C语言编程员指南

吴益民 译

H

中国科学院希望高级电脑技术公司

C语言编程员指南

吴益民　　译

H

中国科学院希望高级电脑技术公司
一九九一年五月

前 言

此书的写作是出于我自己对于一个可用的、易理解的和可读的C语言参考书的需要。标准C经典，《C程序设计语言》，由Brian Keringhan和Dennis Ritchie合写的，包括40页详尽的参考手册，但它看来更直接地针对程序语言的理论工作者和编译器作者而不是针对程序员。其它关于C的很多书是自学教科书，它对初学者也许管用，但对要解决一个专门问题的人来说帮助却很小。

这种参考资料的缺乏部分是由于C和Unix操作系统的不断普及。许多计算机制造商为他们的硬件采用了UNIX作为操作系统因为它解除了开发他们自己的操作系统的负担而同时又能利用大量软件库。不幸的是，相同的这些制造商相信C的立即可用性也能减轻他们为他们的系统提供适当文档的责任——他们简单地让用户去看标准Unix书，虽然有时他们也在他们自己系统环境下出一些书。而有趣的是，由传统的大型机制造商提供的更原始的语言和专用的操作系统都配有很好的文档。

这本书不用一页页地读，它可放在程序员的案头以备查询，象一个作家在工作需要时查阅字典一样。我已经努力使每个条目完整和自包含，虽然有许多条目交叉引用。把这些相关的材料放在一起当然会有一定的重复。

许多例子已用Unix C编译器，Microsoft MS_DOS C编译器和CP/M 80 A2tec II C编译器测试过。所有例子都可在这三种实现上运行，除了书中特别指明的。

在写此书时。还没有一个国际公认的标准C，并且，K &-R所写的参考手册成为了普遍接受的标准。美国国家标准局提出了C标准的一个草案，称为X3J11，但是这个标准它在完成之前还要进行一些修改。而一个竞争的标准（X/OPEN）已经问世了。在任何情况下，任何标准的成功都依赖于支持它的厂商和广泛程度。看来在一段时间里C还会继续有多种细小差别版本的状态。

然而，C的K &R描述是几乎所有实现的基础并且是这里要描述的语言。当一个特性是某实现特有的，要明显地标明。

此书开始是一个简短的C个人指导，是为有经验的程序员自学时浏览C的需要而准备的。一个有其它模块化结构语言如Pascal或Algol工作经验的程序员会发现这个人指导，词典、好的C编译器和一定程度的坚持就足以获得C的工作知识。经验只能由实践而来。

附录提供了许多词典中提到的C特性的演示例子。程序员要浏览一下例子，主要不是为了实际应用，而是看看有经验的程序员所写的“好的C码”。

另外，调试段给程序员在困难、复杂和不愉快的任务中提供帮助，而不管程序员是新手还是有经验的。

目　录

第一章　C语言个人指导

这章是面向对高级语言有一定编程经验的程序员的个人指导。模块结构化语言方面的知识如PASCAL会很有用。

这个个人指导可和本书的词典结合起来读。指导仅仅介绍一下条目，而不是讲清它，通过查词典，你会得到一个详细的解释。只通过使用个人指导和词典，一个有经验的程序员就可以学会C语言并编程上机。

C语言产生于二十世纪七十年代初，作者是 Dennis Ritchie，美国电话电报公司贝尔实验室的程序员，他也参加了UNIX操作系统的设计。C的名字看来是B的自然后继，B是从CPLC(Basic Combined Programming Language)而来。

C是众多的语言中唯一不是由政府委员会和计算机公司开发的，它是程序员为自己使用而开发的。因此，它具有这些特征：简单且效力高，简明且富于表达，不太严格。

C语言的一个设计目标是减少输入一个程序时所需要的击键次数。

C中的语句数很少，有45个运算符，很多是在所有语言都有的基本操作上变化而来的。其它语言提供的基本语句、象输入输出、在C中没有，它们需由实现者把它作为外部库的一部分功能提供。然而，虽然它的语法很简朴，C语言却既适于复杂的高层任务如编译器、算术计算器的编写，也适于低层的任务，如位操作和设备驱动。事实上，95%的UNIX操作系统是用C写的。

