第3章　阅读“C”程序

学习阅读用“C”语言编写的程序是必须的，否则就无法阅读UNIX的源代码。

与自然语言相类似，读比写要容易一些。即使如此，你仍然需要非常细心地学习那些巧妙之处。

与“C”语言直接相关的“UNIX文献”有两种，它们是：

《C参考手册》，作者是丹尼斯·里奇

《C语言程序设计导引》，作者是布里安·柯林汉

从现在开始，你就应当阅读这些文献，并且要尽可能一次又一次地反复阅读它们，这样才会愈来愈深入地了解C语言。

学习UNIX源代码并不要求你能够编写“C”程序。但是，如果有机会，那么你应当至少编写一些小的C语言程序。这是一种学习程序设计语言的良好方法，使用这种方法你会较快地理解如何正确使用下列的“C”语言组成部分：

分号；

“=”和“==”；

“{”和“}”；

“++”和“--”；

说明；

寄存器变量；

“if”和“for”语句；

等等

你将发现“C”是一种在存取和处理数据结构和字符串方面非常方便的语言，而这种操作占了操作系统相当大的部分。作为一种面向终端的要求简明而紧凑表达式的语言，“C”使用一种大字符集，并且使很多符号，例如“*”、“&”等得到很频繁的使用。在这方面“C”语言可与APL相比。

“C”语言的很多特征令人想到PL/1，但“C”在结构化程序设计方面提供的设施比PL/1要强得多。

3.1　某些选出的例子

下面的例子都直接选自UNIX源代码。

3.2　例1

最简单的过程并不做任何实际工作，在源代码中这种过程出现了两次，它们是“nullsys”

(2864)和“ nulldev ”(6577)。

```
6577 nulldev ()
     {
     }
```

其中并没有任何参数，但是一对园括号“ (”和“)仍旧是需要的。一对花括号“ { ”和“ } ”界定了过程体，在本例中过程体为空。

3.3 例2

下面一个例子内容稍多一些：

```
6566 nodev ()
     {
       u.u_error = ENODEV;
     }
```

其中增加了一条赋值语句。该语句用一分号终止，它是语句的组成部分，而非 Algol类语言中的语句分隔符。

“ ENODEV ”是一个定义的符号，在实际编译之前由编译程序的预处理器将这种符号代换成一个与其相结合的字符串。 ENODEV在0484行被定义为19。UNIX与此相关的惯例是：定义符号由大写字符组成，而其他符号则使用小写字符。

“ = ”是赋值算符，“ u.u_error ”是结构“ u ”的一个元素(参见0419行)。注意，“ . ”算符用于从一个结构中选出一个元素。该元素名是“ u_error ”。在UNIX中，结构元素命名通常都用这种方式 —— 开头是一个区分符，然后是一个下划符，最后是一个名字。

3.4 例3

```
6585 bcopy (from, to, count)
     int *from, *to;
     {
       register *a, *b, c;
       a = from;
       b = to;
       c = count;
       do
         *b++ = *a++;
       while (--c);
     }
```

