基本 C

尼克 • 帕兰特著

这份史丹佛计算机科学教育文档试图总结 C 语言的所有基本特性。覆盖范围相当快，所以它最适合作为复习或对于有其他语言编程背景的人来说。主题包括变量、int 类型、浮点类型、提升、截断、运算符、控制结构(if、while、for)、函数、值参数、引用参数、结构体、指针、数组、预处理器和标准 C 库函数。

最新版本始终保存在其史丹佛计算机科学教育图书馆 URL http://cslibrary.stanford.edu/101/。请将您的意见发送到 nick.parlante@cs.stanford.edu。

我希望你能本着善意的精神分享并享受这份文件——尼克 • 帕兰特, 4/2003，加州史丹佛大学。

目录

C 语言

C 语言是专业程序员的语言。它的设计初衷是尽可能少地妨碍别人。Kernighan 和 Ritchie 在他们的书《*C 编程语言》中写下了最初的语言*定义，这是他们在 AT&T 公司研究的一部分。Unix 和 c++诞生于同一个实验室。几年来，我一直使用 AT&T 作为我的长途运营商，以感谢所有的 CS 研究，但听到“感谢您使用 AT&T”的一百万次已经耗尽了这种善意。

有些语言是宽容的。程序员只需要对事物如何工作有基本的认识。代码中的错误被编译时或运行时系统标记，程序员可以蒙混过关，并最终修复问题，使其正确工作。C 语言不是这样的。

C 编程模型是程序员确切地知道他们想要做什么，以及如何使用语言构造来实现这个目标。这种语言让专业的程序员通过避开他们的方式，在最短的时间内表达他们想要的东西。

C 语言是“简单的”，因为语言中的组件数量很少——如果两个语言功能或多或少完成相同的事情，C 语言将只包括一个。C 语言的语法是简洁的，并且语言没有限制什么是“允许的”——程序员几乎可以做任何他们想做的事情。

C 语言的类型系统和错误检查只存在于编译时。编译后的代码运行在一个精简的运行时模型中，没有针对错误类型转换、错误数组索引或错误指针的安全检查。没有垃圾收集器来管理内存。取而代之的是程序员手动管理堆内存。所有这些都使得 C 语言快速但脆弱。

分析——C 适合的地方

由于上述特性，C 对于初学者来说很难。一个功能在一种上下文中可以正常工作，但在另一种上下文中就会崩溃。程序员需要了解这些功能是如何工作的，并正确使用它们。另一方面，功能的数量是相当少的。

像大多数程序员一样，我也有过真正讨厌 C 语言的时候。它可能令人恼火地顺从——你输入了一些错误的东西，它有一种编译良好的方式，只是在运行时做一些你意想不到的事情。然而，随着我成为一个更有经验的 C 程序员，我逐渐喜欢上了 C 的直来直去的风格。我学会了不要落入它的小陷阱，我欣赏它的简洁。

也许最好的建议就是小心行事。不要输入你不懂的东西。

调试花费太多时间。对你的 C 代码是如何使用内存的有一个脑海中的画面(或真实的图画)。这在任何语言中都是好的建议，但在 C 语言中，这是至关重要的。

Perl 和 Java 比 C 更“可移植”(您可以在不同的计算机上运行它们而无需重新编译)。Java 和 c++比 C 更结构化。结构对于大型项目很有用。C 语言最适合性能很重要的小型项目，程序员有时间和技能让它在 C 语言中工作。无论如何，C 语言是一种非常流行和有影响力的语言。这主要是因为 C 语言简洁(即使是最小的)的风格，它没有烦人或令人遗憾的结构，以及编写 C 编译器的相对轻松。

其他资源

- 《*C 编程语言*》(第 2 版)，Kernighan 和 Ritchie 著。一本薄薄的书，多年来被所有 C 程序员奉为圣经。由该语言的最初设计者编写。解释相当简短，所以这本书作为参考比初学者更好。
- http://cslibrary.stanford.edu/102/指针和内存——比本文更详细地介绍了局部内存、指针、引用参数和堆内存，内存确实是 C 和 c++中最难的部分。
- http://cslibrary.stanford.edu//103/链表基础-一旦你了解了指针和 C 的基础知识，这些问题是获得更多练习的好方法。

第一节
基本类型和操作符

C 语言提供了一组标准的、最小的基本数据类型。有时这些被称为“基本”类型。更复杂的数据结构可以由这些基本类型构建而成。

整数类型

C 语言中的“整数”类型组成了一个整数类型族。它们的行为都像整数一样，可以混合在一起并以类似的方式使用。差异是由于用于实现每种类型的比特数(“宽度”)不同——更宽的类型可以存储更大范围的值。

char ASCII 字符——至少 8 位。明显的“汽车”。作为一个实际问题，char 基本上总是一个字节，它是 8 位，足以存储一个 ASCII 字符。8 位提供了-128 的符号范围。127 或无符号范围是 0..255。字符还需要是机器的“最小可寻址单位”——内存中的每个字节都有自己的地址。

短的小整数——至少 16 位，提供了-32768..32767 的有符号范围。典型的大小是 16 位。不太常用。

int 默认整数——至少 16 位，典型为 32 位定义为计算机“最舒适”的大小。如果你真的不关心整数变量的范围，声明它 int，因为这可能是一个合适的大小(16 或 32 位)，这对机器很好。

长整型——至少 32 位。典型的大小是 32 位，这给出了大约 20 亿的符号范围。+ 20 亿。有些编译器支持 64 位整型的“long long”。

整数类型之前可以有一个 unsigned 限定符，这个限定符不允许表示负数，但可以将所能表示的最大正数加倍。例如，short 的 16 位实现可以在 range 中存储数字

-32768 . .32767，而 unsigned short 可以存储 0..65535。你可以认为指针是 unsigned long 在具有 4 字节指针的机器上的一种形式。在我看来，除非真的需要，否则最好避免使用 unsigned。它往往会引起更多的误解和问题，超出了它的价值。

附加:便携性问题

C 语言没有定义整数类型的确切大小，而是定义了下界。这使得在广泛的硬件上实现 C 编译器变得更加容易。不幸的是，它偶尔会导致程序在 16 位整型机器上的运行与在 32 位整型机器上的运行不同的 bug。特别是，如果您正在设计一个将在几台不同机器上实现的函数，那么使用 typedef 来设置类型是一个好主意，例如 32 位 int 的 Int32 和 16 位 int 的 Int16。这样，您就可以创建函数 Foo(Int32)的原型，并确信每台机器的类型都已设置，因此该函数确实接受 32 位 int 型。这样代码在所有不同的机器上的行为都是一样的。

字符常量

字符常量是用单引号(')写成的，比如'A'或'z'。字符常量`A`实际上只是普通整数值 65 的同义词，65 是 ASCII 中的值

大写字母“A”。还有一些特殊大小写字符常量，例如'\t'表示 tab，用于表示不方便在键盘上键入的字符。

“ 一个 ”　　大写'A'字符换行字符制

' \ n '　　表符

' \ t '

' \ '　　“null”字符——整数值 0(不同于字符数字“0”)

\012 八进制中值为 12 的字符，即十进制 10

整型常量

源代码中的数字，如 234，默认类型为 int。它们后面可以跟一个'L'(大写或小写)，表示常量应该是一个很长的常量，如 42L。整数常数可以以 0x 开头，表示它是用十六进制表示的——0x10 是表示数字 16 的方式。类似地，一个常数可以写成八进制，在它前面加上“0”——012 是表示数字 10 的一种方式。

类型组合和提升

整数类型可以在算术表达式中混合在一起，因为它们基本上都是整数，只是宽度不同。例如，char 和 int 可以在算术表达式中组合，例如('b' + 5)。编译器如何处理这样的表达式中出现的不同宽度?在这种情况下，编译器在组合值之前，会“提升”较小的类型(char)与较大的类型(int)的大小相同。提升是在编译时根据表达式中值的**类型**来确定的。提升不会丢失信息——它们总是将一种类型转换为兼容的、更大的类型，以避免丢失信息。

陷阱——int 溢出

我曾经有一段代码，它试图用表达式(k * 1024)计算缓冲区中的字节数，其中 k 是一个 int，表示我想要的字节数。不幸的是，这是在一个 int 恰好是 16 位的机器上。因为 k 和 1024 都是 int，所以没有提升。对于 k >= 32 的值，乘积太大，容纳不了 16 位的 int，导致溢出。编译器可以在溢出情况下做任何它想做的事情——通常高阶位就消失了。修复代码的一种方法是将其重写为(k * 1024L)——long 常量强制提升 int。追踪这个 bug 并不有趣——这个表达式在源代码中看起来很合理。只有在调试器中跳过关键行才能显示溢出问题。“专业程序员的语言。”这个例子还演示了 C 只基于表达式中的**类型**进行提升的方式。编译器不会考虑 32 或 1024 的值来意识到操作将溢出(一般来说，无论如何，这些值在运行时之前都不存在)。编译器只看编译时的类型，在这个例子中是 int 和 int，就认为一切正常。

浮点类型

float 单精度浮点数　　典型尺寸:32 位

double 双精度浮点数典型尺寸:64 位

long double 可能是更大的浮点数(有点模糊)源代码中的常数，如 3.14 默认键入 double，除非后缀为'f' (float)或'l' (long double)。单精度约等于 6 位

Precision 和 double 表示精度约为 15 位。大多数 C 程序使用 double 进行计算。使用 float 的主要原因是，如果需要存储许多数字，可以节省内存。关于浮点数，最需要记住的是它们不是精确的。例如，下面这个 double 表达式的值是多少?