虽然C在大范围的操作系统和计算机上实现了。它的流行多半是因为UNIX的发展。从微机到大型机都配有C语言，如IBM PC 上有两打以上的C编译器上市。用C编程保证简单地移植一个程序。因为可移植性中的很多恼人问题能通过标准函数库中的予处理来解决。

程序员喜欢C是因为他们发现它容易使用。它的语法不干扰算法的自然表达。然而，程序员既有做任何事的自由，也有留意他的程序的责任。“自由和易用”的另一面是“滥用的危险”——C语言允许但不原谅错误。

定义变量后，可以无视语言的限制、混用它的类型。在算术表达式中，不同类型的变量能自动转换来求得表达式的解。实参和正规参量的类型一致性是不检查的。

在UNIX和其它一些系统中有个程序Lint用来查错。不把这个任务交给编译器，是为了保证编译器小而快。

源文件：

一个C语言程序由字符文件组成。这些字符文件通过编译成为目标文件，这些目标文件依次链结并和库链结起来产生一个可执行的目标程序。

即使最简单的程序，编译和连接这两步也是不可避免的。因为标准函数库比原始的C指令集提供了更多的功能。程序员可以自由地把它们常用的函数加进库中或者创建一个新的库。

把一个程序分成几个小文件可以节省编译时间，因为没有修改的文件就不用再编译

了，而且小的文件也容易编辑。

所以，当一个程序太大时，可以分成两个或更多的文件。修改其中一个文件仅需要重新编译，其余的就不再用编译。连接器会把各个目标文件和库连接起来产生可执行程序。

编译时引用的文件用予处理文件语句#include引用。

标识符：

变量和函数标识符由英语的二十六个字母，允许大小写，和10个数字组成，标识符必须由字母开头。虽然一些编译器不允许下划线作为标识符开头，但它们也可用在标识符的其它地方。

小写字母和大写字母是不同的。例如，abc和aBC不一样。

标识符中前八个字符是有用的。若两个标识符前八个字符相同但第九个字符不同，编译器认为这两个标识符是一样的。

在C中，下面这种命名法很常见：

一般用小写字母，常量用大写字母，常量用予处理#define语句定义。类型定义时字母从i到n表示整型标识符，c表示字符型标识符，s表示串型标识符。

数据定义和说明：

变量必须在使用之前定义。下面是常见的数据类型：

类型	定义	类型长度
char	字符	8位
short int	短整数	16位
int	整数	和实现有关
long int	长整数	32位
float	浮点数	32位
double	双精度数	64位
unsigned char	无标志字符	8位
unsighed int	无标志整数	和实现有关

虽然这些长度是常用的，但程序员应该根据具体实现来决定。

其它数据类型有：

1、数组。

2、结构，不同类型的变量构成的组。

3、指针，一个含有另一个变量地址的变量。

4、联合，两个或两个以上的变量，它们占据同一块内存空间。

5、枚举，取值于有限集合元素的变量。

6、串，以NULL字符结束的字符数组。

基于以上这些类型，可用typedef语句定义新的数据类型。

数据定义和数据说明不同之处是它保存空间。

在函数外定义的或说明的数据从定义点起到源文件尾是可见的。在函数内定义的数据仅在函数内可见。在复合语句中定义的数据仅在复合语句中可见。

变量可有四种存贮方式，如下：

static　在程序开头，存贮空间静态保留。

auto　函数开始执行时分配空间，函数结束时释放空间。

extern　存贮空间在另一个目标文件中分配，通过连接器访问。

register　只要可能就存入高速寄存器。

常量：

正文中常量值以下列方式规定：

类型	语法	例子
字符（char）	单引号	'a'
串（string）	双引号	"abc"
整数（int）		123
八进制整数（octal jnt）	O	0377
十六进制整数（hex int）	Ox	0×2f
长整数（long int）	t或L	123L
浮点数（float）	小数或	3.24
	科学位数法或	3.2E－24
双精度数(double)	同浮点数	

