全的表示长度
- `ptrdiff_t`: 用于处理指针算术运算
- `intptr_t` `uintptr_t`: 用于存储指针地址

#### 1.理解size\_t

`size_t`类型表示C中任何对象所能达到的最大长度。它是无符号整数，因为负数在这里没有意义。它的目的是提供一种可移植的方法来声明与系统中可寻址的内存区域一致的长度。`size_t`用做sizeof操作符的返回值类型，同时也是很多函数的参数类型，包括malloc和strlen。

在声明诸如字符数或者数组索引这样的长度变量时用size\_t是好的做法。

打印size\_t类型的值要小心，因为这是无符号的值。推荐的格式说明符是%zu。不过，某些情况下不能用这个说明符，作为替代，可以考虑%u或%lu。


#### 2对指针使用sizeof操作符

sizeof操作符可以用来判断指针长度。

当需要使用指针长度时，一定要用sizeof操作符。

函数指针的长度是可变的，对于同一个程序来说，不同的编译选项会导致其使用不同的指针长度。

#### 3.使用intptr\_t和uintptr\_t

 `intptr_t`和`uintptr_t`类型用来存放指针地址。他们提供了一种可移植且安全的方法声明指针，而且和系统中使用的指针长度相同，对于把指针转化成为整数形式来说很有用。

`uintptr_t`是`intptr_t`的无符号版本。对于大部分操作，用`intptr_t`比较好。`uintptr_t`不像`intptr_t`那样灵活。

当可移植性和安全性变得重要时，就应该使用这些类型。

避免把指针转换成整数。如果指针是6.位，整数只有4字节时就会丢失信息。

## 1.3 指针操作符

### 1.3.1 指针算术运算

#### 1.给指针加上整数

给指针加上一个整数实际上加的这个整数是这个整数和指针数据类型对应字节数的乘积。

#### 4.指针相减

一个指针减去另一个指针会得到两个地址的差值。这个差值通常没什么用，但可以判断数组中的元素顺序。指针之间的差值是他们之间相差的“单位”数，差的符号取决于操作数的顺序。

`ptrdiff_t`类型表示两个指针差值的可移植方式。

### 1.3.2 比较指针

指针可以用标准的比较操作符来比较。当把指针和数组元素相比时，比较结果可以用来判断数组元素的相对顺序。

## 1.4 指针的常见用法

### 1.4.2 常量与指针

#### 1.指向常量的指针

可以将指针定义为指向常量，这意味着不能通过指针修改他所引用的值。

我们不能解引指向常量的指针并改变指针所引用的值，但可以改变指针。指针的值不是常量。指针可以改为引用另一个整数常量，或者普通整数。这样做不会有问题。声明只是限制我们不能通过指针来修改引用的值。

```
1  const int limit=500;
2  const int *pci;
3  pci=&limit;
```

把pci声明为指向整数常量的指针意味着：

- pci可以被修改为指向不同的整数常量
- pci可以被修改为指向不同的非整数常量
- 可以解引pci以读取数据
- 不能解引pci从而修改他指向的数据

数据类型和const关键字的顺序不重要。

## 2.指向非常量的常量指针

一个指向非常量的常量指针，意味着指针不可变，但是它指向的数据可变。

```
1  int num;
2  int *const cpi = &num;
```

- cpi必须被初始化为指向非常量变量；
- cpi不能被修改；
- cpi指向的数据可以被修改。

无论cpi引用了什么，都可以解引cpi然后赋一个新值。所以如果试图把cpi初始化为指向常量limit，那么常量就可以修改了，这样是不对的，因为常量不可以被修改，所以会产生一个警告。

## 3.指向常量的常量指针

这种指针本身不能被修改，它指向的数据也不能通过它来修改。

```
1  const int * const cpci = &limit;
```

与指向常量的指针类似，不一定只能将常量的地址赋给cpci。

声明指针时必须进行初始化。如果不进行初始化就会产生语法错误。



## 4.指向“指向常量的常量指针”的指针

```
1  const int * const * pcpci;
```

总结

指针类型	指针是否可修改	指针指向的数据是否可修改
指向非常量的指针	是	是
指向常量的指针	是	否
指向非常量的常量指针	否	是
指向常量的常量指针	否	否

## 2 C的动态内存管理

静态变量或全局变量，内存处于程序的数据段，会被自动清零。数据段是一个区别于可执行代码和运行时系统管理的其他数据的内存区域。

C99引入了变长数组（VLA）。数组长度在运行时而不是编译时确定。不过，数组一旦创建出来就不能再改变长度了。

### 2.1 动态内存分配

malloc 函数的参数指定要分配的字节数。如果成功，它会返回从堆上分配的内存的指针。如果失败则会返回空指针。sizeof 操作符使应用程序更容易移植，还能确定在宿主系统中应该分配的正确字节数。

