联想计算机丛书之一

C程序员最常见错误分析

林雪柏　王江编译

北京联想计算机集团公司

一九九〇年十一月

C程序员最常见错误分析

林雪柏　王江　编译

北京联想计算机集团公司
一九九零年十一月

译 序

本书深入细致地分析了C语言中各种易犯的错误，书中含有大量的实例，每一个例子说明一种常见的错误，其中的许多错误即使是经验丰富的专家也会感到非常棘手。凡是编过程序的人都一定能够体会到程序的调试、排错是最费时间、最伤脑筋的，而本书的目的正是帮助读者减小犯错误的可能性，提高程序排错的能力，使读者在实践中能够节省大量的调试时间，并且使程序的可靠性得到显著的提高。

本书的作者 **Andrew Koenig** 是 **AT&T** 公司贝尔实验室软件技术中心的成员，他是一位具有 **20** 多年编程经验的专家，在程序设计方法方面曾出版过多本专著。本书浓缩了作者十多年来用C进行程序设计的经验，书中对于如何避免差一错误(最常犯的错误之一)、怎样理解指针与数组间的微妙关系、如何理解和构造函数说明等问题进行了透彻的论述，并且把C语言中各种常见的错误划分为词法上的错误、语法上的错误、语义上的错误、由连接器所引起的错误、由库函数所引起的错误、以及由预处理器所引起的错误和可移植性方面的错误等七大类型，它们涉及到了C语言的所有方面，是非常齐全的。在有关C语言的众多书籍中，能够如此详尽地论述各种常犯的错误的，到目前为止还只有这一本。本书的初稿在作为贝尔实验室的内部论文发表时就受到了读者的热烈欢迎，人们纷纷向贝尔实验室图书馆索取那份论文的拷贝。同时本书短小精悍、内容丰富。有许多材料在其它的C语言书籍中是很难找到的。因此，本书对于所有使用C语言的人来说，都是一本极为难得的理想读物。

由于本书的这许多优点，我们特地向广大的C语言用户推荐此书。原文中个别错误的地方和印刷上的错误，在译文中已经改正过来。但由于时间较为仓促，加上我们的水平有限，不足之处在所难免，敬请广大读者指正。

译者

一九九零年十月

目　录

引 言

我的第一个计算机程序是在 1966 年用 Fortran 写的。当时我想计算并打印出直到 10,000 的 Fibonacci 数：即序列 1，1，2，3，5，8，13，21，...的元素，其中从第二个数起，每一个数都是它前面的二个数之和。当然，它是不能完成任务的：

```
      I = 0
      J = 0
      K = 1
1     PRINT 10,K
      I = J
      J = K
      K = I+J
      IF( K - 10000 ) 1,1,2
2     CALL EXIT
10    FORMAT( I10 )
```

Fortran 程序员将很容易地发现，此程序漏掉了一个 END 语句。而当我增加了 END 语句之后，这个程序却仍然通不过编译，只产生了神秘的信息，ERROR 6。

经过仔细地阅读手册，最终发现了问题的所在:我所使用的 Fortran 编译程序不能够处理超过 4 位的整型常数。将 10000 改为 9999 就解决了这个问题。

我编写我的第一个 C 程序是在 1977 年。当然，它是不能完成任务的：

```
#include <stdio.h>
main()
{
        printf( "Hello world" );
}
```

这个程序在我第一次编译时就通过了。虽然它的结果有一点特别：在终端上的输出是这样的：

```
%cc   prog.c
%a.out
Hello world%
```

这里%字符是系统的提示符，它是系统用来告诉我该输入了的字符串。%紧跟在 Hello World 信息的后面，是因为我忘了告诉系统在它之后开始一个新行。在第 3.10 节里将讨论在这个程序中的一个更微妙的错误。

在这二种类型的问题间，有着一种实质上的区别。在 Fortran 的例子中包含有二个错误，但它的实现足以把它们指出来。而后面的这个 C 程序却是在技术上是正确的——从机器的观点来看，它没有错误。因此这时就不会有诊断信息。机器精确地执行了我让它去做的事情；它只是没有按照我在脑子里所想的那样去做。