所有的浮点数被看作双精度类型。

换码字符：

下列的固定的常量是在实现中规定的：

\n　换行

\t　制表符

\r　回车

\v　垂直制表

\b　退格

\″　双引号

\′　单引号

\\　反斜线

\nnn　八进制字符值

原始的反斜线作为换码字符。这些常量的使用使程序员从过细地熟悉计算机的字符集中解放出来。

运算符：

C中有45个运算符，分成15组。每组中所有算符有相同的优先级。组之间优先级排列，从高到低。结合性标在每组前。

1、左结合

（ ）	函数调用	getc(stdin)
〔 〕	数组元素引用	i〔17〕
→	指向结构成员指针	alt_ptr→c
.	结构成员引用	alt. c

2、右结合

−	负号	−i
++	增1	i++
−−	减1	i−−
!	逻辑非	! found
~	求反	~0×7f
*	间址	*c−ptr
&	求地址	&i
sizeof	变量或类型长度	sizeof(i)
(type)		(int)c

3、左结合

*	乘	i*j
/	除	i/j
%	求模	i%j

4、左结合

+	加	i+j
−	减	i−j

5、左结合

<<	位左移	i<<2
>>	位右移	i>>4

6、左结合

<	小于	i<j
<=	小于等于	i<=6
>	大于	i>2
>=	大于等于	i>=j

7、左结合

==	等于	if(i==5)
!=	不等于	if(i!=5)

8、左结合

&	逐位与	c&033

9、左结合

^	逐位异或	c^0317

10、左结合

|	逐位或	c|0333

11、左结合

&&	逻辑与	i==5&&j==6

12、左结合

||	逻辑或	i==5||j==6

13、右结合

? :	条件表达式	i>4? i:j

14、右结合

=	赋值	i=7
*=	乘后赋值	i*=3
/=	除后赋值	i/=4
%=	模后赋值	i%=4
+=	加后赋值	i+=2
-=	减后赋值	i-=3
&=	按位与后赋值	i &=0333
^=	按位异或后赋值	i^=0177
|=	按位或后赋值	i|=0177
<<=	位左移后赋值	i<<=2
>>=	位右移后赋值	i>>=3

15、左结合

,	求值并且丢弃	i=5,j

表达式：

一个表达式是下列之一：

1、一个变量名

2、一个函数调用

3、一个数组名

4、一个常量

5、一个函数名

6、对结构成员的一次引用

7、对一个数组成员的一次引用

8、以上几个用括号起通过适当的运算符连起来。

语句

后跟一个分号的表达式被认为是一个语句。例如：i+1 是表达式，而 i+1；是语句。

其它语句是：

1、 ；，空语句

2、if（表达式）语句

3、if（表达式）语句1 else 语句2

4、while（表达式）语句

5、do 语句 while（表达式）

6、for（表达式1；表达式2；表达式3；）语句

7、switch 整数表达式

```
{
case 常量1：语句
case 常量2：语句
:
}
```

8、break

9、continue

10、goto 标号

11、return

12、return 表达式

一个复合语句（或块）是用大括号括起来的一个语句。例如：

```
{
  i++;
  j++;
}
```

是一个由两个语句组成的复合语句，它可以作为一个语句使用。

数组和指针：

指针是含有其它变量地址的变量。C 中指针需要定义，它的类型是它所指向的类型的指针。

指针可以进行几种简单的算术操作。可以从一个指针上减去一个整数值也可给它加上一个整数值。但是象把两个指针相加，或把指针除以一个整数都是无意义的，也是不允许的。

在很多机上指针和整数没有什么区别，它们占有相同大小的内存。象整数那样操作指针是可能的。编译器也许会提出警告信息，也许不会，但结果常常是不可移植的。这样一个程序也许可在一号机器上正常运行但却不能在另一台机器上正常运行。

数组是有相同类型的一组变量，它们占据内存连续的一块空间。例如，定义：

```
int i[10];
```

定义一个十个整数的数组，从i[0]到i[9]。注意在C中，数组的下标是从0开始的。

定义：

```
int *p_i;
```

定义一个变量叫p_i，它作为指向整数的指针。

表达式*p_i意味着“p_i指向的变量”。在C中，数组名代表数组的首址，所以语句：

```
p_i=i;
```

置p_i指向数组i的第一个整数。

假设p-i指向i[0]所以p_i+1指向i[1]。指针加1是指指针指向下一个元素而不是增加一个字节。若指针指向字符数组，加1意味着“加一个字符变量的长”，若指针指向整数数组，则指针加1意味着“加一个整数变量的长”。