此过程的功能非常简单：它将指定的字数从一组连续单元

此过程有3个参数。第2行

```
int *from,*to;
```

说明前两个变量是指向整型的指针。因为对第三个参数没有

三个局部变量a、b和c被指定为寄存器型。寄存器易于

a和b是指向整型的指针，而c是一个整型。寄存器型说明也

```
register int *a,*b,c;
```

这强调了寄存器型与整型的关连。

对以“ do ”开始的3行语句应当仔细观察。如果“ b ”是指向整型的指针，那么：

```
*b
```

表示被指向的整型数。于是，为了将“ a ”所指向的值复制到b所指向的单元，我们可使用下面的语句：

```
*b=*a;
```

如果我们将此写成：

```
b=a;
```

那么这将使b的值与a值相同，也就是“ b ”和“ a ”将指向同一位置。至少在这里，这并不是所要求的。

从源到目的复制了第一个字后，需要使“ b ”和“ a ”的值增1，以分别指向下一个字。为此可写成

```
b=b+1;a=a+1;
```

但是“ C ”提供了一种简短的表示法(在变量名较长时，这种方法更加有用。)：

```
b++;a++;
```

或者

```
++b;++a;
```

在这里“ b++ ”和“ ++b ”并无区别。

但是，“ b++ ”和“ ++b ”可用作表达式中的一部分，在此种情况下，它们的作用就会不同。在两种情况下，对b增1的作用是相同的，但对于b++而言，进入表达式的值是其初始值，

后值。

‘ ＋＋ ”相同的规则。于是，“ －－C ”使进入表达式的值

用的，在UNIX源代码中到处都用到这两个算符。注意，在

这两个算符也可用于指向结构的指针。如果一个指针说
对其实施增量操作，该指针的实际值增加结构的长度。
用：

简单单完成了这一切。 do_while语句的结尾是：

先求值，然后被测试 (测试的是减1后的值)。若其值非0，

，那么此循环决不会正常终止。如果可能产生此种情况，

3.5 例4

```
4613 getf (f)
     {
       register *fp, rf;
       rf = f;
       if (rf < 0 || rf >= NOFILE)
         goto bad;
       fp = u.u_ofile[rf];
       if (fp != NULL)
         return (fp);
     bad:
       u.u_error = EBADF;
       return (NULL);
     }
```

参数“f”按系统默认为整型，并直接复制到寄存器型变量“ rf”中。(这种模式将得到普遍应用，因此读者会对此熟悉起来，我们在此也就无需再多作说明。)

下面三个是简单关系表达式：

```
rf<0 rf>=NOFILE fp!=NULL
```

若它们为真，则取值1，若为假，则取值0。第一个表达式测试“ rf”的值是否小于0；第二个表达式测试“ rf”值是否大于或等于NOFILE”的定义值，第三个表达式测试“ fp”的值是否不等于“ NULL”(“ NULL”定义为0)。

由“if”语句测试的条件是包含在园括号中的算术表达式。如若此表达式大于 0，则该测试成功，然后执行后面的语句。于是，若“ fp”的值是001375，那么

```
fp!=NULL
```

为真，作为算术表达式中的一项，其值取为1。因为此值大

```
return(fp);
```

从而终止“ getf”的执行，并将“ fp”的值作为“ getf”的

表达式

```
rf<0‖rf>=NOFILE
```

是两个简单关系表达式的逻辑或。

“goto”语句及相关标号的一个实例将在后面说明。

u-ofile是嵌入在结构u中的一个整型数组，其第rf个元

过程getf将一个值返回给调用过程。该值或者是“fp”值

3.6 例5

```
3113 wakeup (chan)
     {
       register struct proc *p;
       register c, i;
       c = chan;
       p = &proc[0];
       i = NPROC;
       do {
         if (p->p_wchan == c) {
           setrun (p);
```

```
    }
    p++;
  } while (--i);
}
```

本例与上一个例子有许多相似之处，但是本例中有一个新概念——一个结构数组。本例中使人感到有些困惑的是数组和结构都被称为“ proc ”(是的，“ C ”语言允许这样做)。该结构和数组说明为下列形式：

```
#150 struct proc
     {
       char p_stat;
       ..........
       int p_wchan;
       ..........
     } proc [NPROC];
```

“ p ”是指向一结构的指针型的寄存器变量，该结构的类型为“ proc ”。

```
p=&proc〔0〕;
```

将proc数组第一个元素的地址赋予“ p ”。其中的算符“ & ”在这里的意义是“ 的地址 ”。

注意，若一数组有几个元素，则这些元素的下标号为 0、1、…(n-1)。“ C ”语言允许将上面的语句写成如下更简单的形式：

```
p=proc;
```

在“ do ”和“ while ”之间有两条语句。其中，第一条语句可改写成下列较简单的形式：

```
if(p－＞p_wchan==C)setrun(p);
```