本书将致力于第二种类型的问题。此外，它还将致力于那些在 C 中所特有的容易出错

的地方。例如，考虑下面的这个程序片断，它是用来初始化一个具有 N 个元素的整型数组的：

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

在许多 C 的实现上，这个程序都将会陷入无限循环!在第 3.6 节里对其原因进行了说明。

程序设计的错误代表了一个程序与在程序员头脑中该程序的模型之间出现了偏差。由于这些错误的出现是非常自然的，因此就很难对它们进行分类。我试图按照它们与考查一个程序的不同方法之间的联系来对它们进行了分组。

从一个较低的层次上看，一个程序就是一个由“符号”，或“标记”所构成的序列，就象一本书是一个由单词所构成的序列一样。将一个程序分离成符号的过程叫做“词法分析”。在第一章中考查了那些由于 C 的词法分析器的工作方式而引起的问题。

还可以将构成一个程序的那些标记看成是一个由语句、说明所构成的序列，就象可以把一本书看成是由句子所构成的一个序列一样。在这两种情况里，程序或书的含义都是由这些标记或单词是怎样组成一个大一点的单元的细节所决定的。在第二章里处理了那些由于误解了这些“语法”上的细节而可能会出现的错误。

在第三章中讨论了那些在意义方面的错误概念：一个程序员打算说某件事情，而实际上却说了另一件别的事情。在这里，我们假设您已很好地理解了 C 语言的词法的和语法的各种细节，因而将把注意力集中在“语义”的细节上。

第四章指出了一个 C 程序通常是由几个部分所组成的，各个部分分别进行编译，然后再把它们结合到一起。这个过程叫做“连接”，它是程序与它的环境之间的一部分联系。

在这个环境中，还包括某些“库程序”的集合。虽然，严格地讲，库程序并不是语言的一部分，但是，对于每一个要想完成一件有用的事情的 C 程序来说，它都是很基本的。特别地，有几个库程序是被几乎每一个 C 程序所采用的．同时又存在足够多的途径使您可能错误地使用它们，因此在第五章里的讨论将会是很有价值的。

第六章将说明我们所写出来的程序并不是实际上我们所运行的程序，预处理器首先要对它进行处理。虽然各种预处理器在实现上会有些不同，但是我们还是能够指出与多数实现所共有的一些有用东西。

第七章讨论可移植性问题——即一个程序可能在一种实现上能够运行，而在另一种实现上就不能运行的原因。要想正确地完成哪怕象整数算术运算那么简单的任务，其困难的程序也是惊人的。

第八章给出了有关保护性的程序设计的一些劝告，以及在其它各章里的练习题的答案。

最后，在附录里讨论了三个常用的、但却经常被误解的库函数。

练习题 0-1. 您会去购买一辆由一个具有很高的声誉的公司所制造的汽车吗？如果他们告诉您他们已经失去了声誉，情况会有变化吗?对于您来说，为您的用户找出错误的真正代价将会是什么？

练习题 0-2. 每隔 10 英尺放一根篱笆柱子，为了围起一条 100 英尺长的篱笆，需要多少根柱子？

练习题 0-3. 您是否曾做饭时用刀切伤过自己的手？菜刀是否可以做得更安全些？您是否愿意去使用一把经过这样修改的菜刀？

第一章　易犯的词法错误

当我们阅读一个句子时，我们通常都不会去考虑构成此句子的每个单词中的单个字母的含义。事实上，字母本身的含义是很少的：我们把它们组合成单词，并将意义赋给所组成的那些单词。

对于用 C 或其它语言所写的程序来说也是如此。程序里单个的字符孤立地看，并没有任何意义，只有在上下文中才会有意义。因此，在

```
P -> S = " -> "
```

里，一字符的二次出现表示的是二件不同的事情。更精确地说，每一个-的出现都是一个不同的“标记”的一个部分：第一个-的出现是->的一部分，而第二个-的出现则是一个字符串的一部分。再者，标记->所具有的意义与构成它的那二个字符的意义相差很远。