有两种方式指向数组元素，或者用数组引用（例如，i[4]），或者用合适的指针（例如，*(p_i+4)）。

函数：

在所有的源程序编译连接形成可执行的目标程序时，须有一个函数叫 main，可执行程序中的第一个可执行语句就是main中的第一个可执行语句。

C 中所有函数在同一个层上，不能在一个函数中定义另一个函数。甚至，所有源程序中的函数以及和源程序目标文件连接的库中的函数都是互相可见的。唯一的例外是函

数定义为static，这时函数只在定义它的源程序中可见。

调回函数时允许带参数，参数只能传值。C中，数组作为指针被传送，指针可被传送。在一些C版本中，结构可作为参数传送。

函数不能修改参数的值，除了通过引用传过来的数组和指针内容。对这个函数来说，只要参数的地址是可见的，就可以修改参数的值。

函数中定义的变量可以是auto,static和register。对在函数中定义的extern变量，不保存空间。函数内定义的变量对外界来说是不可见的。

函数可以用return语句返回值。若返回值的类型在函数头没有定义，就假设为int，若返回值不是int，则函数要在使用它之前在源文件中定义或说明。

函数名是指向函数的指针，它可作为参数传给别的函数。

函数体，即它的变量定义，说明和语句，用花括号括起来。

附录A是很多C版本中常用的函数的一张表。

附录B是一些串操作函数的源码。

附录C是一些用来给字符文件编密码和解密码的源码。

附录D是从多项式表达式构造一颗二叉树的函数源码。

予处理：

C编译的第一个方面就是予编译。予编译时，去掉注释。替换文本，有条件地引进或调出程序中的一部分。UNIX中，予编译可以单独运行。予编译在别的语言中也有运用的。

以#打头的行是予编译语句，它们是：

1、#define；进行参数替换。用作定义符号常数和参数代的宏。

2、#if常量表达式 语句 #endif；若表达式非零，在编译时把列#endif的语句调进来。

3、#if常量表达式语句#else 语句 #endif；

除了 #else 同上。

4、#ifdef 标识符 语句 #endif；假若这时标识符已定义，把列 #endif 的语句调进来。

5、#ifdef 标识符 语句 #else 语句 #endif；除了 #else 同上。

6、#ifndef 标识符 语句#endif，假若这时标识符没有定义，则把#endif之前的语句调进来。

7、#ifndef 标识符 语句#else 语句#endif；除了#else 同上。

8、#include“文件名”或#include<文件名>；在编译时调进文件。

9、#line n “文件”；

10、#undef 标识符。取消标识符的定义。

main 的参数：

在很多操作系统中，main 函数输入两个变量：argc 说明在命令行中调用程序时的参数个数，整数。argv，一个指向字符指针数组的指针，数组是指向参数的。

程序例子：

以下是一个有两个函数的例子，用来表示C程序的一段结构。

```
#define  CONSTANT1  5
#define  CONSTANT2  4
main(argc, argv)
int argc;
char *argv[ ];
{
  int  j=CONSTANT1;
  int  k=CONSTANT2;
  int  product;
  product=multiply(j, k);
}
multiply(m, n)
int    m, n;
{
  return m*n;
}
```

main函数中没有用到argc和argv参数。形式参数在左花括号前定义。函数自己的变量在花括号后定义。main有三个auto变量，j、k和product，它们都是整型。

multiply返回一个整型值，所以不同在main中说明。在main中的赋值语句调用multiply，带参数j和k并把multiply返回赋值给product。

注意在函数开头没有显示的关键字。

multiply在定义时带有形参m和n，和在main中调用时的实参j和k相对应。return语句计算m和n的乘积并回送。

第二章　调试

1945 年，格雷斯·玛蕾·霍普一个美国海军上校正为计算机 MarR Ⅱ 工作。她发现一个蛾子飞进机器的继电器中使它不能工作。她小心地拿掉了蛾子把它钉在工作日志上并且告诉她的上司她拿掉了虫子（“debugged”）。这本钉有蛾子的工作日志现在仍可在弗吉尼亚达尔根的海军地面武器中心博物馆看到。

人们可以从学校、书本或在正规的商业训练中学到编程。因为编程是定义好的、结构化的。调试却是非结构化的、无方向的、没有重点的、靠运气的。在学校里不教调试的规则和方法。一个新程序员只有靠自己的才智和同事的帮助才能学会。