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0)     //这个等于 1.0 吗?
```

这个和可能等于 1.0，也可能不等于 1.0，而且它可能因不同类型的机器而异。出于这个原因，你永远不应该比较浮点数之间的相等(==)——而应该使用不等(<)比较。要意识到，一个在不同计算机上运行的正确的 C 程序在其浮点计算的最右侧数字上可能会产生略有不同的输出。

评论

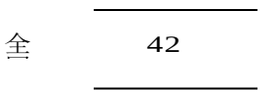
C 语言中的注释由斜线/星号组成:/* ..评论..*/可能跨越多行。c++引入了一种注释形式，以两个斜杠开始，并延伸到行尾://注释直到行尾//注释形式非常方便，许多 C 编译器现在也支持它，尽管从技术上讲它并不是 C 语言的一部分。

和精心选择的函数名一样，注释也是编写良好代码的重要组成部分。注释不应该只是重复代码所说的内容。注释应该描述代码**完成**了什么，这比翻译每个语句的功能要有趣得多。注释还应该说明一段代码中有什么棘手或不明显的地方。

变量

与大多数语言一样，变量声明会在运行时在内存中分配并命名一个区域，以保存特定类型的值。从语法上讲，C 语言把类型放在前面，然后是变量的名称。下面的代码声明了一个名为“num”的 int 变量，第二行代码将值 42 存储到 num 中。

```
int
num;Num
= 42;
```



变量对应于一个内存区域，该区域可以存储给定类型的值。绘图是思考程序中变量的一种很好的方式。把每个变量画成一个盒子，盒子里有当前值。这可能看起来像一个“初学者”的技巧，但当我被一些可怕的复杂编程问题所淹没时，我总是求助于画画来帮助思考问题。

在运行时分配变量(如 num)时，它们的内存不会被清除或以任何方式设置。变量从随机值开始，在依赖它们的值之前，由程序将它们设置为一些合理的值。

C 语言中的名称区分大小写，因此“x”和“x”指的是不同的变量。名称可以包含数字和下划线(_)，但不能以数字开头。可以在类型之后声明多个变量，变量之间用逗号分隔。C 是一种经典的“编译时”语言——变量的名称、它们的类型以及它们的实现，都是编译器在编译时刷出来的(而不是像解释器那样在运行时弄清楚这些细节)。

```
浮点数 x、 y、 z、
x;
```

赋值运算符=

赋值运算符是单个等号(=)。

```
i = 6;
i = i + 1;
```

赋值运算符将值从右边复制到左边的变量。赋值操作符也会像表达式一样返回新赋值的值。一些程序员会使用这个特性来编写下面的代码。

```
y = (x = 2 * x);      //double x， 并将 x 的新值放在 y 中
```

截断

与提升相反，截断将值从一种类型移动到更小的类型。在这种情况下，编译器只是删除额外的位。它可能会也可能不会生成一个关于信息丢失的编译时警告。将一个整数赋值给一个较小的整数(例如..将 Long 赋值给 int，或将 int 赋值给 char)会删除最高有效位。从浮点类型赋值给整数会删除数字的小数部分。

```
char
ch;int
我;


i = 321;
ch = i;      //截断 int 值以适应 char 类型

// ch 现在是 65
```

赋值操作会删除整数 321 的上半部分。数字 321 的下 8 位表示数字 65(321 - 256)。所以 ch 的值将是(char)65，正好是“A”。

将浮点类型赋值给整数类型会去掉数字的小数部分。下面的代码将 i 设置为值 3。当将浮点数赋值给整数或将浮点数传递给一个接受整数的函数时，会发生这种情况。

```
双 π;int
我;


pi = 3.14159;

I = pi;//截断 double 值以适应 int 值// I 现在是 3
```

陷阱——int 和 float 的算术运算

这是一个 int 和 float 运算可能导致问题的代码示例。假设下面的代码应该将家庭作业分数缩放到 0..20 的范围是 0..100。

```
{
   int 分数;
…//假设 score 的取值范围为 0..20 以某种方式
```

```
    Score = (Score / 20) *
100;…
```
//否——score/20 被截断为 0

不幸的是，对于这个代码，score 几乎总是设置为 0，因为表达式(score/20)中的整数除法将为 score 小于 20 的每个值的 0。解决方法是强制商数被计算为浮点数……

```
分数=((双)分数/ 20)* 100;得分=(得分/ 20.0)* 100;Score = (int)(Score / 20.0) * 100;
```
// OK——从 cast 进行浮点除法// OK——从 20.0 进行浮点除法

//否——(int)将浮点数截断回 0

No Boolean——使用 int

C 没有一个独特的布尔类型——用 int 代替。该语言将整数 0 视为 false，所有非零值视为 true。所以这个语句…

```
i = 0;

While(i -
    10){…
```

将执行，直到变量 I 获得值 10，此时表达式(I - 10)将变为 false(即 0)(稍后我们将看到 while()语句)

数学运算符

C 包含常见的二进制和一元算术运算符。优先级表见附录。就我个人而言，我只是随意地使用圆括号，以避免因对优先级的误解而产生任何 bug。运算符对操作数的类型很敏感。所以有两个整数参数的除法(/)会做整数除法。如果任何一个参数是浮点数，它会做浮点除法。所以(6/4)求值为 1，而(6/4.0)求值为 1.5——除法之前 6 被提升为 6.0。

+加减/除法*乘法%

余数(mod)

一元递增运算符:++——

一元运算符++和——对变量中的值进行递增或递减操作。这两个操作符都有“前”和“后”变体，它们做的事情略有不同(下面解释)

var++	增量	“post”变体
++*var*	增量	“以前”变体

var -
- - - - 衰减衰减 “后”变“前”
var 变

```
Int I = 42;
我+ +;// I 的增量// I 现在是
43
我,;// I 递减// I 现在是 42
```

前和后变化

前/后变化与在表达式内嵌套带有自增或自减操作符的变量有关——整个表达式应该表示变量在更改之前或之后的值吗?我**从来没有这样使用过**运算符(见下文)，但是一个例子看起来像……

```
Int I =
42;int j;


J = (i++ + 10);
//I 现在是 43
//j 现在是 52(不是 53)


J = (++i + 10)
// i 现在是 44
// J 现在是 54
```

C 语言编程的聪明与自我问题

依靠这些操作符前后变化的差异，是 C 程序员自我作秀的一个经典领域。语法有点棘手。它让代码更短一些。这些特性促使一些 C 程序员炫耀他们有多聪明。C 语言邀请这种事情，因为它有很多领域(这只是一个例子)，程序员可以使用简短而密集的代码获得复杂的效果。

如果我想让 j 依赖于 I 在递增之前的值，我写……

```
j = (i + 10);
i++;
```

或者我想让 j 使用增量之后的值，我写…

```
i++;
j = (i + 10);
```

这样不是更好吗?(编辑)构建做一些很酷的事情的程序，而不是灵活的语言语法的程序。语法——谁在乎呢?

关系运算符

这些操作符作用于整数或浮点数，并返回 0 或 1 布尔值。

= 平等

!=	不等于大于小于
>	大于等于小于等
<	于
>=	

要查看 x 是否等于 3，可以这样写:

```
If (x == 3)···
```

陷阱 = =

一个绝对经典的陷阱是当你想要比较(==)时写赋值(=)。这不会是这样的问题，除非不正确的赋值版本编译正常，因为编译器假定你是指使用赋值返回的值。这很少是你想要的

```
if (x = 3) ...
```

这并没有检验 x 是否为 3。这将 x 设置为值 3，然后将 3 返回给 if 进行测试。3 不是 0，所以每次都算作“true”。这可能是 C 语言初学者最常犯的错误。问题在于，编译器帮不了什么忙——它认为两种形式都没问题，所以唯一的防御就是编码时极度警惕。或者在编码前在手背上用大写字母写“=≠==”。这个错误绝对是经典，调试起来很吃力。小心!而且需要我说:“专业程序员的语言。”

逻辑运算符

值 0 为 false，其他值为 true。运算符从左到右求值，一旦可以推断出表达式的真或假，运算符就会停止。(这样的运算符被称为“短路”)在 ANSI C 中，这些进一步保证使用 1 来表示真，而不仅仅是一些随机的非零位模式。然而，有许多 C 程序使用 1 以外的值来表示 true(例如非零指针)，因此在编程时，不要假设一个真布尔值一定是 1。

!	布尔非(一元)
&	布尔与布尔
&|	或

按位

C 包含在位级别操作内存的运算符。这对于编写底层硬件或操作系统代码很有用，其中数字、字符、指针等的普通抽象…都不够用——这是一种越来越罕见的需求。位操作代码的“可移植性”往往较差。如果没有程序员的干预，代码可以在不同类型的计算机上正确地编译和运行，那么代码就是“可移植的”。位操作是

通常用于无符号类型。特别是，当用于无符号值时，移位操作保证将 0 位移到新空出的位置。

~	按位求反(一元)-在整个按位和过程中将 0 翻转为 1，将 1 翻
&	转为 0
|	按位
^	按位或
>>	右移右侧(RHS)(除以 2 的幂)左移 RHS(乘以 2 的幂)
<<	

不要把位操作符和逻辑操作符混淆了。位连接词是一个字符宽(&，|)，而布尔连接词是两个字符宽(&&，||)。位运算符的优先级高于布尔运算符。如果你用&，而你想说的是&&，编译器永远不会帮你解决类型错误。就类型检查器而言，它们是相同的——它们都接受并产生整数，因为没有不同的布尔类型。

其他赋值操作符

除了简单的=运算符之外，C 还包含许多表示基本=的变体的简写运算符。例如，“+=”将右手边的运算符加到左手边。X = X + 10;可以化简为 x += 10;。如果 x 是一个像下面这样的长表达式，这是最有用的，在某些情况下，它可能运行得更快一些。

```
人->relatives.mom。numChildren += 2;          //将 children 增加 2
```

下面是赋值简写操作符的列表…

+ =、 - =	
* =，=	按 RHS 增加或减少乘以或除以 RHS 乘以或除以 RHS
% =	
> > = <	位向右移 RHS(除以 2 的幂)位向左移 RHS(乘以 2 的幂)
&=， |=，	位向和、或、或按 RHS

第二节控制结构

花括号{}

C 使用大括号({})将多个语句分组在一起。语句按顺序执行。有些语言允许你在任何一行上声明变量(c++)。另一些语言则坚持变量只能在函数的开头声明(Pascal)。C 语言走中间路线——变量可以在函数体内声明，但必须跟在`{`后面。更现代的语言，比如 Java 和 c++，允许你在任何一行上声明变量，这很方便。

If 语句

在 c 中，if 和 if-else 都可用。<*表达式*>可以是任何有效的表达式。表达式周围的括号是必需的，即使它只是一个单一的变量。

```
If(<表达式>)<语句>                    //没有{}或 else 子句的简单形式

If(<表达式>){<语句>
    <语句>                            //用{}来分组语句的简单形式
}

If(<表达式>){
    <声明>}                           //完整的 then/else 格式
其他{
    <声明>}
```

条件表达式-or-三元运算符

条件表达式可以用作一些 if-else 语句的简写。条件运算符的通用语法是:

```
< expression1 > ?<表达式 2>:<表达式 3>
```

这是一个表达式，不是一个语句，所以它代表一个值。运算符的工作原理是求表达式 1 的值。如果为真(非零)，它计算并返回 expression2。否则，求值并返回 expression3。

三元运算符的经典例子是返回两个变量中较小的那个。每隔一段时间，下面的形式正是你所需要的。而不是……

```
If (x < y) {
    min = x;
}
其他{
    Min = y;}
```

你只要说……

```
Min = (x < y) ?X: y;
```

Switch 语 句

switch 语句是一种特殊形式的 if 语句，用于根据整数的值有效地分离不同的代码块。switch 表达式被求值，然后控制流跳转到匹配的 const-expression case。case 表达式通常是 int 或 char 常量。switch 语句可能是 C 语言中语法最笨拙、最容易出错的特性了。

```
Switch(<表达式>){
    Case <const-expression-1>:
        <语句>
        打破;

    Case <const-expression-2>:
        <语句>
        打破;

    Case <const 表达式 3>:Case      //这里我们将 case 3 和 case 4 结
    <const 表达式 4>:<语句>
        打破;

    默认://可选<statement>
}
```

每个常量都需要有自己的 case 关键字和一个以冒号(:)结尾。一旦执行跳到一个特定的 case，程序将继续运行从该点开始的所有 case——上面的例子中使用了这种所谓的“fall - through”操作，以便表达式-3 和表达式-4 运行相同的语句。显式的 break 语句是退出切换所必需的。省略 break 语句是一个常见的错误——它可以编译，但会导致无意的 fall-through 行为。

为什么 switch 语句的 fall-through 行为会这样工作呢?我能想到的最好的解释是，最初 C 是为汇编语言程序员的受众开发的。汇编语言程序员已经习惯了带有 fallthrough 行为的跳表，所以 C 就是这么做的(用这种方式实现也相对容易)。不幸的是，现在 C 语言的受众已经大不相同了，fall-through 行为被普遍认为是这门语言的一个可怕部分。

While 循 环

while 循环在每次循环之前计算测试表达式，因此如果条件最初为 false，它可以执行零次。它需要像 if 一样的括号。

```
While(<表达式>){<语
    句>
}
```

延伸的循环

类似于 while，但在循环的底部有测试条件。循环体总是至少执行一次。do-while 在这门语言中是一个不受欢迎的领域，如果可能的话，大多数人都会尝试直接使用 while。

```
{做
    <>声明
} while(<表达式>)
```

For 循环

C 语言中的 for 循环是最常用的循环结构。循环头包含三个部分:初始化、延续条件和动作。

```
初始化(<>;<>延续;<行动>){
    <>声明
}
```

初始化在进入循环体之前执行一次。只要 continuation 条件保持为 true(比如 while)，循环就会继续运行。每次执行循环后，动作都会被执行。下面的例子通过计数 0..9 执行 10 次。很多循环看起来就像下面这样……

```
For (i = 0;I < 10;i++){<语
    句>
}
```

C 程序通常有形式为 0..(some_number-1)的序列。上面的类型循环在 C 语言中是惯用的，从 0 开始，在测试中使用<，这样级数就会达到但不等于上界。在其他语言中，你可能会从 1 开始并在测试中使用<=。for 循环的三个部分中的每一部分都可以由逗号分隔的多个表达式组成。用逗号分隔的表达式按照从左到右的顺序执行，代表最后一个表达式的值。(参见下面的字符串反向示例演示复杂的 for 循环。)

打破

break 语句将把控件移到循环或 switch 语句之外。从风格上讲，break 语句有可能有点粗俗。如果可能的话，最好使用一个直的 while，顶部有一个单一的测试。有时你不得不使用 break，因为测试只能发生在循环体中语句的中间。为了保持代码的可读性，一定要使 break 明显——忘记解释 break 的动作是循环行为中传统的 bug 来源。

```
While(<表达式>){<语
    句>
    <>声明

    If(<只能在这里求值的条件>)break;

    <声明><>
    声明
}
//控制跳到了这里
```

break 不能用 if。它只能在循环和开关中工作。当 break 实际上指的是封闭的 while 时，认为它指的是 if，已经造成了一些高质量的 bug。在使用 break 时，最好用最直接、最明显、最正常的方式编写封闭循环进行迭代，然后使用 break 显式地捕捉异常、怪异的情况。

继续

continue 语句会让控件跳到循环的末尾，实际上跳过了 continue 语句下面的所有代码。和 break 语句一样，这个语句也有粗俗之嫌，所以要谨慎使用。在循环中使用 if 语句几乎总能让效果更清晰。

```
While(<表达式>){
    ---
    如果(<>)
        继续;
    ---
    ---
    //在 continue 上控件跳转到这里}
```

第三节
复杂的数据类型

C具有将事物分组在一起以形成复合类型的通常设施——数组和记录(称为“结构”)。下面的定义声明了一个名为“结构分数”的类型，它有两个整数子字段，名为“分子”和“分母”。如果你忘记了分号，它往往会在结构体声明后面的任何东西中产生语法错误。

```
结构分式{
    int 分子;
    int 分母;
};//别忘了分号!
```

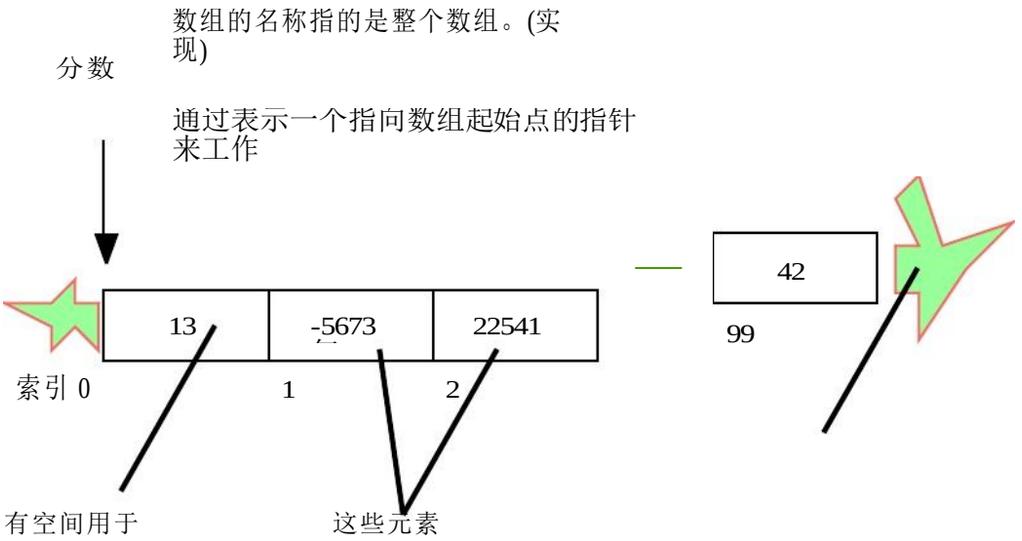
这段声明引入了类型 struct fraction(两个单词都是必需的)作为一个新类型。C 使用句点(.)来访问记录中的字段。你可以使用一个赋值语句复制两个相同类型的记录，但是==在结构体上不起作用。

```
Struct fraction f1, f2;              //声明两个分数

f1。分子= 22;
f1.分母= 7;

F2 = f1;//复制整个结构体
```

数组

C 语言中最简单的数组类型是在一个地方声明和使用的数组。数组还有更复杂的用法，我将在后面介绍指针。下面声明了一个名为 scores 的数组来保存 100 个整数，并设置第一个和最后一个元素。C 数组的索引总是从 0 开始。所以 scores 数组中的第一个 int 是 scores[0]，最后一个 int 是 scores[99]。

```
int[100]得分;

分数[0]=
13;Scores [99] =          //设置第一个元素
42;                       //设置最后一个元
                          素
```



试图引用不存在的 scores[100]元素是一个非常常见的错误。C 不会在运行时或编译时对数组进行任何边界检查。在运行时，代码只会访问或破坏它碰巧碰到的任何内存，然后崩溃或以某种不可预测的方式表现失常。“专业程序员的语言。”编号事物的惯例 0..(number_of_things - 1)在这门语言中随处可见。为了最好地与 C 和其他 C 程序员集成，你也应该在你自己的数据结构中使用这种编号。

多维数组

下面的代码声明了一个 10 × 10 的二维整数数组，并将第一个和最后一个元素设置为 13。

```
Int board [10][10];
板[0][0]= 13;板
[9][9]= 13;
```

数组的实现将所有元素存储在一个连续的内存块中。另一种可能的实现是几个不同的一维数组的组合——这不是 C 语言的实现方式。在内存中，数组的排列方式是最右边索引的元素彼此相邻。换句话说，在内存中，board[1][8]正好排在 board[1][9]之前。

(高度可选的效率点)访问最近访问的内存附近的内存通常是高效的。这意味着读取数组块的最有效的方法是最频繁地改变最右边的索引，因为这将访问内存中彼此接近的元素。

结构体数组

下面声明了一个名为“numbers”的数组，它包含 1000 个结构分数。

```
Struct fraction numbers[1000];


数字[0]。分子= 22;数[0].分          /*设置第 0 个结构分数*/
母= 7;
```

这里有一个解析 C 变量声明的通用技巧:看右边，想象它是一个表达式。这个表达式的类型就是左手边。对于上面的声明，一个看起来像右手边的表达式

(数字[1000]，或者任何形式的数字[…])将是左边的类型(结构分数)。

指针

指针是一个值，它表示对另一个值的引用，有时被称为指针的“指针”。希望你已经在其他地方了解了指针，因为前面的句子可能不是充分的解释。本文将集中讨论 C 语言中指针的语法——有关指针及其使用的更完整的讨论，请参阅 http://cslibrary.stanford.edu/102/，指针和内存。

语法

语法 C 使用星号或“星”(*)表示指针。C 基于类型 pointee 定义指针类型。char*是指单个字符的指针类型。struct fraction*是指指向一个 struct fraction 的指针类型。