单词“标记”(token)指的是程序的一个部分。它扮演着与句子里的单词相同的角色：在某种意义上，它的每次出现都表示着同一件事情。一个相同的字符序列可能在一个上下文中属于一个标记，而在另一个上下文中又属于另一个完全不同的标记。在编译程序中，将程序划分为标记的那个部分，通常被称为一个“词法分析器”。

再看另一个例子，考虑语句：

```
if( x > big ) big = x;
```

在这个语句中的第一个标记是 if，它是一个关键字。下一个标记是左括号，接下来标识符 x，“大于”符号。标识符 big，如此类推下去。在 C 里，在标记之间，我们总是可以插入多余的空白(空格、制表、或换行)，因此，我们可以写成：

```
if
(
x
>
big
)
big
=
x
;
```

本章将探讨在标记的意义、以及在标记和构成它的字符之间的关系这二个方面上的一些常见的错误的概念。

§1.1　=不是==

大多数从 Algol 演变过来的程序设计语言，如 Pascal 与 Ada，都是使用:=来作为赋值符号，用=来作为比较运算符。而 C 则是使用=来作为赋值符号，用==作为比较运算符。这是很方便的：赋值运算比比较运算出现得更为频繁，因此，短一点的符号书写的次数就会更多。再者，C 将赋值处理成一个运算符，因此可以方便地写出多重赋值(例如

a=b=c)，并且赋值可以嵌入到一个大一点的表达式中。

这个方便引出了一个潜在的问题：在某人想要做一个比较的地方，他却不小心地写成了一个赋值。因此，下面的这条语句，它看起来将在 x 等于 y 时执行一个 break:

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

但实际上它却把 x 置为 y 的值，然后再检查此值是否为非零。我们再考虑下面的循环，它想要跳过一个文件里的空格、制表、与换行:

```
while ( c = ' '|| c == '\t' || c == '\n' )
      c = getc(f);
```

这个循环在与' '的比较中错误地使用了=，而不是用==。由于=运算符的优先级低于||运算符，因此，这个“比较”将把整个表达式

```
' ' || c == '\t' || c == '\n'
```

的值赋给 c。' '的值是非零，因此此表达式的值将为 1，而与 c(先前)的值无关。因此该循环将会吃掉整个文件。在这之后它将做什么就取决于所用的特定的实现是否允许一个程序在到达文件结束时，仍保持继续读入。如果允许的话，则此循环将会永远运行下去。

某些 C 的编译程序试图通过对形式为 e1=e2 的条件给出警告信息的办法来帮助它们的用户。当要把一个值赋给一个变量，然后再检查此值是否为 0 时，对于这样的编译程序，为了避免产生警告信息，可以考虑把此比较显式地写出来。换句话说，就是不要写成:

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

而应写成:

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

这将有助于把您的意图表示得更为明确。在第 2.2 节里，我们将讨论为什么在 x = y 外的括号是必不可少的。

从另一个方面把事情搞混也是有可能的:

```
if ((filedesc == open(argv[i],0))<0)
   error( );
```

在本例中的 open 函数在它发现一个错误时返回-1，而在它成功地打开了指定的文件时返回零或一个正数。这个程序片段想要把 open 函数返回结果保存在 filedesc 中，并用时检查 open 的执行是否成功。然而，第一个==应为=。按照所写的那样，它将对 filedesc 与 open 的返回结果进行比较，然后再检查此比较的结果是否为负数。当然，它是永远不会为负数的：==的结果总是 0 或 1 而决不会是负数。因此 error 函数就不会被调用。所有的东西看起来都很正常，但是 filedesc 的值将仍为它以前的值，而与 open 的返回结果没有任何关系。某些编译程序可能会警告您与 0 的比较是没有作用的，但是，您不应该指望总能得到这种警告。