早期计算机设计时，人们对生成软件的过程了解很少。很多程序员既是工程师也是数学家，他们认为他们原来领域的工作方法也适合于开发程序。后来，才发现编程和他们原领域的问题完全不同，有很多问题是和人类思维的复杂性紧紧交织在一起的。

第一代的软件工程师低估了编程的复杂性和他们的出错能力。硬件工程师也一样，他们认为计算机只需调试一次，就象调试汽车一样。但是他们忽视了关键的差别，就是汽车每次都是做同样的事而计算机常常是运行不同的程序。

那个时代一个五位的计算机正好有 32 个操作码。每种位的组合都是合法的指令。虽然有小错，计算机也能运算，当它终于停止时，程序计数器离出错的地方已经很远了。当有错时。这些早期计算机简单地停机，没有信息、编码或其它东西。程序员只能检查控制台的灯，查找出错指令的地址，拨一些开关，看看内存单元中是什么，然后，程序员只有靠自己了。没有汇编，没有高级语言，没有编译器，也没有操作系统。符号调试器更没有听说过也没有梦见过。

在这种环境下，程序员要检查每次运行的输出以发现错误。他用一种叫“书桌检查”的技术调试，意思是他人工地编辑和测试他的程序而不事实上运行它。他把程序在书桌上铺开，一条指令一条指令地小心查看，标出变量、寄存器的变化，同时查出逻辑错误。

那时的程序员不象今日一样利用计算机开发和测试程序。在那时典型的批处理站，他希望能一天编译一次他的程序。他向操作员提交一摞钻孔卡片，希望能在夜间编译运行。第二天上返回时，他要浏览一下清单，修正找到的错误，再提交作业。程序员要花大量时间做书桌检查，因为一个小错会耽搁几天工作。

交互环境的开发戏剧性地改变了编程的实践。今天，在程序中作一下小修改很容易。几分钟内，程序可以通过编译、调试，再编译这个过程。符号调试器的普遍运用使程序员可以交互地监控一个运行时的程序（进程），同时可以在源程序的符号级操作数据和指令。这样，书桌检查的工作量就不大了。

老式的环境使程序员工作起来缓慢和小心。书桌检查是一种有价值的可靠和有效的查错工具，但是在现代环境中它是不合时宜的。因为在打印一份清单然后铺在书桌上的时间里，一个人已经执行了几个编译、调试循环证实了一些猜想。但是，这种方法有时

使程序员花很多时间来一点点地啃难题的边缘，因为他不再很多地理解他的程序，抓不到困难的关键，离错误很远。

归纳方法（测试每个想到的猜测）并不是比演译方法（书桌检查）更好的调试方法。前者，可以分清错误的症状，但只有很小的注意面，对确定问题的根本原因很少有帮助。另一方面，后者，针对问题的原因可以使人猜想到症状。为了更有效，可以把两者结合起来。

一种科学的调试方法：

科学实验的方法是不断地重复实验得到一致的结果，使研究者能建立一个可测的假设并解释结果。通过独立的因子的作用实验，可以肯定一些假定，否定一些假设，或者决定那些因子的作用是和问题有关的，那些是无关的。

同样的方法可以应用到调试上。一个功能错误的程序可以一遍遍地运行，每次选择一些变量、指令或环境的变化来发现那个因素影响程序的行为。计算机是程序员的实验室。

调试过程分为四步：

1、收集可得到的有关信息。

2、重现问题，从程序的错误行为中发现模式。

3、对错误行为形成一个猜测，这要基于对模式的分析。

4、通过改变程序或数据证实或证伪猜测，用大量的真实数据验证问题解决后的正确程序。

下列是以上四步的详细讨论。

1、收集可得到的有关信息

决定那些信息是有关的常常是不容易的。事实上，人们经常低估可得到的信息的多少。依靠别人的报告是不可靠的，最好对别人说的检查两遍。非技术的用户夸大症状，混淆当前的错误和以前的错误，他们的分析很少达到目的，程序员只能靠自己的眼睛看到的东西，甚至容许他看他想看的而不是看实际存在的。

“一个职员打电话来说定货单处理系统在统计发票时出错。问题出现时，花瓣从菊花上掉下来，打印机正在跳过零。”