#### 内存泄漏

导致内存泄漏的情况可能如下：

- 丢失内存地址；
- 应该调用 free 函数却没有调用（有时候也称为隐式泄漏）。



在释放用 struct 关键字创建的结构体时也可能发生内存泄漏。如果结构体包含指向动态内存分配的内存指针，那么可能需要在释放结构体之前先释放这些指针。

## 2.2 动态内存分配函数

函数	描述
malloc	从堆上分配内存
realloc	在之前分配的内存块的基础上，将内存重新分配为更大或者更小的部分
calloc	从堆上分配内存并清零
free	将内存块返回堆

分配的内存会根据指针的数据类型对齐，比如说，4字节的整数会分配在能被4整除的地址边界上。堆管理器返回的地址是最低字节的地址。

### 2.2.1 使用malloc函数

malloc 函数从堆上分配一块内存，所分配的字节数由该函数唯一的参数指定，返回值是 void 指针，如果内存不足，就会返回 NULL 。此函数不会清空或者修改内存，所以我们认为新分配的内存包含垃圾数据。

函数原型：

```
1 void* malloc(size_t);
```

如果参数是负数就会引发问题。在有些系统中，参数是负数会返回 NULL 。

如果 malloc 的参数是0，其行为是实现相关的：可能返回 NULL 指针，也可能返回一个指向分配了0字节区域的指针。如果 malloc 函数的参数是 NULL ，那么一般会生成一个警告然后返回0字节。



因为当 malloc 无法分配内存时会返回 NULL ，在使用它返回的指针之前先检查 NULL 是不错的做法。

#### 1.要不要强制类型转换



- 这样可以说明 malloc 函数的用意；
- 代码可以和C++（或早期的C编译器）兼容，后两者需要显示的类型转换。

### 5.静态、全局指针和malloc

初始化静态或全局变量时不能调用函数。对于静态变量，可以通过后面用一个单独的语句给变量分配内存来避免这个问题。但是全局变量不能用单独地赋值语句，因为全局变量是在函数和可制行代码外部声明的，赋值语句这类代码必须出现在函数中。

在编译器看来，作为初始化操作符的=和作为赋值操作符的=不一样。

## 2.2.2 使用calloc函数

calloc 会在分配的同时清空内存。该函数的原型如下：

```
1 void *calloc(size_t numElements, size_t elementSize);
```

清空内存的意思是将其内容置为二进制0。

calloc 函数会根据 numElements 和 elementSize 两个参数的乘积来分配内存，如果乘积为0，那么 calloc 可能返回空指针。如果不能分配内存，则会返回 NULL 。

如果内存需要清零可以使用 calloc ，不过执行 calloc 可能比执行 malloc 慢。

## 2.2.3 使用realloc函数

realloc 函数会重新分配内存，函数原型如下：

```
1 void *realloc(void *ptr, size_t size);
```

第一个参数为原内存的指针，第二个参数为请求的大小，返回值为新申请内存的指针。具体情况总结如下：

第一个参数	第二个参数	行为
-------	-------	----

第一个参数	第二个参数	行为
空	-	同malloc
非空	0	原内存块被释放
非空	比原来内存块小	利用当前的块分配更小的块，但不保证多余的内存被清空
非空	比原来内存块大	够的话在当前位置接着追加，不够则在其他位置重新分配更大的块，并把内容复制过来

## 2.3 用free函数释放内存

当使用 free 函数释放内存，原指针仍指向该内存，且内存里的值仍可能没有改变，这种情况称为**迷途指针**。

如果传递给 free 函数的参数是空指针，通常他什么都不做。如果不是 malloc 分配的内存，则行为将是未定义的。

应该在同一层管理内存的分配和释放。比如说，如果是在函数内分配的内存，那么就应该在同一个函数内释放它。

### 2.3.2 重复释放

重复释放同一块内存会造成运行时异常。

这两种情况都是重复释放，第二种更为隐蔽一些：

```
1  int *pi = (int*) malloc(sizeof(int));
2  *pi = 5;
3  free(pi);
4  ...
5  free(pi);
```

```
1  p1 = (int*) malloc(sizeof(int));
2  int *p2 = p1;
```

```
3  free(p1);  
4  ...  
5  free(p2);
```

## 2.3.3 堆和系统内存

堆的大小可能在程序创建后就规定不变了，也可能可以增长。不过堆管理器不一定会在调用 `free` 函数时将内存返还给操作系统。释放的内存只是可供应用程序后续使用。所以，如果程序先分配内存然后释放，从操作系统的角度看，释放的内存通常不会反映在应用程序的内存使用上。