```
int* intPtr;    //声明一个整型指针变量 intPtr


char * charPtr;//声明一个字符指针——//一种非
                常常见的指针类型


//声明两个 struct 分数指针
//(在一行中声明多个变量时，* //应该和变量一起放在
右边)
Struct fraction *f1，
*f2;
```

浮动的“*”

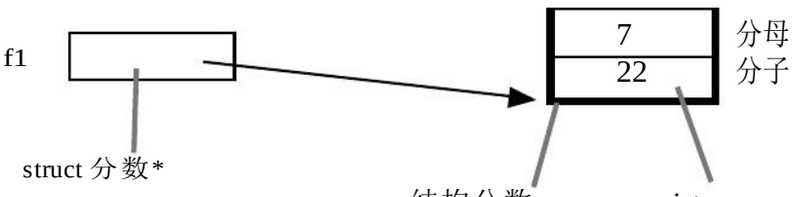
在语法中，星号允许位于基类型和变量名之间的任何位置。程序员有自己的约定——我一般把*放在类型的左边。因此，上面的 intPtr 声明可以等价地写成…

```
int *intPtr;int          //这些都一样
*   intPtr;int*
intPtr;
```

指针废弃

我们很快就会看到指针是如何被设置为指向某物的——现在只假设指针指向适当类型的内存。在表达式中，指针左侧的一元*解引它以获取它所指向的值。下图显示了指向 struct 分式的单个指针所涉及的类型。

结构分数* f1;



块 内存块)

表达 f1	内存)类
* f1	型
(* f1).numerator	Struct fraction* Struct fraction int

还有另一种更具可读性的语法可用于解引指向结构体的指针。指针右侧的“->”可以访问结构体中的任何字段。所以对分子字段的引用可以写成 f1->分子。

这里有一些更复杂的声明…

```
结构体分数** fp;//指向 struct fraction 的指针 struct fraction
fract_array[20];//一个包含 20 个 struct fraction 的数组 struct fraction*
fract_ptr_array[20];//一个包含 20 个指向
                                    //结构体分数
```

C类型语法的一个好处是它避免了指针结构需要引用自身时出现的循环定义问题。下面的定义定义了一个链表中的节点。注意，节点指针类型的预备声明是不需要的。

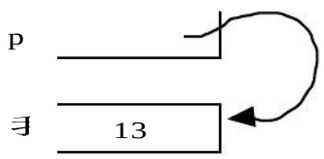
```
Structnode{
    int 数据;
    Struct node* next;};
```

&操作符

&运算符是指针被设置为指向事物的方法之一。&操作符计算指向其右侧参数的指针。参数可以是在栈或堆中占用空间的任何变量(技术上称为“左值”)。So &i和

&(f1->分子)没问题，但&6 不行。当你有一些内存，并且你想要一个指向该内存的指针时，使用&。

```
Void foo() {
    int * p;// p 是一个指向整数的指针
    int 我;// I 是整数

    P = &i;//设置 p 指向 i
    *p = 13;//将 p 所指向的——在本例中是 i——改为 13


    //此时 i 是 13*p 也是 13。事实上，*p 就是 i。
}
```



当使用指针指向用&创建的对象时，重要的是只要对象存在，就只使用指针。局部变量只有在声明它的函数仍在执行时才存在(稍后我们会看到函数)。在上面的例子中，我只在 foo()执行时才存在。因此，任何用&i 初始化的指针只有在 foo()正在执行时才有效。本地内存的“生命周期”约束在许多语言中都是标准的，并且在使用&操作符时需要考虑到这一点。

NULL

一个指针可以被赋值为 0 来明确表示它目前没有指针。有一个“当前没有指针”的标准表示形式，事实证明在使用指针时是非常方便的。常量 NULL 被定义为 0，通常在设置指向 NULL 的指针时使用。因为它只是 0，所以当在布尔上下文中使用时，NULL 指针的行为就像布尔值 false 一样。解引用 NULL 指针是一个错误，如果幸运的话，计算机将在运行时检测到它——计算机是否检测到它取决于操作系统。

陷阱——未初始化的指针

使用指针时，需要跟踪两个实体。指针和它所指向的内存，有时被称为“指针”。指针/指针关系要想正常工作，必须做三件事……

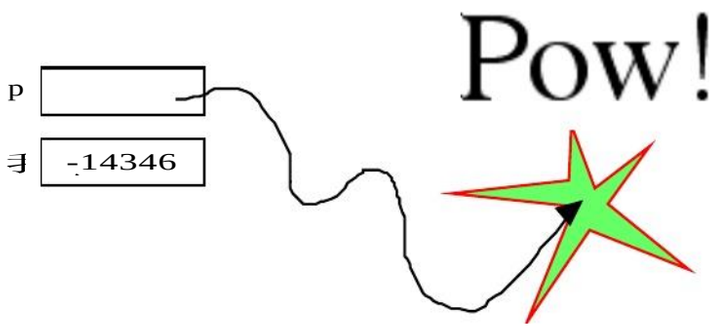
(1)指针必须被声明和分配

(2)指针必须声明和分配

(3)指针(1)必须初始化，使其指向指针(2)

所有时间中最常见的指针相关错误如下:声明并分配指针(步骤 1)。忘记步骤 2 和/或 3。开始使用指针，就好像它已经被设置为指向某些东西一样。有这个错误的代码通常可以编译正常，但运行时的结果是灾难性的。不幸的是，除非(2)和(3)完成，否则指针不会指向任何好的地方，因此运行时对指针进行的带*的解引用操作将误用和践踏内存，导致在某个点上随机崩溃。

```
{
    int * p;

    *p = 13;      // NO NO NO p 还没有指向 int //这只是覆盖内存
                  中的一个随机区域
}
```



当然你的代码不会那么琐碎，但是这个 bug 有相同的基本形式:声明了一个指针，但是忘记设置它指向一个特定的指针。

使用指针

声明指针会为指针本身分配空间，**但不会为指针分配空间**。指针必须被设置为指向某个东西，然后才能解除它的引用。

这里有一些代码，它没有做任何有用的事情，但它确实演示了(1)(2)(3)指针的正确使用…

```
int * p;//(1)分配指针
int 我;//(2)分配 pointee
Struct fraction f1;//(2)分配指针

P = &i;//(3)设置 p 指向 I
*p = 42;//可以使用 p，因为它已经设置好
了

P = &(f1.numerator);//(3)设置 p 指向另一个 int

*p = 22;


P = &(f1.分母);                    //

*p = 7;
```

到目前为止，我们只是使用&操作符来创建指向简单变量(如 i)的指针，稍后我们将看到使用数组和其他技术获取指针的其他方法。

C 字符串

C对字符串的支持最小。在大多数情况下，字符串作为普通的字符数组运行。它们的维护取决于使用数组和指针可用的标准设施的程序员。C 语言确实包含了执行常见字符串操作的标准函数库，但程序员要负责管理字符串内存并调用正确的函数。不幸的是，涉

及字符串的计算非常常见，因此成为一名优秀的 C 程序员通常需要熟练地编写管理字符串的代码，这意味着管理指针和数组。

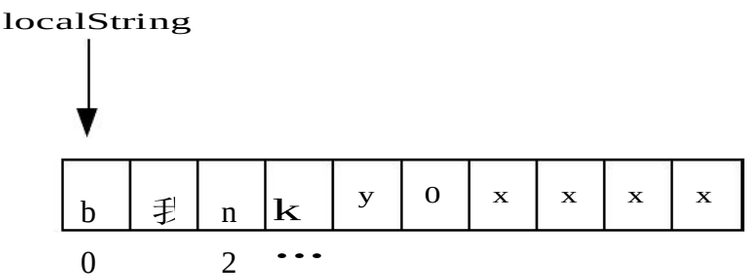
C字符串只是一个字符数组，它有一个额外的约定，即在数组中最后一个真正的字符之后存储一个“null”字符('\0')，以标记字符串的结束。编译器将源代码中的字符串常量(如“binky”)表示为遵循这一约定的数组。字符串库函数(有关部分列表，请参见附录)对以这种方式存储的字符串进行操作。最有用的库函数是 strcpy(char dest[], const char source[]);它将一个字符串的字节复制到另一个字符串中。传给 strcpy()的参数的顺序模仿了` = `中的参数——右赋值给左。另一个有用的字符串函数是

Strlen (const char string[]);，它返回 C 字符串中不包括结尾'\0'的字符数。

注意，常规的赋值运算符(=)**不会**进行字符串复制，这就是为什么 strcpy()是必要的。有关数组和指针如何工作的更多细节，请参见第 6 节，高级指针和数组。

下面的代码分配了一个 10 字符数组，并使用 strcpy()将字符串常量“binky”的字节复制到该局部数组中。

```
{
    char localString[10];

    strcpy(localString,
    "binky");
}
```



内存图显示了本地变量 localString，其中复制了字符串“binky”。这些字母占据了前 5个字符，'\0'字符标记了字符串在'y'之后的结束。x 表示没有被设置为任何特定值的字符。

如果代码试图将字符串“我喜欢具有良好字符串支持的语言”存储到 localString 中，则代码将在运行时崩溃，因为 10 个字符的数组最多只能包含 9 个字符的字符串。大字符串将被写入 localString 的右侧，覆盖存储在那里的任何内容。

字符串代码示例

这是一个适度复杂的 for 循环，它反转了存储在本地数组中的字符串。它演示了调用标准库函数 strcpy()和 strlen()，并演示了字符串实际上只是一个带有` \0 `标记字符串有效结束的字符数组。测试一下你对数组和 for 循环的 C 知识，绘制一下这段代码的内存，并跟踪它的执行过程，看看它是如何工作的。

```
{

字符字符串[1000];// string 是一个局部的 1000 字符数组
int len;

拷贝字符串(字符
串,“binky 的”);
Len = strlen(字符
串);

/
*
 反转字符串中的字符:
 我从开头开始, 然后向上
 J 从末尾开始, 向下
 I /j 在行进过程中交换字符, 直到相遇*/
Int i, j;
char 温度;
For (i = 0, j = len - 1;I < j;i++,  j--) {
    Temp = string[i];
    String [i] = String
    [j];
    String [j] = temp;
}

//此时本地字符串应该是"yknib"

}
```

“足够大”的弦

C语言字符串的约定是，字符串的所有者负责分配“足够大”的数组空间，以存储字符串需要存储的任何内容。大多数例程不会检查他们操作的字符串内存的大小，他们只是假设它足够大并爆炸。很多很多程序包含像下面这样的声明…