只有在错误发现才能决定那些信息是关键的那些是偶然的或“噪音” 收集信息时，应该避免对它们的重要性轻率地作出固定的，不牢靠的判断。

2、再现问题，从程序的错误行为中发现模式

假若一个问题不能再现，从纯粹经院角度看，不能说问题解决了。另一方面，一个不会再现的问题很难说是个问题。

很多程序是能随人们的意愿再现的。通过变换数据、指令、程序的环境，人们可以从每次运行中提取尽可能多的信息以发现影响程序行为的独立因素。但是，一个典型的程序有成千成百行语句，它们在一个复杂的环境中操纵了成打的变量和运算符。这些不可数的因素中那个应该检查它和问题的相关性？

要问的第一个问题是“什么变了？”，假若程序过去工作正常却突然停止正常工作，那么一些东西被改变了。当问题没有出现时，程序员是不注意变化的。对程序员来说，最基本的一条就是发生了变化。不管别人让他相信别的什么，他的工作就是找到那

个变化。

“一天数据处理经理打电话来说我们的一个通讯软件包停止工作了。我问他什么改变了，当然他说什么都没变。我说什么都没变不可能，但是他很固执。问了几次后，我决定唯一能做的事就是花三小时开车到售货站。我到那儿时，当我看见调制调解器的灯时，我要他演示一下问题。令我惊诧的是没有一个灯亮，甚至 DTR 。问题是调制调解器的电源插头没有插进墙上的插座。

当然，程序可能没有改变，问题已经存在但以前没有注意。这种情况下，要问问为什么是这样。是从前没有仔细地检查结果，还是新数据激发了潜伏的错误。

有时，问题只在特定的条件下发生。而在一般和简单的条件下再现问题才是有用的。

3、基于对模式的分析，对错误行为的原因形成一个猜测。

对模式的分析使程序员有希望对出错误的地方，即错误在那一部分作为猜测。假若症状表明和新修改的一般码有关，很好，因为新修改的码是可疑的，假使症候不是直接指向那点。事实上，当程序修改后，立即检查新修改的段是很有价值的，不管改变和结果之间的关系多远。

程序员应该在浏览可疑码时，检查下列类型错误：

（a）加错。如写 i++ 代替 i--。

（b）两个函数的通讯的不一致性。

（c）定义变量时的错误，导致程序以不希望的方式操作它。

经验说明在错误的复杂性和基本的错误引起的复杂性之间存在反比关系。简单错误，象写 ++ 代替 --，会在程序中引起很多不明显的奇怪的反应，同时真正的主要的错误立即引起明显的效果。花在查一个错误的时间越多，隐藏在它后面的麻烦越多。

必须指出，大错误，象系统设计的错误不服从这儿提出的方法。不计划，一定失败是大家所知的。没有人计划查错。但每个人都查错。

在查错中，应该记住：

（a）人们看他们想看的，而对在眼前的视而不见。当你看不出源码错误时，请别人看看。也许对程序不熟悉的人会立即找到错误。

（b）假若你长期没有进展，休息一下澄清大脑。散散步或喝杯啤酒，回家睡个觉。当你回来时。你会以新鲜的心情研究问题并解决它。

（c）当你在你查的地方查不到错误时，看看别的地方。看着同一段代码，口中说“这不可能”，是没用的。假如一种事真的不可能，它不会发生，在你看的地方没有东西表明它是怎样发生的，就应该看看别处。

（d）假如你发现错误，即使它和现在的问题无关，也要解决它。

（e）当你发现引起出错的错误时，不要立即重新编译。查查还有其它错误没有。

（f）假如你不知道从那儿开始查错。你就从程序头开头一直到程序尾。你要相信查出的东西是值得你的一番努力的。

（g）假设你找不到人来向他解释问题，关上门不让别人看见，面对墙解释问题。从头开始不要放过任何东西。这样会使你组织你的思想会帮助你发现忽略了的问题。虽然这看来很奇怪，但它管用。

4、通过改变程序或数据证实或证伪猜测，用大量的真实数据验证问题解决后的正确

程序。

这一步和在实验室做实验一样，充满了许多同样的问题。对程序员来说，系统化地工作是很重要的，心中要有一个清楚的计划，然后贯彻下去。

程序员对程序的理解只是一系列他认为对的假设，这些假设不过是他的盲目自信。调试要求他辨别每一个假设，看那些假设是由事实支持的，那些不是。结果常常令人吃惊。假如程序员可以任意地使用符号调试器，最有效的搜索是一条一条指令得走可疑的函数，查找对关键变量的不正常操作。目标是把可疑的函数减少到一个，然后在这个函数逐步查找直到找到出错指令。假如没有符号调试器，可以在可疑函数中写入一些 printf 语句来代替。