## 3 指针和函数

局部变量也称为自动变量，他总是分配在栈帧上。

### 3.1 程序的栈和堆

#### 3.1.1 程序栈

程序栈和堆共享同一块内存区域。程序栈通常占据这块区域的下部，而堆用的则是上部。

函数被调用时，占向上“长出”一个栈帧。当函数终止时，其栈帧从程序栈上弹出。栈帧所使用的内存不会被清理，但最终可能被推到程序栈上的另一个栈帧覆盖。

动态分配的内存来自堆，对向下“生长”。随着内存的分配和释放，堆中会布满碎片。尽管堆是向下生长的，但这只是个大体方向，实际上内存可能在堆上的任意位置分配。

#### 3.1.2 栈帧的组织

栈帧由以下几种元素组成：

- 返回地址：函数完成后要返回的程序内部地址
- 局部数据存储：为局部变量分配的内存
- 参数存储：为函数参数分配的内存
- 栈指针和基指针：运行时系统来管理栈的指针



栈指针通常指向栈顶部。基指针（帧指针）通常存在并指向栈内部的地址。这两个指针都不是C指针。如果运行时系统用C实现，这些指针倒真是C指针。

C把块语句当作“微型”函数，会在适合的时机将其推入栈和从栈中弹出。

将栈帧推到程序栈上时，系统可能会耗尽内存，这种情况称为栈溢出。**要牢记每个线程通常都会有自己的程序栈。**

## 3.2 通过指针传递和返回数据

**传递参数（包括指针）时，传递的是他们的值。**也就是说，传递给函数的是参数的一个副本，当传递大型结构体的时候，使用指针会比直接复制参数值使得程序运行更快。

### 3.2.4 返回指针

函数返回指针时可能存在几个潜在的问题：

- 返回为初始化的指针
- 返回指向无效地址的指针
- 返回局部变量的指针
- 返回指针但是没有释放内存

### 3.2.5 局部数据指针

返回局部变量的指针，一旦函数执行完栈帧就被弹出了，尽管数据可能还在，但如果之后继续执行其他函数，该内存就会被覆盖。

如果使用静态变量，把作用域限制在函数内部，但是分配在栈帧外面，避免其他函数复写变量值。但每次调用该函数都会重复利用这个变量，这样相当于每次都把上一次调用的结果覆盖掉了。此外，静态数组必须声明为固定长度，这样会限制函数处理变长数组的能力。

### 3.2.7 传递指针的指针

#### ☐ 实现自己的free函数

```
1 void saferFree(void **pp){
```

```

2     if(pp != NULL && *pp != NULL){
3         free(*pp);
4         *pp = NULL;
5     }
6 }

```

如果调用这个函数时没有显示的把指针类型转换为 void 会产生警告，执行显示转换就不会有警告。下面这个宏生去了函数使用者做类型转换和传递指针的地址：

```

1  #define safeFree(p) safeerFree((void**)&p)

```

## 3.3 函数指针

### 3.3.1 声明函数指针

使用函数指针时一定要小心，因为C不会检查参数传递是否正确。

```

1  int (*f1)(double); //传入 double, 返回 int
2  void (*f2)(char*); //传入 char 指针, 没有返回值
3  double* (*f3)(int,int); //传递两个整数, 返回 double 指针

```

不要把返回指针的函数和函数指针搞混。

```

1  int *f4(); //f4是一个函数, 返回一个整数指针
2  int (*f5)(); //f5是一个返回整数的函数指针
3  int* (*f6)(); //f6是一个返回整数指针的函数指针

```

### 3.3.2 使用函数指针

```

1  int (*fptr1)(int);
2
3  int square(int num){
4      return num*num;
5  }
6
7  ...
8
9  int n = 5;

```

```
10  fptr1 = square; // fptr1 = &square;
11  printf("%d sward is %d\n",n,fptr1(n));
```

在这种上下文环境中，编译器会忽略取地址符操作，所以你可以写出来，但没必要这么做。

为函数指针声明一个类型定义会比较方便，类型定义的名字是声明的最后一个元素。

```
1  typedef int (*funcptr)(int);
2
3  ...
4
5  funcptr fptr2;
6  fptr2 = square;
7  printf("%d sward is %d\n",n,fptr1(n));
```

### 3.3.3 传递函数指针

```
1  int add(int num1, int num2){
2      return num1 + num2;
3  }
4
5  int subtract(int num1, int num2){
6      return num1 - num2;
7  }
8
9  typedef int (*fptrOperation)(int, int);
10
11 int compute(fptrOperation operation, int num1, int num2){
12     return operation(num1, num2);
13 }
14
15 ...
16
17 printf("%d\n", compute(add, 5, 6));
18 printf("%d\n", compute(sub, 5, 6));
```

## 返回函数指针

```
1  fptrOperation select(char opcode){
2      switch(opcode){
3          case '+': return add;
4          case '-': return subtract;
```

```

5      }
6  }
7
8  int evaluate(char opcode int num1, int num2){
9      fptrOperation operation=select(opcode);
10     return operation(num1,num2);
11 }
12
13 ...
14
15 printf("%d\n", evaluate('+', 5, 6));
16 printf("%d\n", evaluate('-', 5, 6));