亦即，省略了原来的花括号。“ C ”是一种自由格式语言，行之间的文本安排并不重要。

作为参数传送给 setrun。(所有参数都传送值。)

的元素“ p_wchan ”的值是否相等。注意，如果将此写成下

语句是 p++;它将“ p ”的值增加“ proc ”结构的长度。(编

需知道“ p ”所指向的结构种类。当不需要作这种考虑时，
简单地说明为

程序则并不坚决要求这样做。
写成下列形式，但其效率会差一些：

```
……
i=0;
do
  if(proc〔i〕.p_wchan==c)
    setrun(&proc〔i〕);
while(++i＜NPROC);
```

3.7 例6

```
5336 geterror (abp)
     struct buf *abp;
     {
       register struct buf *bp;
       bp = abp;
       if (bp->b_flags&B_ERROR)
         if((u.u_error=bp->b_error)==0)
           u.u_error = EIO;
     }
```

此过程检查是否已出错，若出错指示器“ u.u_error ”没有设置，则将其设置为通用出错指示(“ EIO ”)。

“ B_ERROR ”的值为04(请查看4575行)，所以只设置31位，可将其用作屏蔽字以提取第 2 位的值。用在下列表达式

```
bp－＞b_flags&B_ERROR
```

中的算符“ & ”对两个算术值实施按位逻辑与。

如果“ bp ”指向的“ buf ”结构的元素“ b_flags ”的第 [illegible] 于1。

如果已有一个错，则表达式

```
(u.u_error=bp－＞b_error)
```

被求值，然后与0相比较。在此表达式中包括了一赋值符‘ [illegible] 予“ u.u_error ”后，该表达式的值是u.u_error的值。

将赋值用作表达式的一部分是非常有用的，而且是常用

3.8 例7

```
3420 stime ()
     {
       if (suser()) {
         time[0] = u.u_ar0[R0];
         time[1] = u.u_ar0[R1];
         wakeup (tout);
       }
     }
```

在本例中，应当注意过程“ suser ”返回一值，然后“ i [illegible] 为真，则执行在“｛”和“｝”中的三条语句。

注意，在不带参数调用一过程时，仍旧应当写一对空圆括号，例如：

```
suser( )
```

3.9 例8

“C”提供了一条件表达式。若“a”和“b”是整型变量，那么：

```
(a>b?a:b)
```

是一个表达式，其值是a和b中的较大者。

但是如果“a”和“b”被视为不带符号整型数，则此表达式并不能正常工作。为此可使用下一过程

```
6326 max (a, b)
     char *a, *b;
     {
       if (a > b)
         return(a);
       return (b);
     }
```

其中的窍门是将“ a”和“ b”说明为指向字符的指针，在进行比较时起无符号整型的作用。

该过程体也可写成如下形式：

```
{
  if (a > b)
    return (a);
  else
    return (b);
}
```

得“ else”在这里并无作用。

不同的、看来有些陌生的表达式。第一个是：

```
7679 gchar()
     {
       return (*u.u_dirp++ & 0377);
     }
```

u.u_dirp++”的值是一字符。(指针增量是副作用。)

寸，将进行符号位扩展。将其与 0377作按位与，则额外的

1。

]如0377)都被解释为八进制整型数。

```
1771 cmag(n)
     {
       return ((n+127)>>7);
     }
```

其返回值是：“n除以128，然后取整为下一个整数”。

注意，使用右移操作“>>”优于除操作“/”。

3.11 例10

上面说明过的很多要点集合在下一例子中：

```
2134 setrun (p)
     {
       register struct proc *rp;
       rp = p;
       rp->p_wchan = 0;
       rp->p_stat = SRUN;
       if (rp->p_pri < curpri)
         runrun++;
       if (runout != 0 &&
           (rp->p_flag&SLOAD) == 0) {
         runout = 0;
         wakeup (&runout);
       }
     }
```