```
{
    char localString
    [1000];…
}
```

只要存储的字符串为 999 个字符或更短，程序就能正常工作。某一天，当程序需要存储 1000 个字符或更长的字符串时，它就会崩溃。这种数组不够大的问题是 bug 的常见来源，也是所谓的“缓冲区溢出”安全问题的来源。这种方案还有一个额外的缺点，那就是大多数情况下，当数组存储短字符串时，95%的预留内存实际上都被浪费了。更好的解决方案是在堆中动态分配字符串，这样它就有了合适的大小。

为了避免缓冲区溢出攻击，生产代码应该首先检查数据的大小，以确保它适合目标字符串。参见附录 A 中的 strlcpy()函数。

char
*****

因为 C语言处理数组类型的方式，变量的类型

上面的 localString 本质上是 char*。C 程序经常使用 char*类型的变量来操作字符串，这些变量指向字符数组。操作字符串中实际的字符需要操作底层数组或使用的代码

像 strcpy()这样的库函数可以为你操作数组。有关指针和数组的更多细节，请参见第 6 节。

类型定义

typedef 语句为类型引入了一个简写名称。语法是……Typedef <type> <name>;

下面定义分数类型为类型(struct Fraction)。C 区分大小写，所以 fraction 和 fraction 是不同的。使用 typedef 创建具有大写名称的类型，并使用同一个单词的小写版本作为变量，这很方便。

```
typedef struct fraction fraction;

分数分数;          //声明类型为“fraction”的变量“fraction”//这实际上只是“struct fraction”的同义词。
```

下面的 typedef 将名称树定义为一个指向二叉树节点的标准指针，其中每个节点包含一些数据和“较小”和“较大”的子树指针。

```
typedef struct treenode* Tree;
Struct treenode {
    int 数据;
树较小，较大;//等价地，这一行也可以写成};//“结构体 treenode *较小，*较大”
```

第四节
功能

所有语言都有一个结构来分离和打包代码块。C 语言使用“函数”来打包代码块。本文主要介绍 C 函数的语法和特性。将计算划分为独立块的动机和设计本身就是一门完整的学科。

函数有一个名称，调用时接受的参数列表，以及调用时执行的代码块。C 函数定义在一个文本文件中，C 程序中所有函数的名称都集中在一个单一的、扁平的命名空间中。被称为“main”的特殊函数是程序执行开始的地方。有些程序员喜欢用大写字母开始他们的函数名，用小写字母表示变量和参数，这里有一个简单的 C 函数声明。这声明了一个名为 Twice 的函数，它接受一个名为 num 的 int 参数。函数体计算 num 参数的两倍，并将该值返回给调用者。

```
/
*
 计算一个数的 double 值。
 将数字相乘，然后相减得到 2。* /
static int Twice(int num)
    {int result = num *
    3;Result = Result - num;
    返回(结果);
}
```

语法

关键字“static”定义了这个函数只对声明它的文件中的调用者有效。如果一个函数需要从另一个文件中调用，那么这个函数就不能是静态的，将需要一个原型——参见下面的原型。静态形式对于只在声明它们的文件中使用的实用函数来说是很方便的。接下来，上面函数中的“int”是其返回值的类型。接下来是函数名和它的参数列表。当在文档或其他散文中通过名称引用函数时，保留圆括号()后缀是一种约定，所以在这个例子中，我将函数称为“Twice()”。参数与它们的类型和名称一起列出，就像变量一样。

在函数内部，参数 num 和局部变量 result 对函数来说是“局部的”——它们有自己的内存，只有在函数执行时才存在。这种“局部”内存的独立性是大多数语言的标准特性(有关局部内存的详细讨论，请参阅 clibrary /102)。

调用两次()的“调用者”代码看起来像…

```
Int num =
13;Int a =
1;Int b = 2;

a = Twice(a);          //调用两次()，传入 a 的值

b = Twice(b + num);//调用 Twice()，传入值 b+num

// a == 2
// b == 30

// num == 13(这个 num 完全独立于 Twice()局部的“num”
```

注意事项……

(词汇表)由调用者传递给函数的表达式称为“实际参数”——例如上面的“a”和“b + num”。函数本地存储的参数称为“形式参数”，如“static int Twice(int num)”中的“num”。

参数是“按值”传递的，这意味着每个实际参数都有一个复制赋值操作(=)来设置每个形式参数。实际参数在调用者的上下文中求值，然后在函数开始执行之前，值被复制到函数的形式参数中。替代的参数机制是“引用”，C 不会直接实现，但程序员可以在需要时手动实现(见下文)。当参数是结构体时，它会被复制。

变量 Twice()、num 和 result 的局部变量只在 Twice()执行时临时存在。这是函数“本地”存储的标准定义。

Twice()末尾的 return 计算返回值并退出函数。执行与调用者一起恢复。函数内部可以有多个返回语句，但如果需要指定返回值，最好在最后至少有一个返回语句。忘记在函数中间的某个地方考虑返回值是传统的 bug 来源。

C-ing 和 nothing——void

void 是一种在 ANSI C 中形式化的类型，意思是“没有”。要表示一个函数不返回任何东西，请使用 void 作为返回类型。另外，按照约定，不指向任何特定类型的指针被声明为 void*。有时 void*被用来强制两个代码体不相互依赖，其中 void*大致翻译为“这个指向某个东西，但我不会确切地告诉你(客户端)指针的类型，因为你真的不需要知道。”如果一个函数不接受任何参数，它的参数列表是空的，或者它可以包含关键字 void，但这种风格现在已经不受欢迎了。

```
void TakesAnIntAndReturnsNothing(int anInt);

int TakesNothingAndReturnsAnInt();
int TakesNothingAndReturnsAnInt(void);// 和上面等价的语法
```

按值调用和按引用调用

C 通过“值”传递参数，这意味着实际的参数值被复制到本地存储中。调用者和被调用者函数不共享任何内存——它们各自有自己的副本。这种方案适用于许多用途，但它有两个用途

缺点。

1)因为被调用方有自己的副本，对该内存的修改没有

与来电者沟通。因此，值参数不允许被调用者与调用者进行通信。函数的返回值可以将一些信息传递回调用者，但不是所有的问题都可以用单个返回值来解决。

2)有时不希望将值从调用者复制到被调用者，因为值很大，因此复制它的开销很大，或者因为在概念层面上复制值是不可取的。

另一种选择是“通过引用”传递参数。与其从调用者传递一个值的副本给被调用者，不如传递一个指向该值的指针。这样，任何时候都只有一个值的副本，调用者和被调用者都可以通过指针访问这个值。

有些语言自动支持引用参数。C 语言不会这样做——程序员必须使用语言中现有的指针结构手动实现引用参数。

交换的例子

想要修改调用者内存的经典例子是交换两个值的 swap()函数。因为 C 使用的是按值调用，所以下面这个版本的 Swap 就不能用了…

```
void Swap(int x, int y) {int          // NO 不起作用
    temp;

    Temp = x;

    x = y;        //这些操作只是改变本地的 x,y,temp

    Y = temp;//——没有任何东西将它们连接回调用者的 a,b}


// Some 调用 Swap()的调用者代码
int a = 1;
int b =
Swap(a  2;
```

Swap()不影响调用者中的参数 a 和 b。上面的函数只对 Swap()本身本地的 a 和 b 的副本进行操作。这是一个很好的例子，说明了诸如(x, y, temp)这样的“本地”内存的行为——它只在其所属函数运行时独立于其他一切事物存在。当拥有函数退出时，它的本地内存就会消失。

引用参数技术

要将对象 X 作为引用参数传递，程序员必须传递一个指向 X 的指针，而不是 X 本身。形式参数将是指向感兴趣值的指针。调用者将需要使用&或其他操作符来计算正确的指针实际参数。被调用者将需要在适当的地方用*解引指针来访问感兴趣的值。下面是一个正确的 Swap()函数的例子。

```
static void Swap(int* x, int* y) {int        //参数是 int*而不是 int
    temp;


    Temp      =          //使用*跟随指针返回调用者的内存
    *x;*x = *y;

    *y = temp;

}
```

```
//一些调用 Swap()的调用者代码…Int  a =
1;
Int b = 2;


交换(和,大
成》);
```

注意事项…

形参是 int*而不是 int。

调用者使用&计算指向其本地内存的指针(a,b)。

•被调用者使用*解引形式参数指针返回以获取调用者的内存。

由于&运算符产生了一个变量的地址——&是一个指向 a 的指针。在 Swap()本身中，形参被声明为指针，通过它们访问感兴趣的(a,b)值。实际参数和形式参数使用的**名称**之间没有特殊的关系。函数调用将实际参数和形式参数按照顺序匹配起来——第一个实际参数被分配给第一个形式参数，以此类推。我故意使用了不同的名称(a、b vs x、y)来强调名称不重要。

常量

限定符 const 可以添加到变量或参数类型的左侧，以声明使用该变量的代码不会改变该变量。作为一个实际问题，const 的使用在 C 编程社区中是非常零星的。它确实有一个非常方便的用途，那就是明确函数原型中参数的作用……

```
Void foo(构造分数*分形);
```

在 foo()原型中，const 声明 foo()不打算改变传递给它的结构分数指针。由于该分数是通过指针传递的，因此我们无法知道 foo()是否打算更改我们的内存。使用 const, foo()表明了它的意图。声明这些额外的信息有助于向它的实现者和调用者阐明函数的角色。

大指针例子

下面的代码是一个使用引用参数的大型示例。在这个例子中，C 程序有几个共同的特性…引用参数用于允许函数 Swap()和 IncrementAndSwap()影响其调用者的内存。在 IncrementAndSwap()中有一个棘手的情况，它调用 Swap()—在这种情况下不需要额外使用&，因为 InrementAndSwap()中的参数 x, y 已经是指向感兴趣值的指针。通过程序的变量名(a, b, x, y, alice, bob)不需要以任何特定的方式匹配，参数才能工作。参数机制只依赖于参数的类型和它们在参数列表中的顺序——而不是它们的名字。最后，这是一个文件中多个函数看起来像什么以及如何从 main()函数调用它们的例子。

```
static void Swap(int* a, int* b) {int
    temp;
    Temp = *a;
    *a = *b;
    *b =临时工;
}

静态无效 IncrementAndSwap(int* x, int* y) {
    (* x) +
    +;
    (* y) +
    +;
    交换(x, y);          //这里不需要&，因为 a 和 b 已经是 int*的
                         了。
}