```

### 3.3.5 使用函数指针数组

```

1  typedef int (*operation)(int, int);
2  operation operations[128]={NULL};

```

也可以不用 typedef 来声明这个数组：

```

1  int (*operations[128])(int, int)={NULL};

```

### 3.3.7 转换函数指针

我们可以将指向某个函数的指针转换为其他类型的指针，不过要谨慎使用，因为运行时系统不会验证函数指针所用的参数是否正确。也可以再转回来，得到的结果和原指针相同，但函数指针的长度不一定相等。

无法保证函数指针和数据指针转换后正常工作。

void\* 指针不一定能用在函数指针上。

## 4 指针和数组

一种常见的错误观点是“数组和指针是完全可以互换的”。尽管数组名字有时候可以当作指针来用，但数组的名字不是指针。数组表示法也可以和指针一起使用，但两者明显不同，也不一定能互换。尽管数组使用自身的名字可以返回数组地址，但名字本身不能作为赋值操作的目标。

## 4.1 数组概述

不要混淆二维数组和指针的数组，它们很类似，但是行为有点差别。

### 4.1.1 一维数组

对数组做 `sizeof` 操作会得到为该数组分配的字节数，要知道元素的数量，只需将数组长度除以元素长度。

## 4.2 指针表示法和数组

我们可以只用数组名字，也可以对数组的第一个元素用取地址操作符，这些写法是等价的，都会返回 `vector` 的地址。

```
1  int vector[5];
2
3  printf("%p\n",vector);
4  printf("%p\n",&vector[0]);
```

有时候也使用 `&vector` 这个表达式获取数组地址，不同于其他表示法，这么做返回的是整个数组的指针，其他两种方法得到是整数指针。

给指针加上一个整数会把它持有的地址增加这个整数和数据类型长度的乘积。

数组表示法可以理解为“偏移并解引”操作。

### 数组和指针的差别

```
1  int vector[5] = {1, 2, 3, 4, 5};
2  int *pv = vector;
```

`vector[i]` 生成的代码和 `*(vector+i)` 生成的不一样，`vector[i]` 表示法生成的机器码从位置 `vector` 开始，移动 `i` 个位置，取出内容。而 `*(vector+i)` 表示法，生成的机器码则是从 `vector` 开始，在地址上增加 `i`，然后取出这个地址中的内容。尽管结果是一样的，生成的机器码却不一样，对于大部分人来说，这种差别几乎无足轻重。



sizeof 操作符对数组和同一个数组的指针操作也是不同的。对 vector 调用 sizeof 操作符会返回20，就是这个数组分配的字节数。对 pv 调用 sizeof 操作符会返回4，就是指针的长度。

pv 是一个左值，左值表示赋值操作符左边的符号。左值必须能修改。像 vector 这样的数组名字不是左值，它不能被修改。

## 4.3 用malloc创建一维数组

用malloc创建的一维数组也可以使用数组表示法，但是用完之后要记得释放内存。

## 4.4 用realloc调整数组长度

如果 realloc 分配成功，我们不需要释放 buffer，因为 realloc 会把原来的缓冲区复制到新的缓冲区，再把旧的释放。如果试图释放 buffer，十有八九程序会终止，因为我们试图重复释放同一块内存。

## 4.7 指针和多维数组

```
1 int matrix[2][5] = {...}
2 int (*pmatrix)[5] = matrix;
```

上面的整条声明语句将 pmatrix 定义为一个指向二维数组的指针，该二维数组的元素类型是整数，每列有5个元素。如果我们把括号去掉就声明了5个元素的数组，数组元素的类型是整数指针。

matrix+1 得到的是数组的第二行，要得到数组第一行的第一个元素应使用 \*(matrix[0]+1)。

## 4.8 传递多维数组

要传递 matrix，可以这么写：

```
1 void display2DArray(int arr[][5], int rows){
2 ...
```

或者这么写：

```
1 void display2DArray(int (*arr)[5], int rows){
2  ...
```

这两种写法都指明了数组的列数，这很必要。第一种写法是数组指针的一个隐式声明，第二种写法则是指针的一种显示声明。

下面的声明是错误的：

```
void display2DArray(int *arr[5], int rows){
```

尽管不会产生语法错误，但是函数会认为传入的数组拥有5个整数指针。

也可能遇到下面这样的函数：

```
1 void display2DArrayUnknownSize(int *arr, int rows, int cols){
2  ...
```

要调用这个函数可以这么写：

```
1 dispaly2DArrayUnknownSize(&matrix[0][0], 2, 5);
```

在函数内我们无法像下面这样使用数组下标：

```
1 printf("%d ", arr[i][j]);
```

原因是没有将指针声明为二维数组。我们可以用一个下标，这样写只是解释为数组内部的偏移量，不能用两个下标是因为编译器不知道一维的长度：

```
1 printf("%d ", (arr+i)[j]);
```