§1.2 &和|不是&&或||

对于==运算符，是会很容易地漏掉一个=的，因为有许多的其它语言都使用=来作为

比较运算符。同样也很容易搞错&与&&，或| 与||，特别是由于C中的&与|运算符与它们在其它语言中的对应符号是不同的。第3.8节将讨论这些运算符的精确含义。

§1.3 贪婪的词法分析

某些C的标记的长度只有一个字符，例如/，*，与=。其余的C的标记的长度则为几个字符，例如/*，==，以及标识符。当一个C的编译程序碰到一个后面跟着一个*的/时，它必须能够确定是把这二个字符当作二个分开的标记呢，还是把它们作为一个单独的标记来处理。C利用一个简单的规则来解决这个问题："重复地吃掉最大的可能块"。即，将一个C程序转换成标记的方法是：从左向右扫描程序，每次都取最长的可能标记。这种策略有时也叫做"贪婪的"，或更通俗地讲，叫做"最大量地吃进"策略。Kernighan与Rithie曾把这种策略说成是："如果在输入流中，直到一个给定的字符为止，都已被分析成标记了，那么，下一个标记将包括可能构成一个标记的最长的那个字符串。"标记(字符串或字符常数除外)永远不会含有嵌入其中的空白空间(空格、制表、或换行)。

因此，例如，==是一个单一的标记，而= =则是二个标记，表达式

```
a---b
```

的含义与

```
a-- -
```

的含义相同，而不是

```
a- --b
```

类似地，如果一个/是某个标记的第一个字符，并且/后立即跟着一个*，那么这二个字符就开始了一段注释，而不管上下文是什么。

下面的这条语句，看起来象是将要把y置成x的值被由P所指向的那个值相除之后所得的结果：

```
y = x/*P          /* P指向除数 */;
```

实际上，/*开始了一段注释，因此编译程序将简单地尽快吃进程序的正文，直到*/出现时为止。换句话说，这条语句仅仅把y置成了x的值，而根本没有去检查P。把这条语句重写成：

```
y = x / *P      /* P指向除数 */;
```

或甚至是：

```
y = x/(*P)          /* P指向除数*/;
```

将会使得它去执行在注释里所指出的除法。

这种类型的近二义性(near-ambiguity)在其它上下文里可能会引起麻烦。例如，在一段时期里，C曾使用=+来表示现在记为+=的运算符所代表的意义。某些C的编译程序仍然还接受这种已经废弃了的用法；这样的一个编译程序将会把

```
a=-1;
```

处理成代表

```
a =- 1;
```

的意义，它的含义与

```
a = a - 1;
```

的含义相同。

这将会令一个想要表示：

```
a = -1;
```

的程序员大吃一惊。

这种过时的编译程序还将把:

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

处理成:

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

即使/*看起来象是一个注释。

这样的老式编译程序还将把复合的赋值处理成二个标记。这样的一个编译程序将把:

```
a >> = 1;
```

看成是没有问题的，而一个严格的 ANSI C 的编译程序将会拒绝接受这样的写法。

§1.4 整型常数

如果一个整型常数的第一个字符是数字 0，那么该常数就会被当成 8 进制的。因此 10 与 010 所表示的是不同的二个数。再有，许多 C 的编译程序都会接受 8 与 9 作为“8 进制的”数字，而不会给出任何错误信息。这种不寻常的结构的意义是从 8 进制数的定义推导出来的。例如，0195 意味着 $1\times8^2+9\times8^1+5\times8^0$，它等价于 141(十进制)或 0215(八进制)。很显然我们是反对这种用法的。ANSI C 则禁止这样使用。

注意找出在下面的上下文中的由于不小心而写出的八进制值:

```
struct{
    int part_number;
    char *description;
} parttab[ ] = {
046, "left-handed widget",
047, "right-handed widget",
125, "frammis"
};
```

§1.5 串与字符

单引号与双引号在 C 中所表示的是二件非常不同的事情，在某些上下文中，把它们搞混了所产生的结果将会令人感到意外，而不是产生错误信息。

一个括在单引号里的字符仅仅是某个整数的另一种写法，该整数在具体实现的排序序列中，与这个给定的字符相对应。因此，在一个 ASCII 的实现里，'a'与 97 所表示的东西是完全相同的。