请思考这一过程做了些什么以检查你对“C”的理解程度。

关于此过程可能有两个附加的特征需要了解：

- “&&”是用于关系表达式的逻辑与。(“前面引进的“‖”则是关系表达式中的逻辑或。)
- 最后一条语句包含表达式

```
&runout
```

在语法上这是一个地址变量，但在语义上只是一个独特的位
要使用的方法实例。程序员由于某种特殊的目的需要一独
足了要求，其精确值则并无意义。对此问题的一个解决方
址。

3.12 例11

```
4056 bawrite (bp)
     struct buf *bp;
     {
       register struct buf *rbp;
       rbp = bp;
       rbp->b_flags |= B_ASYNC;
       bwrite (rbp);
     }
```

最后第2条语句值得引起注意，它原本可以写成如下形式：

```
rbp－>b_flags=rbp  －>b_flag |B_ASYNC;
```

在此语句中，位屏蔽“ B_ASYNC ”与“ rbp－＞b_flags ”相“ 或 ”。符号“ | ”是算术值的逻辑或。

这是一个在UNIX中非常有用的结构的实例，这种结构可以节省程序员的很多劳动。如果■是任一二进制算符，则：

```
x=x■a;
```

(其中，“ a ”是一表达式)可被改写成更加简洁的如下形式：

```
x=■a;
```

使用这种结构的程序员应当注意空白字符的位置。因为

```
x=+ 1;
```

不同于

```
x= +1;
```

请考虑下列表达式的意义：

```
x=+1;?
```

此表达式具岐义性，其结果与编译的具体实现有关。

3.13 例12

```
but it wouldn't fit on the line.  As
noted earlier, the use of "proc" as an
alternative to the expression
"&proc[0]" is acceptable in this con-
text.


This kind of "for" statement is almost
a cliche in UNIX so you had better
learn to recognize it.  Read it as

     for p = each process in turn
```

一个非常通用的语法结构，功能强，表达紧凑。

1》中有清晰的描述，在此不再复述。

```
IOFILE-1 do
```

力来自于它给予了程序员很大的自由，可以选择在花括号

```
$949 signal (tp, sig)
{
    register struct proc *p;
    for (p=proc;p<&proc[NPROC];p++)
      if (p->p_ttyp == tp)
        psignal (p,sig);
}
```

在本例的“for”语句中，指针变量“p”逐个指向“proc”数组中的每一个元素。

实际上，原先的代码是：

```
for(p=&proc [0];p<& proc〔NPROC〕;p++)
```

但是这超过了本书UNIX源代码页中的行长。正如前面所指出的，用“proc”代替表达式“&proc〔0〕”在此上下文中是可以接受的，因此将原先的代码改写成现今的形式。

这种“for”语句在UNIX中几乎随处可见，所以应当仔细学习它。将其读成

```
for p=each process in tu断言(指向每一进程)
```

注意，“& proc〔NPROC〕”是“proc”数组第NPROC＋1个元素的地址(当然该元素实际并不存在)，也就是说，这是该数组所占用存储区后的第一个单元的地址。

我们再次指出：在前一个例子中

```
i++
```

表示“向整型数i加1”，而在本例中

```
p++
```

表示“将p移动指向下一个结构”。

3.15 例14

```
0470 lpwrite ()
     {
       register int c;
       while ((c=cpass()) >= 0)
         lpcanon(c);
     }
```