这里传递的是 `&matrix[0][0]` 而不是 `matrix`，尽管 `matrix` 也能运行，但是会产生编译警告，原因是指针类型不兼容。`&matrix[0][0]` 表达式是一个整数指针，而 `matirx` 则是一个整数数组的指针。

 在传递二维以上的数组时，除了第一维以外，需要指定其他维度的长度。

## 4.9 动态分配二维数组

当我们用 `malloc` 这样和函数创建二维数组时，在内存分配上会有几种选择。由于我们可以将二维数组当作数组的数组，因而“内层”的数组没有理由一定要是连续的。如果对这种数组使用下标，数组的不连续对程序员是透明的。

内存的连续性还会影响复制内存等其他操作，内存不连续就可能需要多次复制。

## 4.9.2 分配连续内存

方法一：

```
1  int rows = 2;
2  int columns = 5;
3  int **matrix = (int **) malloc(rows * sizeof(int *));
4  matrix[0] = (int *) malloc(rows * columns * sizeof(int));
5  for(int i=1; i < rows; i++)
6      matrix[i] = matrix[0] + i * columns;
```

方法二：

```
1  int *matrix=(int *)malloc(rows * columns * sizeof(int));
```

后面的代码用到这个数组时不能使用下标，必须手动计算索引。

## 4.10 不规则数组和指针

复合字面量是一种C构造，前面看起来像类型转换，后面跟着花括号括起来的初始化列表。

```
1  (const int) {100}
2  (int[3]) {10, 20, 30}
```

通过复合字面量创建数组：

```
1  int (*(arr1[])) = {
2      (int[]) {0, 1, 2},
3      (int[]) {3, 4, 5},
4      (int[]) {6, 7, 8}};
5
6  int (*(arr2[])) = {
7      (int[]) {0, 1, 2, 3},
```

```
8     (int[]) {4, 5},  
9     (int[]) {6, 7, 8}};
```

这两个数组所有类存都是连续的，其中第二个是变长数组。都可以使用数组和指针表示法，但对于变长数组在遍历的时候可能要对每行单独写一个循环，不是很方便。

## 5 指针和字符串

### 5.1 字符串基础

字符串的长度是字符串中除了 NUL 字符之外的字符数。为字符串分配内存时，要记得为所有的字符加上 NUL 字符分配足够的空间。

记住，NULL 和 NUL 不同。NULL 用来表示特殊的指针，通常定义为 `((void*)0)`，而 NUL 是 `char`，定义为 `\0`，两者不能混用。

字符常量通常由一个字符组成，也可以包含很多字符，比如转义字符。在 C 中，它们的类型是 `int`，`char` 的长度是 1 字节，而字符字面量的长度是 4 字节。这个看似异常的现象乃语言设计者有意为之。

#### 5.1.1 字符串声明

声明字符串的方式有三种：字面量、字符数组和字符指针。字符串字面量是用双引号引起来的字符序列，常用来进行初始化，它们位于字符串字面量池中。不要把字符串字面量和单引号引起来的字符搞混——后者是字符字面量。

#### 5.1.2 字符串字面量池

定义字面量时通常会将其分配在字面量池中，多次用到同一个字面量时，字面量池中通常只有一份副本。这样会减少应用程序占用的内存。大部分编译器有关闭字面量池的选项，一旦关闭，字面量可能生成多个副本，每个副本拥有自己的地址。

GCC 用 `-fwritable-strings` 选项来关闭字符串池。在 Microsoft Visual Studio 中，`/GF` 选项会打开字符串池。

字符串字面量一般分配在只读内存中，所以是不可变的。字符串字面量不存在作用域的概念。

由于在有的编译器中（比如GCC），字符串字面量是可以修改的，因此最好把变量声明为 `const` 类型。

## 5.1.3 字符串初始化

### 1. 初始化char数组

```
1 char header1[] = "Media Player";
2
3 char header2[13];
4 strcpy(header, "Media Player");
```

下面的赋值是不合法的，我们不能把字符串字面量的地址赋给数组名字。

```
1 char header3[];
2 header3 = "Media Player";
```

### 2. 初始化char指针

```
1 char *header = (char*) malloc(strlen("Media Player")+1);
2 strcpy(header, "Media Player");
```

再决定 `malloc` 函数要用到的字符串长度时，要注意以下事项。

- 一定要记得算上终结符 `NUL` 。

不要用 `sizeof` 操作符，而是用 `strlen` 函数来确定已有字符串的长度。 `sizeof` 操作符会返回数组和指针的长度，而不是字符串的长度。

我们可以将字符串字面量的地址直接赋给字符指针，不过，这样不会产生字符串的副本。

```
1 char *header = "Media Player";
```

试图用字符字面量来初始化 `char` 指针不会起作用。因为字符字面量是 `int` 类型，这其实是尝试把整数赋给字符指针。这样将常会造成应用程序在解引指针时终止。

```
1 char* prefix = '+'; //不合法
```