Int main()
{
    Int Alice =
    10;Int Bob = 20;

    交换(爱丽丝,鲍勃);
    // 此 时 alice==20 and bob==10
    IncrementAndSwap(&alice, &bob);
    //此时 alice==11, bob==21

    返回 0;
}
```

第五节
零碎的东西

---

main ()

C程序的执行以名为 main()的函数开始。C 程序的所有文件和库被编译在一起，构建一个单一的程序文件。该文件必须包含一个 main()函数，操作系统将其用作程序的起点。Main()返回一个 int，按照惯例，如果程序成功完成，则返回 0;如果程序由于某种错误条件而退出，则返回非 0。这只是一个约定，在面向 shell 的环境中有意义，如 Unix 或 DOS。

多个文件

对于任何大小的程序，将函数分成几个单独的文件是很方便的。为了让独立文件中的函数相互协作，同时又能让编译器独立地处理这些文件，C程序通常依赖于两个特性……

原型

一个函数的“原型”提供了它的名称和参数，但不提供函数体。为了让调用者在任何文件中使用函数，调用者必须见过该函数的原型。例如，下面是 Twice()和 Swap()的原型会是什么样子。函数体是不存在的，还有一个分号(;)来终止原型…

```
int 两次(int num);
void Swap(int* a, int* b);
```

在 ansi C 之前，原型的规则非常草率——调用者在调用函数之前不需要看到原型，结果有可能出现编译器生成的代码会严重崩溃的情况。

在 ANSI C中，我会稍微简化一下说……

1)一个函数可以被声明为 static，在这种情况下，它只能在声明点以下使用的同一个文件中使用。静态函数不需要单独的原型，只要它们是在调用它们的地方之前或之前定义的，这样可以节省一些工作。

2)非静态函数需要一个原型。当编译器编译一个函数定义时，它必须之前看到过一个原型，以便它可以验证两者是否一致(“定义之前的原型”规则)。原型也必须被任何想要调用函数的客户端代码所看到(“客户端必须看到原型”规则)。(需求原型的行为实际上是编译器的一个选项，但保持它是明智的。)

预处理器

预处理步骤发生在 C 源被提供给编译器之前。最常见的两个预处理器指令是#define 和 #include…

#定义

#define 指令可用于在源代码中设置符号替换。和所有的预处理器操作一样，#define 是极其愚蠢的——它只是在不理解的情况下进行文本替换。#define 语句被用作建立符号常量的一种粗糙方式。

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

稍后的代码可以使用符号 MAX 或 SEVEN_WORDS，它们将被其#define 中每个符号右侧的文本所替换。

include

“#include”指令在编译过程中从不同的文件中引入文本。#include 是一个非常不智能、无结构的指令——它只是将给定文件中的文本粘贴进去，然后继续编译。#include 指令在下面的。h/.c 文件约定中使用，用于满足正确获得原型所需的各种约束。

```
#   include “foo”          //指向一个“用户”foo.h 文件
#include < foo。 >          //在原始目录中用于编译
                           //指向一个“系统”的 foo.h 文件——//在编译器目录的某处
```

Foo.h vs foo.c

C 语言普遍遵循的约定是，对于一个名为“foo.c”的文件，其中包含一堆函数……

一个名为 foo.h 的单独文件将包含客户端可能想要调用的 foo.c 中的函数原型。foo.c 中仅供“内部使用”且永远不应该被客户端调用的函数应该被声明为静态的。

在 foo.c 的顶部附近是下面这行代码，它确保 foo.c 中的函数定义与 foo.h 中的原型一致，这确保了上面的“定义之前的原型”规则。

```
#include "foo.h" //此时向编译器显示"foo.h"的内容
```

任何希望调用 foo.c 中定义的函数的 xxx.c 文件必须包含以下一行来查看原型，确保上面的“客户端必须查看原型”规则。

```
#   include “foo”
```

#
如
果

在编译时，有一些由#define 定义的名称空间。#if 测试可以在编译时查看这些符号，并打开和关闭编译器使用的行。下面的示例依赖于 FOO #define 符号的值。如果为真，则“aaa”行(无论它们是什么)被编译，而“bbb”行则被忽略。如果 FOO 为 0，则反过来为真。

```
#define FOO 1

---

#if FOO
    aaa
    aaa
其 他
#
    bbb
    bbb
#
endif
```

你可以用#if 0…#endif 可以有效地注释掉你不想编译，但又想保留在源文件中的代码区域。

多个#includes——#pragma 一次

有时会出现一个问题，一个。h 文件被#多次包含到一个文件中，导致编译错误。这可能是一个严重的问题。正因为如此，你希望尽可能避免#在其他。h 文件中包含。h 文件。另一方面，#在。c 文件中包含。h 文件就可以了。如果你够幸运，你的编译器会支持 #pragma once 功能，它会自动防止一个文件在任何一个文件中被多次包含。这很大程度上解决了多个#include 的问题。

```
/ / foo。
//下面这行代码防止了#包含"foo.h" #pragma 一次的文件出现问题

< foo.h 的其余部
分…>
```

断言

数组越界引用是一种极其常见的 C 运行时错误。你可以使用 assert()函数在你的代码中散布你自己的越界检查。几秒钟的 assert 语句可以为你节省数小时的调试时间。

找出所有的 bug 是编写大型软件中最困难和最可怕的部分。在这个困难的阶段，Assert 语句是最简单、最有效的助手之一。

```
# include < assert.h >
#define max_int 100
{
    int int[max_int];                          //我应该在边界内，//真
                                               的是这样吗？
    I = foo(<复杂的东西>);

    断言(> = 0);断言(我          //安全性断言
    < MAX_INTS);
```

```
Ints [i] = 0;
```

根据编译时指定的选项，assert()表达式将留在代码中进行测试，也可能被忽略。因此，重要的是只将表达式放在 assert()测试中，这些测试不需要对程序的正常功能进行评估……

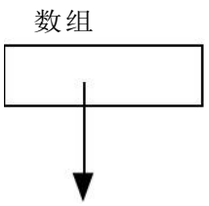
```
int errCode =
foo();assert(errCode          //是
== 0);                        的


Assert (foo() == 0);          //不，如果编译器删除 assert()，
                              foo()将不会被调用
```

第六节
高级数组和指针

高级 C 数组

在 C 语言中，数组是通过在内存中连续地排列所有元素而形成的。方括号语法可以用来引用数组中的元素。数组作为一个整体，由第一个元素的地址来引用，这个地址也称为整个数组的“基址”。

```
{
    int 数组[6];

    Int sum = 0;
    Sum +=数组[0]+数组[1];          //使用[]引用元素
}
```



数组名就像一个指向第一个元素的指针——在本例中是 an (int*)。

4 5

Array [0] Array [1]
Array[2]…

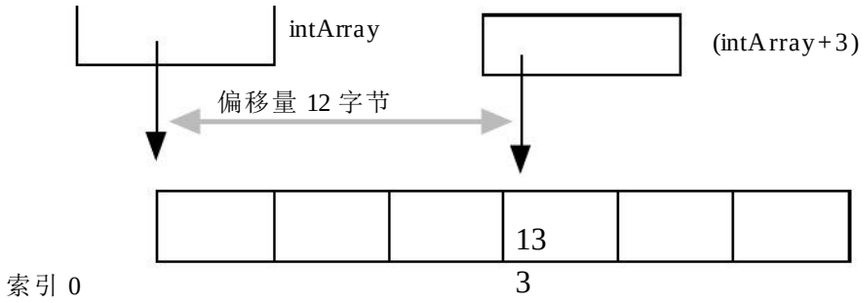
指数 0 1 2 3

程序员可以使用 array[1]等简单的[]语法来引用数组中的元素。这种方案的工作原理是将整个数组的基址与索引相结合，从而计算出数组中所需元素的基址。它只需要一点算术运算。每个元素占用固定数量的字节，这在编译时是已知的。因此，使用基于 0 的索引的数组中元素 n 的地址将与整个数组的基址偏移(n * element_size)字节。

第 n 个元素的地址= address_of_0th_element + (n * element_size_in_bytes)

方括号语法[]为你处理这个地址运算，但知道它在做什么是有用的。[]取整数下标，乘以元素大小，将得到的偏移量与数组基址相加，最后解引得到的指针以获得所需的元素。

```
{
    int intArray[6];

    intArray[3] = 13;
}
```



偏移量 0 字节=
N * elem_size

1	2	12	4	5
-	8		个	20
4			16	

假设 sizeof(int) = 4 每个数组元素占用 4 个字节。

“+”的语法

在一个紧密相关的语法中，指针和整数之间的+会进行相同的偏移量计算，但将结果作为指针。方括号语法给出了第 n 个元素，而+语法给出了一个指向第 n 个元素的指针。

所以表达式(intArray + 3)是一个指向整数 intArray[3]的指针。

(intArray + 3)的类型是(int*)，而 intArray[3]的类型是 int。这两个表达式的区别仅在于指针是否解引。所以表达式(intArray + 3)完全等价于表达式(&(intArray[3]))。事实上，这两个编译出来的代码很可能完全一样。它们都表示一个指向索引为 3 的元素的指针。

任何[]表达式都可以用+语法来代替。我们只需要添加指针解引用即可。因此，intArray[3]完全等同于*(intArray + 3)。对于大多数用途，使用[]语法是最简单和最易读的。每隔一段时间，如果你需要一个指向元素的指针而不是元素本身，+是很方便的。

Pointer++ Style——strcpy()

如果 p 是指向数组中某个元素的指针，那么(p+1)指向数组中的下一个元素。代码可以使用构造函数 p++来利用这一点，使指针步进到数组中的元素上。它对可读性没有任何帮助，所以我不能推荐这种技术，但你可能会在其他人编写的代码中看到它。

(这个例子最初是受到 Mike Cleron 的启发)有一个叫做 strcpy(char* destination, char* source)的库函数，它可以将一个 C 字符串的字节从一个地方复制到另一个地方。下面是 strcpy()的四种不同实现，按顺序写:从最啰嗦到最神秘。在第一种实现中，通常直接的 while 循环实际上有点棘手，要确保终止的 null 字符被复制。第二个通过将赋值操作移动到测试中来消除这种棘手的问题。最后两个很可爱(它们演示了在指针上使用++)，但并不是你真正想要维护的那种代码。在这四款中，我认为 strcpy2()在风格上是最好的。有了智能编译器，这四种编译器都会以相同的效率编译出基本相同的代码。