这是一个“while”语句的例子，应当将此语句与前面
比较(请对照Pascal的“while”和“repeat”语句)。

该过程的意义是：

当“cpass”的返回结束为正时，不断调用它，并将该
注意，在变量“C”的说明语句中使用了“int”，虽然

3.16 例15

下面的例子是从原先的代码中简化出来的：

```
5B01 seek ()
     {
       int a[2];
       register *fp, t;
       fp = getf (u.u_ar0[R0]);
       ..........
       t = u.u_arg[1];
       ..........
       switch (t) {

       case 1:
       case 4:
         a[0] += fp->f_offset[0];
         dpadd (a, fp->f_offset[1]);
         break;
```

```
    default:
      n[0] =+ fp->f_inode->i_size0
        &0377;
      dpadd(n,fp->f_inode->i_size1);

    case 0:
    case 3:
      ;
    }
    ............
}
```

注意对两个字的数组“ n ”的说明方式以及“ getf ”的使用(它出现在例4中)。

依赖于括号中表达式的值，“ switch ”语句作多路转移选择。每个可转移部分都有“ case 标号 ”。

如果“ t ”的值是1或4，则顺序执行一组操作。

如果“ t ”的值是D或3，则不执行任何操作。

如果“ t ”的值是其他，则执行标号“ default ”后的一组操作。

注意，用“ break ”跳出“ switch ”语句，从而执行紧跟switch语句的下一条语句。不使用“ break ”，则按“ switch ”语句中的正常执行序列执行。

于是，在“ default ”操作的尾端通常要使用“ break ”。因为“ default ”后的几个case ”实际并不执行任何操作，所以在这里省略“ break ”是安全的。

这两组操作实施一个 32位整数与另一个的加法。“ C ”编译的稍后版本将支持“ long ”(“ 长 ”)类型变量，这将使这种代码更易于编写和阅读。

```
)

6672 closei (ip, rw)
     int *ip;
     {
       register *rip;
       register dev, maj;
       rip = ip;
       dev = rip->i_addr[0];
       maj = rip->i_addr[0].d_major;
       switch (rip->i_mode&IFMT) {

       case IFCHR:
         (*cdevsw[maj].d_close)(dev,rw);
         break;
         case IFBLK:
           (*bdevsw[maj].d_close)(dev,rw);
         }
         iput (rip);
       }
```

在本例中有一些有趣的特征。

对“d_major”的说明是：

```
struct {
  char d_minor;
  char d_major;
}
```

这使得赋予“maj”的值是赋予“dev”值的高字节。

在本例中，“switch”语句只有两个非空case，没有“default”。两个非空case中的操作，例如

```
(*bdevsw〔maj〕.d_close)(dev,rw);
```

初看起来很难理解。

首先应当注意到这是一个过程调用，参数为“dev”和“rw”。

其次，“bdevsw(以及“cdevsw”)是结构数组，其“d_close”元素是一指向函数的指针，亦即：

```
bdevsw〔maj〕
```

是结构名，而

```
bdevsw〔maj〕.d_close
```

是该结构的一个元素，它是一个指向函数的指针，于是：

```
*bedsw〔maj〕.d_close
```

是一函数的名字。第一对圆括号是“语法上的糖衣”，它帮助编译程序正确了解它所处理项的位置。

3.18 例17

我们以本例结束本章

```
main() psig ()
    {
      register n, p;
      ..........
      switch (n) {

      case SIGQIT:
      case SIGILN:
      case SIGTRC:
      case SIGIOT:
      case SIGEMT:
      case SIGFPT:
      case SIGBUS:
      case SIGSEG:
      case SIGSYS:
          u.u_arg[0] = 0;
          if (core())
            n += 0200;
      }
      u.u_arg[0]=(u.u_arg[00]<<8) | n;
      exit ();
    }
```

其中，“ switch ”依据“ n ”的值进行转移位置选择，然后执行一组操作。

当然也可以用很长的“ if ”语句来代换“ switch ”语句，例如：

```
if(n==SIGQIT ‖ n==SIGINS ‖ …
        … ‖ n==SIGSYS)
```

这既不清晰，效率也不高。

注意一个八进制数与“ n ”相加的方法，以及用两个8位值如何构成一16位值。