正确的做法是像下面这样用 malloc 函数：

```
1 prefix = (char*)malloc(2);
2 *prefix = '+';
3 *(prefix+1) = 0;
```

### 3.从标准输入初始化字符串

这里会出问题是因为我们在使用 command 变量之前没有为其分配内存：

```
1 char *command;
2 printf("*Enter a Command:");
3 scanf("%s",command);
```

要解决这个问题需要首先为指针分配内存，或者使用定长数组代替指针。

### 4.字符串的位置小结

```
1 char* globalHeader = "Chapter";
2 char globalArrayHeader[] = "Chapter";
3
4 void displayHeader(){
5     static char* staticHeader = "Chapter";
6     char* localHeader = "Chapter";
7     static char staticArrayHeader[] = "Chapter";
8     char localArrayHeader[] = "Chapter";
9     char* heapHeader = (char*)malloc(strlen("Chapter")+1);
10    strcpy(heapHeader,"Chapter");
11 }
```

## 5.2 标准字符串操作

### 5.2.1 比较字符串



strcmp 函数原型：

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

函数返回以下三种值之一：

- 负数：如果按字典序（字母序）s1比s2小就返回负数。
- 0：如果两个字符串相等就返回0。
- 正数：如果按字典序s1比s2大就返回正数。

比较字符串的错误方法：

第一种，试图用复试操作符比较：

```
1 char command[16];
2 ...
3 if(command = "Quit"){
4 ...
```

首先，这不是作比较，其次，这样会导致类型不兼容的语法错误，我们不能把字符串字面量地址赋给数组名字。

另一种方法是相等操作符：

```
1 char command[16];
2 ...
3 if(command == "Quit"){
4 ...
```

这样会得到假，因为我们比较的是 command 的地址和字符串字面量的地址。相等操作符比较的是地址，而不是地址中的内容，用数组名字或者字符串字面量就会返回地址。

## 5.2.2 复制字符串

strcpy 函数原型：

```
1 char* strcpy(char *s1, const char *s2);
```

## 5.2.3 拼接字符串

strcat 函数原型：

```
1  cahr *strcat(char *s1, const char *s2);
```

此函数把第二个字符串拼接到第一个到结尾。函数不会分配内存，这意味着第一个字符串必须足够长，能容纳拼接后的结果，否则函数可能会越界写入，导致不可预期的行为。

函数的返回值的地址跟第一个参数的地址一样。这在某些情况下比较方便，比如这个函数作为 printf 函数的参数时。

### 正确的字符串拼接：

```
1  char* error = "ERROR: ";
2  char* errorMessage = "Not enough memory";
3
4  char* buffer = (char*)malloc(strlen(error)+strlen(errorMessage)+1);
5  strcpy(buffer,error);
6  strcat(buffer,errorMessage);
7
8  printf("%s\n",buffer);
9  printf("%s\n",error);
10 printf("%s\n",errorMessage);
```

输出：

```
1  ERROR: Not enough memory
2  ERROR:
3  Not enough memory
```

### 不正确的字符串拼接：

```
1  char* error = "ERROR: ";
2  char* errorMessage = "Not enough memory";
3
4  strcat(error, errorMessage);
5  printf("%s\n",error);
6  printf("%s\n",errorMessage);
```

输出：

```
1  ERROR: Not enough memory
2  ot enough memory
```



errorMessage 字符串会左移一个字符，原因是拼接后的结果覆写了 errorMessage。

如果我们像下面这样用 char 数组而不是用指针来存储字符串，就不一定能工作了：

```
1 char error[] = "ERROR: ";
2 char errorMessage[] = "Not enough memory";
```

如果用下面这个 strcat 调用会得到一个语法错误，这是因为我们试图把函数返回的指针赋给数组名字，这类操作不合法：

```
1 error = strcat(error, errorMessage);
```

如果像下面这样去掉赋值，就可能会有内存访问的漏洞，因为赋值操作会覆写栈帧的一部分。

```
1 strcat(error, errorMessage);
```

**一定要专门为拼接结果分配内存。**

```
1 char* path = "C: ";
2 char* currentPath = (char*)malloc(strlen(path)+2);
3 currentPath = strcat(currentPath, "\\");
```

因为在字符串字面量中用了转义序列，所以这里拼接的是一个反斜杠字符。

如果使用字符字面量，那么就会得到一个运行时错误，原因是第二个参数被错误的解释为 char 类型变量的地址。（此处其实是个整数，而参数是 char\*，所以整数被当成了地址。）

```
1 currentPath = strcat(path, '\\');
```

## 5.3 传递字符串

### 5.3.1 传递简单字符串

```
1 size_t stringLength(char* string){
```

```
2
3 ...
4
5 printf("%d\n",stringLength(simpleArray));
6 printf("%d\n",stringLength(&simpleArray));
7 printf("%d\n",stringLength(&simpleArray[0]));
```