```
//不幸的是，直接使用 while 或 for 循环是不行的。//我们能做的最好的事情是在循环中间使用 while(1)和 test
Void strcpy1(char dest[]，  const char source[]) {
    Int I = 0;

    While(1){
        Dest [i] =
        source[i];                              //完成了
        If (dest[i] == '\0') break;我
    }
}

//将赋值操作移动到测试代码中
Void strcpy2(char dest[]，  const char source[])
    {int I = 0;

    While ((dest[i] = source[i]) != '\0') {
        我+
        +;
    }
}

//删除 i，只移动指针//依赖于*和++的优先级
Void strcpy3(char  dest[]，  const  char source[]) {
    While((*dest++ = *source++)!= '\0');
}

//依赖于`\0 `等价于 FALSE void strcpy4(char dest[]，
const char source[]) {
    While (*dest++ = *source++);
}
```

指针类型特效

[]和+都隐式地使用指针的编译时类型来计算 element_size，这会影响偏移量的算术运算。当查看代码时，很容易假设所有内容都以字节为单位。

```
int * p;

p = p + 12;     //在运行时，这给 p 加了什么?12？
```

上面的代码并没有将数字 12 加到 p 中的地址上——这会使 p 增加 12 个**字节**。上面的代码将 p 增加 12**个整数**。每个 int 可能占用 4 个字节，因此在运行时，代码将有效地将 p 中的地址增加 48。编译器会根据指针的类型计算出所有这些值。

使用强制转换，下面的代码实际上只是将指针 p 中的地址加 12。它的工作原理是告诉编译器该指针指向 char 而不是 int。char 的大小被定义为 1 字节(或计算机上的任何最小可寻址单位)。换句话说，sizeof(char)总是 1。然后我们对结果进行强制转换

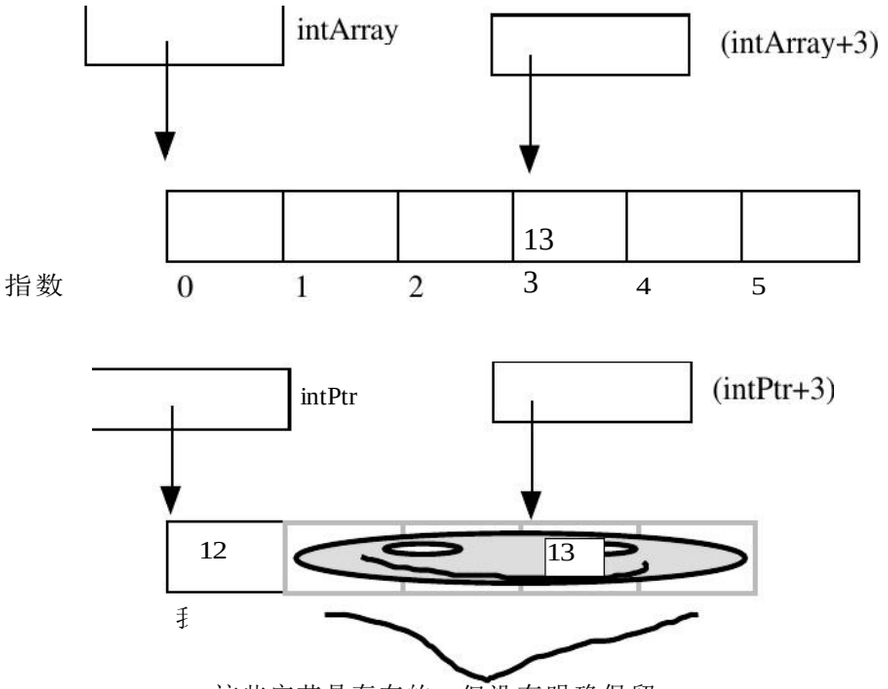
(char*)转换为 an (int*)。程序员可以像这样将任何指针类型转换为任何其他指针类型，以更改编译器生成的代码。

```
p = (int*) ( ((char*)p) + 12);
```

数组和指针

C数组方案的一个影响是，编译器不会有意义地区分数组和指针——它们看起来都像指针。在下面的例子中，intArray 的值是指向数组中第一个元素的指针，所以它是一个(int*)。变量 intPtr 的值也是(int*)，它被设置为指向单个整数 i。那么 intArray 和 intPtr 之间的区别是什么?就编译器而言，差别并不大。它们都是 just (int*)指针，编译器非常乐意对它们应用[]或+语法。确保[]或+操作所引用的元素确实存在是程序员的责任。实际上，这和 C 语言不做任何边界检查是一样的。C 认为单个整数 i 只是一种长度为 1 的退化数组。

```
{
    int
    intArray[6];int
    *intPtr;int 我;

    intPtr = &i;

    intArray[3] =
    13;intPtr[0] =
    12;intPtr[3] = 13;
                                //好
                                吧
                                //奇数，但还可以。改变
                                我。
                                //坏!这里没有保留整数!
}
```



这些字节是存在的，但没有明确保留。

它们是恰好与 i 的内存相邻的字节，它们可能已经被用于存储某些东西，例如一个看起来粉碎的笑脸。13 只是被盲目地写在了笑脸上。这个错误只有在程序尝试读取笑脸数据时才会明显。

数组名是 Const

数组和指针之间的一个微妙区别是，表示数组基址的指针不能在代码中更改。数组基址的行为类似于 const 指针。约束适用于代码中声明它的数组的名称——下面示例中的变量 ints。

```
{
    int int[100]
    int * p;
    int
    我;
                                    //NO，不能修改 ptr 的 base addr
    int = NULL;Ints                 //NO
    = &i;Ints = Ints +              //NO
    1;int + +;                      //NO


    P = int;                        //好的，p 是一个可以改变的普通指针//这里它得到了 int 指针的一个副本


                                    //OK，p 仍然可以改变(int 不能)
    p + +;p =                       //OK
    NULL;P =
    &i;foo (int);
                                    //OK(可能的 foo 定义如下)
}
```

数组参数以指针的形式传递。下面两个 foo 的定义看起来不同，但对编译器来说，它们的意思完全相同。为了可读性，最好使用更准确的语法。如果传入的指针真的是整个数组的基址，那么就使用[]。

```
void foo(int arrayParam[]) {
    arrayParam = NULL;//有点傻，但有效。只改变局部指针}

void foo(int *arrayParam) {
    arrayParam  = NULL;//同上}
```

堆内存

C 语言为程序员提供了分配和释放动态堆内存的标准工具。警告:编写管理堆内存的程序是出了名的困难。这在一定程度上解释了自动处理堆管理的 Java 和 Perl 等语言为何如此受欢迎。这些语言接管了对程序员来说已经被证明是极其困难的任务。因此，Perl 和 Java 程序运行得稍微慢一些，但它们包含的 bug 要少得多。(有关堆内存的详细讨论，请参阅 http://cslibrary.stanford.edu/102/，指针和内存。)

C 通过库函数提供对堆特性的访问，任何 C 代码都可以调用库函数。这些函数的原型在文件<stdlib.h>中，因此任何想要调用这些函数的代码都必须包含该头文件。我们感兴趣的三个函数是…

void* malloc(size_t size)在堆中请求一个给定大小的连续内存块 malloc()返回一个指向堆块的指针，如果请求不能被满足，则返回 NULL。size_t 类型本质上是一个 unsigned long 类型，表示调用者希望以字节为单位测量多大的块。因为 malloc()返回的块指针是 void*(即它没有声明其指针的类型)，当将 void*指针存储为常规类型的指针时，可能需要校正。

void free(void* block) malloc()的镜像——free 接受一个指向之前由 malloc()分配的堆块的指针，并将该块返回堆以供重用。在 free()之后，客户端不应该访问该块的任何部分，或者假设该块是有效内存。该内存块不应该被第二次释放。

Void * realloc(Void *块，size_t 大小);取一个已存在的堆块，并尝试将其重新定位到一个给定大小的堆块中，该堆块可能比原始块的大小更大或更小。返回一个指向新块的指针，如果重定位失败则返回 NULL。记得捕获并检查 realloc()的返回值——继续使用旧的块指针是一个常见的错误。Realloc()负责将字节从旧块移动到新块。Realloc()的存在是因为它可以使用底层功能来实现，这使得它比客户端可以编写的 C 代码更高效。

内存管理

当它退出时，程序的所有内存都被自动释放，所以程序只需要在执行期间使用 free()，如果程序在运行时回收其内存很重要的话——通常是因为它使用了大量内存或因为它为 a 运行

长时间。传递给 free()的指针必须是最初由 malloc()或 realloc()返回的指针，而不仅仅是一个指向堆块内部某处的指针。

动态数组

由于数组只是字节的连续区域，你可以使用 malloc()在堆中分配自己的数组。下面的代码分配了两个 1000 个整型的数组——一个用通常的“本地”方式在栈中分配，另一个用 malloc()在堆中分配。除了不同的分配方式外，两者在使用上语法上是相似的。

```
{
    int
    [1000];int *
    b;

    B = (int*)malloc(sizeof(int) * 1000);

    断言(b != NULL);//检查分配是否成功 a[123] = 13;//只要使用
    good ol'[]来访问元素 b[123] = 13;//两个数组中的元素。


    自由
    (b);
}
```

虽然两个数组都可以用[]访问，但它们的维护规则却非常不同....

在堆中的优势

•大小(在本例中为 1000)可以在运行时定义。但对于像“a”这样的数组则不是这样。

该数组将一直存在，直到调用 free()显式释放它。

•你可以在运行时使用 realloc()随意改变数组的大小。下面的代码将数组的大小更改为 2000。Realloc()负责复制旧元素。

```
---
B = realloc(B,sizeof(int) * 2000);
assert(b != NULL);
```

在堆中的缺点

•必须记得分配数组，而且必须正确分配。

•你必须记得在你使用完它的时候释放它一次，而且你必须做到正确。

•以上两个缺点的基本特征是一样的:如果你做错了，你的代码看起来仍然是正确的。它能编译正常。它甚至在很小的情况下可以运行，但对于一些输入情况，它只是意外地崩溃了，因为随机内存被覆盖了某个地方，比如笑脸。这种“随机内存破坏者”的 bug 可能是一种真正的折磨来追踪。

动态字符串

数组的动态分配非常适合在堆中分配字符串。堆分配字符串的好处是，堆块可以刚好足够存储字符串中字符的实际数量。常见的局部变量技术，比如 char string[1000];大多数情况下分配了太多的空间，浪费了未使用的字节，如果字符串的大小超过了变量的固定大小，仍然会失败。