在另一方面，一个括在双引号里的字符串，则是一个指针的一种速记方法。该指针指向一个无名数组的首字符，此无名数组被用在括号之间的那些字符，与一个其二进制值为零的附加字符一起来进行初始化。

因此，语句

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

等价于

```
char hello[ ] = { 'H', 'e', 'l', 'l', 'o', ' ',
```

```
    'w', 'o' , 'r', 'e', 'd', '\n', 0 }
printf( hello );
```

因为在单引号中的一个字符代表一个整数，而在双引号中的一个字符代表一个指针，所以，编译程序在做类型检查时，通常都能指出用错了二者的地方。因此，例如，若说明

```
char *slasy = '/';
```

则将会产生一条错误信息。因为'/'不是一个字符指针。然而，这些实现并不检查自变量的类型，特别是 printf 的自变量。因此，若说明

```
printf( '\n' );
```

而不是

```
print( "\n" );
```

则会在运行时刻产生一个意外的结果，而不是在编译诊断时给出错误信息。第 4.4 节中对其它情况进行了详细的讨论。

由于一个整数通常都足够大，以致于可以容纳得下几个字符，因此某些 C 编译程序允许在一个字符常数以及一个字符串常数里有多个字符。这意味着若写出'yes'而不是"yes"也将会顺利地通过编译而不会被发现。后者("yes")表示："四个连续的内存单元的首地址，它们分别包含有 y、e、s 以及一个空(null)字符" 而'yes'的含义则是没有精确定义，但许多 C 的实现都把它当成是表示："一个整数，它是由字符 y、e 与 s 的值通过某种办法进行复合而得到的。" 在这二个数量之间的任何相似性都完全是巧合的。

练习题 1-1. 某些 C 的编译程序允许使用嵌套的注释。写一个 C 程序，用它确定是否正在一个这样的编译程序上运行，而不产生任何错误信息。换句话说，就是这个程序应该在这二种关于注释的规则下均是正确的，但对于每一种规则，它所做的工作应该有所不同。提示：在一个括在引号里的字符串中的一个注释符号/*仅仅是这个字符串的一个部分；在一个注释里的一个双引号" "仅仅是此注释的一部分。

练习题 1-2. 如果您正在写一个 C 的编译程序，是否会让用户能使用嵌套的注释？如果您正在使用一个允许嵌套注释的 C 编译程序，会去使用这种特性吗？您对第二个问题的答案是否会影响到对第一个问题的回答？

练习题 1-3. 为什么 n-->0 所表示的是 n-- >0 而不是 n- ->0？

练习题 1-4. a+++++b 所表示的是什么意思？

第二章　易犯的语法错误

为了理解一个 C 程序，仅仅理解了构成它的那些标记是不够的。您还应该理解这些标记是怎样地结合在一起，以形成说明、表达式、语句、以及程序的。通常这些结合都是已经严格地定义了的。但是这些定义有时是违反人的直觉的，或者是容易使人搞混的。本章将考查某些语法结构，这些语法结构不是很容易就能正确地掌握的。

§2.1　理解函数说明

有一次我曾与某人进行过交谈，他正在写一个要单独在一台微处理器上运行的 C 程序。当这台机器被打开时，硬件将会去调用一个子程序，这个子程序的地址被保存在位置为零的内存单元里。

为了仿真打开电源的动作，我们必须设计出一条 C 语句来显式地调用这个子程序。在经过一段时间的思考之后，写出了下面这条语句:

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

象这样的表达式是会令 C 程序员在心里产生恐惧的。但是，他们是可以不用这样的表达式的，因为通常都可以利用唯一的一条简单规则的帮助来相当容易地构造出它们。这条规则是: 按照您所使用的方式来对它进行说明。

每一个 C 的变量说明都有二个部分: 一个类型和一个由被称为说明符的、与表达式类似的东西所组成的表。一个说明符看起来有点象一个表达式，我们期望对它进行求值的结果将会得到所给定的类型。最简单的说明符是一个变量:

```
float   f,g;
```

它指出了，对于表达式 f 与 g，在对它们进行求值时，将会得到类型 float。由于一个说明符看起来象一个表达式，因此可以自由地使用括号:

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

它表示((f))将被求值为一个 float 类型，因此通过推理，可以得出 f 也是一个 float 类型。

与此类似的推理也可以应用于函数与指针类型。例如，

```
float  ff( );
```

它表示表达式 ff( )是一个 float 类型。类似地，

```
float  *pf;
```

它表示*pf 是一个 float 类型，因此 pf 是一个指针，它指向一个 float 类型的数。

这些形式在说明里进行了结合，结合的方式与它们在表达式中所用的相同。因此:

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

所说的是，*g()与(*h)()都是 float 类型的表达式。因为()比*结合得更为紧密，所以*g()与*(g())所表示的都是同一个内容: g 是一个函数，它返回的是一个指向一个 float 类型的指针，而 h 则是一个指针，它指向一个函数，此函数返回的是一个 float 类型。

只要知道了怎样去说明一个给定类型的变量，我们就可以很容易地为此类型写出一个模子: 只需在说明中除去变量的名字和分号，然后再把剩下的全部内容用括号括起来即

可。因此，由于：

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

说明了 h 是一个指向一个返回一个 flaot 类型的函数的指针，所以：

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

就是适用于一个指向一个返回一个 float 类型的函数的指针的一个模子。

现在，我们可以分二个阶段来对表达式(*(void(*))0)()进行分析。

首先，假设我们有一个变量 fp，它包含了一个函数指针，并且想去调用由 fp 所指向的那个函数。这可以通过下面的办法来完成：

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

如果 fp 是一个指向一个函数的指针，那么*fp 就是该函数本身，因此(*fp)()就是引用这个函数的方法。ANSI C 允许将它简写成 fp()，但您应在脑子里记住，它只不过是一种简写而已。

在表达式(*fp)()中，在*fp 两边的括号是必需的，因为函数应用比一元运算符结合得更为紧密。如果没有这对括号，那么*fp()就与*(fp())的含义完全相同。ANSI C 将把它当成是*((*fp)())的简写。

现在，我们已将问题简化为去寻找一个合适的表达式来代替 fp。这个问题是分析的第二个部分。如果 C 能够读出我们关于类型的想法，那么我们能够写出：

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

但是它是不能完成任务的。因为*运算符一定要有一个指针来作为它的操作数。再有，此操作数必须是一个指向一个函数的指针，这样*运算的结果才能被调用。因此，我们需要把 0 放入一个类型的模子中去，这个类型可以被不严格地描述为："指向返回值为 void 的函数的指针"。

如果 fp 是一个指向返回值为 void 的函数指针，则(*fp)()是一个 void 值，并且它的说明将会象这个样子：

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

因此，我们可以写出：

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

这是以说明一个伪变量来作为代价的。但只要我们知道了怎样去说明变量，我们也就知道该怎样去生成此类型的一个常量：只需从变量说明中除去变量的名字即可。因此，我们通过说明：

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

就可以把 0 放入到一个"指向返回值为 void 的函数的指针"类型的模子中去。并且现在我们可以用(void (*)())0 去代替 fp 了：

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

最后的分号使得此表达式变成了一条语句。

在我处理这个问题的那个时候，还不存在象 typedef 说明这样的东西。虽然不利用 typedef 来完成这个例子是展示这些细节的一种好方法，但是 typedef 能使它变得更清楚：

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

这个令人不愉快的例子还具有许多种情况，它们都是C程序员经常会碰到的。例如，考虑signal库函数。在包含有此函数的C的实现中，它带有二个自变量：一个代表所捕获的特定的信号的整数代码，以及一个指向一个用户提供的函数的指针，此函数返回值为void，它用于处理此信号。第5.5节对此函数做了更详细的讨论。

通常，程序员并不会自己去说明signal函数。相反，他们会依赖于系统的头文件signal.h中的一个说明。此头文件是怎样来说明signal函数的呢?