第二个语句中，显示使用了取地址操作符，不过这样写有冗余，没有必要，而且会产生警告。第三个语句中，我们对数组第一个元素用了取地址操作符，这样可以工作，不过有点繁琐。

## 5.3.2 传递字符常量的指针

这样可以用指针传递字符串，同时也能防止传递的字符串被修改。

```
1 size_t stringLength(char* string){
2 ...
```

## 5.3.3 传递需要初始化的字符串

snprintf 函数第一个参数指向缓冲区。第二个参数指定缓冲区的长度，函数不会越过缓冲区写入。

## 5.4 返回字符串

### 5.4.2 返回动态分配内存的地址

这种情况写释放返回的内存是函数调用者的责任，如果用户没有释放返回的内存，则会造成内存泄漏。

返回局部字符串的地址，如果内存被别的栈帧覆写就会损坏，应该避免使用这种方法。

## 6 指针和结构体

### 6.1 介绍

如果使用结构体的简单声明，那么就使用点表示法来访问其字段。如果使用结构体指针，就需要使用箭头操作符。我们不一定非得用箭头操作符，可以先解引指针然后用点操作符。

## 为结构体分配内存

实际长度通常会大于各字段的长度和，因为结构体的各字段之间可能会有填充。某些数据类型需要对齐到特定边界就会产生填充。比如说，短整数通常对齐到能被2整除的地址上，而整数对齐到能被4整除的地址上。

这些额外的内存分配意味着几个问题：

- 要谨慎使用指针算术符运算
- 结构体数组的元素之间可能存在额外的内存

## 6.2 结构体释放问题

再为结构体分配内存时，运行时系统不会自动为结构体内部的指针分配内存。当结构体消失时，运行时系统也不会自动释放结构体内部的指针指向的内存。

## 6.3 避免malloc/free开销

通过维护一个结构体池，当用户需要的时候从池中取一个，如果池中没有了就动态申请一个。当用户不用了返回时，就放入池中，如果池满了就释放掉该结构体。

以 Person 结构体为例：

```
1  #define LIST_SIZE 10
2  Person *list[LIST_SIZE];
3
4  void initializeList(){
5      for(int i=0; i<LIST_SIZE; i++){
6          list[i] = NULL;
7      }
8  }
9
10 Person *getPerson(){
11     for(int i=0; i<LIST_SIZE; i++){
12         if(list[i] != NULL){
13             Person *ptr = list[i];
14             list[i] = NULL;
15             return ptr;
16         }
17     }
18     Person *person = (Person*)malloc(sizeof(Person));
```

```

19     return person;
20 }
21
22 Person *returnPerson(Person *person){
23     for(int i=0; i<LIST_SIZE; i++){
24         if(list[i] == NULL){
25             list[i] = person;
26             return person;
27         }
28     }
29     deallocatePerson(person);
30     free(person);
31     return NULL;
32 }

```

## 7 安全问题和指针误用

### 7.1 指针的声明和初始化

#### 7.1.1 不恰当的指针声明

按照如下写法，对计算机来说是把 ptr1 声明为整数指针， ptr2 声明为整数变量。但对程序员来说可能暗示 ptr1、ptr2 都是指针。

```

1  int* ptr1, ptr2;

```

同一行中把两个变量声明为指针的正确写法如下：

```

1  int *ptr1, *ptr2;

```

每个变量声明单独占一行更好。

用类型定义代替宏定义是另一个好习惯。类型定义允许编译器检查作用域规则，而宏定义不一定会。

```

1  #define PINT int*
2  PINT ptr1, ptr2;

```

在这里结果跟之前错误声明两个指针一样，更好的方法是用下面的类型定义：

```
1  typedef int* PINT;
2  PINT ptr1, ptr2;
```

两个变量均被声明为整数指针。

## 7.1.2 使用指针前为初始化

在初始化指针之前就使用指针会导致运行时错误，有时候将这种指针成为野指针。

## 7.2 指针的使用问题

下面几种情况可能导致缓冲区溢出：

- 访问数组元素是没有检查索引值
- 对数组指针做算术运算时不小心
- 用 gets 这样的函数从标准输入读取字符串
- 误用 strcpy 和 strcat 这样的函数

### 7.2.1 测试NULL

用 malloc 这类函数时一定要检查返回值，否则可能会导致程序非正常终止。

### 7.2.5 错误计算数组长度

将数组传递给函数时，一定要同时传递数组长度。

strcpy 函数允许缓冲区溢出，要谨慎使用 strcpy 这类不传递缓冲区长度的函数。传递缓冲区长度能提供额外的安全屏障。

### 7.2.6 错误使用sizeof操作符

```
1  int buffer[20];
2  int *pbuffer = buffer;
3  for(int i=0; i<sizeof(buffer); i++){
4      *(pbuffer++) = 0;
5  }
```

因为缓冲区长度一字节计算是80（20乘以4字节每元素）。可以在 for 表达式带测试条件中用 `sizeof(buffer)/sizeof(int)` 来避免这个问题。