在修改数据和程序前最好做一下后备。

不要假设对问题的解决显然是正确的而不必去测试正确性。

"有一次我查看一个程序的注释，我不想修改源码，但是很偶然地我在星号和斜线之间打了一个空格。几周后，我修改了源码，编译程序但不通过。我忘记了上次的修改，所以花了大半天时间查错。"

在这种情况下，出现错误的不是问题也不是修改。程序员仍然应该编译和测试一下新版本。

有时，程序中有几处相似的错误，假设程序员只发现了一个。当它被修改后，仍然出错。这也是要求程序员仔细地测试所有变化的一个原因，而不管他们是多么明显和简单。

C 语言中常见错误：

1、把＝＝写成＝

if(i＝5)，可能是个错误，却是C的合法语法，表达式常常求值为真。程序员是想写 if(i＝＝5)。这种错误常出在新学 C 的 PASCAL 程序员。

2、定义一个几个元素的数组，存取元素 n 。

在 C 中，数组的第一个元常常标号为0。例如：int i〔10〕；定义一个从 i〔0〕到 i〔9〕的数组。没有 i〔10〕，虽然 C 编译器不给出错标志。

for(j＝0；j<＝10；j＋＋)　这个循环存取 i 数组将会多存取一个而出错。

3、错误的间址次数

在指针的指针这种情况下，很容易忘掉＊或多加几个＊，并且修改错误的指针。一般情况下，间址超过两次的很少用。

4、定义了指针，但没有给它将指的实体分配空间。

char＊string 这个定义只定义了一个指针而没有分配内存。若 string 是个auto 变量，它的初始值是不可予料的。

这个错误的另一种形式是：

```
char *func( )
 {
  char  Line〔128〕;
  ⋮
  return Line;
 }
```

func 返回一个指向字符的指针，但这个返回值没有意义，因为它指向一个 auto 变量，而这个变量在程序访问它之前就释放了。

5、丢掉一对括号

例如，if(i=max(a, b)==b)和 if((i=max(a, b))==b)不一样。在第一个语句中，i 被赋零或非零值由 max(a, b)是否等于 b 决定。在第二个中，i 被赋 a, b 中的大者。

另一种常见错误是把

if((fp=fopen("file_name", "r"))==NULL) 写成

if((fp=fopen("file_name", "r")==NULL))。

后者打开文件并设文件指针为0(NULL)，而前者是打开文件并给文件指针赋正确值。

缺少括号会导致优先级的问题，即使表达式不能按程序希望的求值。例如，设 p_i 是指向 i 的指针，那么*p_i++是 p_i 加1，而不是 i 加1。为了使 i 加1，表达式应写成

(*p_i)++。

多加括号是无害的，而少加括号可能出问题。

6、一行中在屏幕右端之外的部分的错误

把程序的一部分藏在屏幕的右端不是好的方法，因为在那里出错，你看不见。

7、不正确的注释行结束。

用/ *(/和*之间夹有空格)使到下一条*/之前的所有行变成注释被计算机略去，这个错误很难查。

8、你要连接的函数的变化

别的程序员在你不注意时也许修改了函数。这类错误能很快被查出因为人们总是在找机会责备别人。

9、使用错的版本

有几种错误：

(a)修改了源文件但没有存，所以编译运行了老版本并且进行同样的调试。

(b)没有注意到修改后的程序编译不成功，就运行老版本。

(c)编译了新版本但连接了老版本。

(d)用老数据而不是新数据运行。

(e)用和实际的源程序不对应的程序清单工作。

(f)(a)到(e)的各种组合 。

10、假设你的数据文件正好包含你希望的东西

在调试中，所有假设都要怀疑。只有事实是重要的。

11、实际参量和形式参量的不一致性

C 编译器不验证实参和形参在类型和数目上的一致性。所以容易犯这类错误。在UNIX中，可以用 lint 来定出错误位置，但是 lint 远非一个理想的解决方法，因为它不加区别地给出错误信息而且还得记住运行它，lint 不是编译器的一部分。

12、在函数调用时忘记了函数名