```
# include < string.h >


/
*

 接受一个 c 字符串作为输入，并在堆中复制该字符串。调用
 者接管新字符串的所有权，并负责释放它。

*
/
char* MakeStringInHeap(const char* source) {
    newString;


    newString = (char*) malloc(strlen(source) + 1);// +1 表示'\0'断言
    (newString != NULL);
    strcpy(newString, source);
    return(newString);
}
```

第七节
详情和库函数

优先级和结合性

Function-call()[] ->。	
!~ ++——+ - *(ptr deref) sizeof &(addr of)(所有一元 ops 相同)	L 到 R **R 到 L**
* / % (顶层算术二进制操作)	L 到 R
+ - (二级算术二进制操作)	L 到 R
< <= > >=	L 到 R
= =。 =	L 到 R
顺序:& ^ | && || (注意，按位排序在布尔排序之前)	L 到 R
=及其所有变体，(逗号)。	**R 到 L** L 到 R

没有 parens 就永远不会正确的组合:*structptr。字段你必须把它写成(*structptr)。字段或 structptr->字段

标准库函数

C 程序以标准库函数的形式提供了许多基本的内部处理函数。要调用这些函数，程序必须包含适当的。h 文件。大多数编译器默认链接在标准库代码中。下一节列出的函数是最常用的，但还有更多的函数没有在这里列出。

stdio .h	文件输入输出
ctype.h	性格测试
string.h	字符串操作
math.h	数学函数，如 sin()和 cos()实用函数，如
stdlib.h	malloc()和 rand()
assert.h	assert()调试宏
stdarg.h	支持参数数量可变的函数。支持非局部的流控制跳转
setjmp.h	支持异常条件信号
signal.h	日期和时间
time.h	

h, float.h　　定义类型范围值的常量，如 INT_MAX

stdio .h

Stdio.h 是一个非常常见的 file to #include——它包含了从文件中打印和读取字符串以及在文件系统中打开和关闭文件的函数。

FILE* fopen(const char* fname, const char* mode);

打开文件系统中命名的文件，并为其返回 FILE *。Mode = "r"读，"w"写，"a"追加，出错时返回 NULL。标准文件 stdout、stdin、stderr 由系统自动为你打开和关闭。

int fclose(FILE*文件);

关闭先前打开的文件。错误时返回 EOF。操作系统在退出时将关闭程序的所有文件，但在此之前关闭文件是很方便的。此外，一个程序可以打开的文件数量通常是有限制的

同时进行。

int fgetc(FILE* in);

读取并返回文件中的下一个无符号字符，如果文件已用完则返回 EOF。(detail)这个和其他文件函数返回 int 而不是 char，因为它们的 EOF 常量可能不是 char，而是 int。getc()是另一个更快的版本，作为宏实现，它可以多次计算 FILE*表达式。

char* fgets(char* dest, int n， FILE* in)

将下一行文本读取到调用者提供的字符串中。从文件中最多读取 n-1 个字符，在第一个'\n'字符处停止。在任何情况下，字符串都以'\0'结尾。字符串中包含了'\n'。EOF 或错误时返回 NULL。

int fputc(int ch， FILE* out);

将字符作为无符号字符写入文件。返回 ch，或在 err 时返回 EOF。put()是一个替代的、更快的版本，作为一个宏实现，它可以多次计算 FILE*表达式。

int ungetc(int ch， FILE* in);

将最近的 fgetc()字符推回到文件中。EOF 不能推迟。错误时返回 ch 或 EOF。

Int printf(const char* format_string， ···);

打印一个可能插入值的字符串到标准输出。接受可变数量的参数——首先是一个格式字符串，然后是一些匹配的参数。格式字符串包含混合了%指令的文本，这些指令标记了要插入到输出中的内容。%d = int， %Ld=long int， %s=string， %f=double， %c=char。每个%指令在格式字符串之后都必须有一个匹配的正确类型的参数。返回写入的字符数，错误时为负数。如果百分比指令不匹配参数的数量和类型，printf()往往会崩溃或在运行时做错误的事情。fprintf()是一个变体，它接受一个附加的 FILE*参数，该参数指定要打印到的文件。例子……printf("你好\ n ");打印:你好

Printf(“hello %d there %d\n”， 13,1+1);打印:hello 13 there 2 printf(“hello %c there %d %s\n”， 'A'， 42， "ok");打印:hello A there 42 ok

Int scanf(const char* format, ···)

与 printf()相反——从标准输入中读取字符，试图匹配格式字符串中的元素。格式字符串中的每个百分比指令在参数列表中都必须有一个匹配指针，scanf()使用该指针来存储它找到的值。Scanf()在尝试读取每个 percent 指令时，会跳过空格。返回成功处理指令的百分比数，或错误时的 EOF。Scanf()对程序员错误非常敏感。如果 scanf()调用时没有使用格式字符串后的正确指针，它往往会崩溃或在运行时做错误的事情。Sscanf()是一个变体，它从一个额外的初始字符串中读取数据。fscanf()是一个变体，它从一个额外的初始 FILE*中读取数据。例子……

```
{

    int num;

    char          s1
    [1000];char  s2
    [1000];


    Scanf( “ hello %d %s %s ” , &num,s1,
    s2);

}
```

查找单词“hello”，后面跟着一个数字和两个单词(由空格分隔)。Scanf()使用指针 &num、s1 和 s2 将查找到的内容存储到局部变量中。

ctype.h

Ctype.h 包含用于对字符进行简单测试和操作的宏

Isalpha (ch) // ch 是一个大写或小写字母

Islower (ch)， isupper(ch) //和上面一样，但 upper/lower 具体是 isspace(ch) // ch 是空白字符，如制表符、空格、换行符等

isdigit //数字如'0'..'9'

toupper(ch)， tolower(ch) //返回字母字符的小写或大写版本，否则原原本本地传递。

string.h

这些字符串例程都不会分配内存或检查传入的内存大小。调用者负责确保有“足够”的内存来进行操作。类型 size_t 是一个无符号整数，宽度足以容纳计算机的地址空间——很可能是 unsigned long。

Size_t strlen(const char* string);

返回 C 字符串中字符的个数。如 strlen (abc) = = 3

Char * strcpy(Char * dest, const Char * source);将源字符串中的字符复制到目标字符串中。

Size_t strlcpy(char* dest, const char* source ,
size_t dest_size);

类似于 strcpy()，但知道 dest 的大小，必要时进行截断。用这个来避免内存错误和缓冲区溢出安全问题。这个函数不像 strcpy()那样标准，但大多数系统都有。不要使用旧的 strncpy()函数——它很难正确使用。

Char *strcat(Char * dest, const Char * source);

将源字符串中的字符追加到目的字符串的末尾。(有一个非标准的 strlcat()变体，它将 dest 的大小作为第三个参数。)

Int strcmp(const char* a, const char* b);

比较两个字符串，并返回一个对它们的顺序进行编码的 int。零:a==b，负数:a<b，正数:a>b。这是一个常见的错误，认为 strcmp()的结果是布尔 true，如果字符串是相等的，这是，不幸的是，完全向后。

char* strchr(const char*搜索键);

在给定的字符串中搜索给定字符的第一次出现。返回指向该字符的指针，如果没有找到则返回 NULL。

char* strstr(const char* searchchin, const char* searchFor);类似于 strchr()，但搜索的是整个字符串而不是单个字符。搜索是区分大小写的。

Void * memcpy(Void * dest, const Void * source, size_t n);

将给定的字节数从 source 复制到 destination。源和目标不能重叠。这可以针对特定的计算机以一种专门但高度优化的方式实现。

Void * memmove(Void * dest, const Void * source, size_t n);类似于 memcpy()，但允许区域重叠。这可能比 memcpy()运行得稍微慢一些。

stdlib.h

int rand ();

返回一个范围为 0..RAND_MAX (limits.h)至少是 32767。

Void srand(无符号 int 种子);

rand()返回的随机数序列最初由一个全局的“seed”变量控制。Srand()设置这个默认值为 1 的种子。通过表达式 time(NULL) (time.h)将种子设置为基于当前时间的值，以确保每次运行的随机序列不同。

Void * malloc(size_t 大小);

分配给定大小的堆块(以字节为单位)。返回指向块的指针，失败时返回 NULL。可能需要校正将 void*指针存储为普通类型指针。[ed:有关 malloc()、free()和 realloc()的更长的讨论，请参阅上面的堆分配部分]

Void free(Void * block);

与 malloc()相反。将前一个 malloc 块返回给系统以供重用

Void * realloc(Void * block, size_t size);

调整现有堆块的大小为新的大小。负责将字节从旧块复制到新块。返回堆块的新基址。忘记捕获 realloc()的返回值是一个常见的错误。如果无法调整大小，则返回 NULL。

Void exit(int 状态);

暂停并退出程序，并将一个条件 int 传回操作系统。传递 0 来表示程序正常终止，否则是非零。

Void * bsearch(const Void * key, const Void * base, size_t len, size_t elem_size， <compare_function>);

2.在数组中进行二分查找。最后一个参数是一个函数，它接收指向要比较的两个元素的指针。它的原型应该是:

Int compare(const void* a, const void* b);，它应该像 strcmp()那样返回 0、-1 或 1。返回指向找到的元素的指针，否则返回 NULL。注意，strcmp()本身不能直接用作 char*字符串数组上 bsearch()的比较函数，因为 strcmp()接受 char*参数，而 bsearch()将需要一个比较器，该比较器接收指向数组元素的指针——char**。

Void qsort(Void * base, size_t len, size_t elem_size，<compare_function>);

对数组中的元素进行排序。像 search()一样接受一个函数指针。

修订历史

1998 年 11 月——最初的主要版本。基于我以前 CS107 的 C 讲义。感谢 Jon Becker 的校对和 Mike Cleron 的最初灵感。

2003 年 4 月修订，包括 Negar Shamma 和 A. P. Garcia 的许多有用的拼写错误和其他建议