## 7.2.7 一定要匹配指针类型

总是用适合的指针类型来装数据是个好主意。

## 7.2.9 字符串的安全问题

如果使用 `strcpy` 和 `strcat` 这类字符串函数，稍不留神就会引发缓冲区溢出。 `strncpy` 和 `strncat` 函数可以对这种操作提供一些支持，它们的 `size_t` 参数指定要复制的字符的最大数量。不过，如果字符数量计算不正确，替代函数也容易出错。

C11中（Annex K）加入 `strcat_s` 和 `strcpy_s` 函数，如果发生缓冲区溢出，它们会返回错误，目前只有Microsoft Visual C++支持。 `strcpy_s` 它接受三个参数：目标缓冲区、目标缓冲区的长度以及原缓冲区。如果返回值是0。就表示没有错误发生。

还有 `scanf_s` 和 `wsanf_s` 可以用来防止缓冲区溢出。

`gets` 函数从标准输入读取一个字符串，并把字符保存在目标缓冲区中，它可能会越过缓冲区的声明长度写入。如果字符串太长的话，就会发生缓冲区溢出。

`printf`、`fprintf`、`snprintf` 和 `syslog` 这些函数都接受格式化字符串作为参数，避免**格式化字符串攻击**的一种简单方法是永远不要把用户提供的格式化字符串传递给这些函数。

## 7.2.10 指针算术运算和结构体

我们应该只对数组使用指针算术运算，因为数组肯定分配在连续的内存块上。不过，不应该将他们用在结构体内，因为结构体的字段可能分配在不连续的内存区域。

## 7.2.11 函数指针的问题

如果函数和函数指针的签名不同，不要把函数赋给函数指针，这样会导致未定义的行为。



## 7.3 内存释放问题

## 7.3.2 清除敏感数据

一旦不再需要内存中的敏感数据，马上进行覆写是个好主意。

```
1 char name[32];
2 ...
3 //删除数据
4 memset(name,0,sizeof(name));
```

如果是指针：

```
1 char *name = (char*)malloc(...)
2 ...
3 memset(name,0,sizeof(name));
4 free(name);
```

## 8 其他重要内容

一个操作可能会调用某函数来执行任务，如果实际被调用的函数发生了改变，我们称之为回调函数。

### 8.1 转换指针

有时候容易将句柄和指针搞混。句柄是系统资源的引用，对资源的访问通过句柄实现。不过，句柄一般不提供对资源的直接访问，指针则包含了资源的地址。

#### 8.1.2 访问端口

机器用十六进制地址表示端口，将数据作为无符号整数处理。volatile 关键字修饰符表示可以在程序意外改变变量。用 volatile 关键字可以阻止运行时系统使用寄存器暂存端口值，每次访问端口都需要系统读写端口，而不是从寄存器中读取一个可能已经过期的值。

### 8.2 别名、强别名和restrict关键字



如果两个指针引用同一内存地址，我们称一个指针是另一个指针的别名。如果两个指针引用同一位置，那么任何一个都可能修改这个位置。当编译器生成读写这个位置的代码时，它就不能

通过把值放入寄存器来优化性能。对每次引用，它只能执行及机器别的加载和保存操作。频繁的加载/保存会很低效，在某些情况下，编译器还必须关心操作执行的顺序。

强别名是另一种别名，它不允许一种类型的指针称为另一种类型的指针的别名。

为避免别名问题，可以采用这几种技术：

- 使用联合体
- 关闭强别名
- 使用 char 指针

GCC编译器有如下的编译器选项：

- `-no-strict-aliasing` 可以关闭强别名
- `-fstrict-aliasing` 可以关闭强别名
- `-Wstrict-aliasing` 可以打开跟强别名相关的警告信息

编译器总是假定 char 指针是任意对象的潜在别名，所以，大部分情况下可以安全地使用。

## 8.2.3 使用restrict关键字

用 restrict 关键字可以在声明指针时告诉编译器这个指针没有别名，这样就允许编译器产生更高效的代码。

开发新的代码应该尽量对指针声明使用 restrict 关键字，这样会产生更高效的代码，而修改已有代码可能就不划算了。

一些标准C函数用了 restrict 关键字，包括：

- `void *memcpy(void * restrict s1, const void * restrict s2, size_t n);`
- `char *strcpy(char * restrict s1, const char * restrict s2);`
- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`
- `int printf(const char * restrict format, ... );`
- `int sprintf(char * restrict s, const char * restrict format, ... );`
- `int snprintf(char * restrict s, size_t n, const char * restrict format, ... );`



- `int scanf(const char * restrict format, ... );`

`restrict` 关键字隐含了两层含义：

1. 对编译器来说，这意味着它可以执行某些代码优化
2. 对程序员来说，这意味着这些指针不能有别名，否则操作的结果将是未定义的