从用户定义的信号处理函数开始考虑是最容易的，它可能会被定义成这样：

```
void
sigfunc(int n)
{
    /*  在此对信号进行处理  */
}
```

sigfunc的自变量是一个表示一个信号的编号的整数；从现在开始，我们将把它忽略掉。

上面的(假设的)函数体定义了sigfunc。为了对它进行说明，我们可以写出：

```
void  sigfunc(int);
```

现在，假设我们想要把sfp说明为一个指向sigfunc的变量。如果sfp指向sigfunc，则*sfp就必须表示sigfunc本身，因而*sfp就是可调用的。然后若sig是一个int类型，则(*sfp)(sig)就是一个void类型，因此，我们这样来说明sfp:

```
void  (* sfp) (int);
```

这表明了怎样去说明signal。因为signal返回一个与sfp相同类型的值，因此，我们应能这样来对它进行说明：

```
void (* signal(某件东西))(int);
```

这里的“某件东西”代表signal的自变量的类型，我们还应知道怎样把它写出来。阅读这个说明的一种方法是，把它看成它说明的是：用适当的参数来调用signal，间接引用所得到的结果，然后用一个int类型的参数来调用它，并给出一个void类型的结果。因此signal必须是这样的一个函数：它返回的是一个指向一void类型的函数指针。

至于signal本身的自变量又是什么呢?我们想要说明的是，signal接受二个参数：一个int类型的信号编号，以及一个指向用户定义的信号处理函数的指针。最初，我们是通过说明：

```
void (*sfp)(int);
```

来说明一个指向一个信号处理函数的指针的。sfp的类型是通过从它的说明中除去sfp的方法而得到的，这样所得到的结果是void(*)(int)。再有，signal函数返回的是一个指向对于此信号类型的先前的处理程序的指针；这个指针也是一个sfp。因此，我们可以通过说明：

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

来对signal函数进行说明。

同样，用typedef说明也可以对此说明进行简化：

```
typedef  void (* HANDLER)(int);
HANDLER signal(int,HANLER);
```

§2.2 运算符并不总是具有您想要的优先级

假设已定义了的常数 FLAG 是一个整数，并且在它的二进制表示中，有而且只有一位为 1(也即它为 2 的幂)，同时您想要去测试整型变量 flags 的那一位是否也为 1。为完成这项测试，常用的方法是写出:

```
if( flags & FLAG ) ...
```

它的含义对于大多数 C 程序员来说，都是很明白的: 一条 if 语句测试对括号中的表达式所得的结果是否为 0。为了能起一个说明性的作用，更明显地将这个测试表示出来也许会更好:

```
if( flags & FLAG != 0 )...
```

这条语句现在更容易理解了。但它还是错误的，因为!= 比&结合得更为紧密，因此现在的解释是:

```
if( flags & (FLAG != 0) )...
```

如果 FLAG 是 1，则它可以完成任务(碰巧地)，但对于其它情况，它都不能够完成任务。

假设您具有二个整型变量，hi 与 low，它们的值只能在 0 到 15 之间，而您想要去将一个整型量 r 设置成 8 位的值，它的低位部分是 low 的那些位，而它的高位部分则是 hi 的那些位。完成这项工作的一种自然的方法是写出:

```
r = hi<<4 + low;
```

不幸的是，这样写是错误的。加法比移位结合得更为紧密，因此，这个例子等价于:

```
r = hi << ( 4 + low );
```

这里，有二种方法可以使它变为是正确的。即:

```
r = (hi << 4 ) + low; 或
r = hi << 4 | low;
```

其中第二种方法指出了实质性的问题，它来自于对算术与逻辑运算所进行的混合;在移位与逻辑运算之间的相对优先级是更符合人的直觉的。

要想避免这些问题的一种办法是，对每样东西都加上括号，但是，具有过多括号的表达式又会显得很烦琐。因此，试着去记忆 C 中的优先级是很有用的。

不幸的是，它们有 15 种之多，因此要记熟它们并不总是很容易的。下面给出了一个完整的表。

运算符	结合性
( ) [ ] -> .	左
!~ ++ -- - (类型) * & sizeof	右
* / %	左
+ -	左
<< >>	左
< <= > >=	左
== !=	左