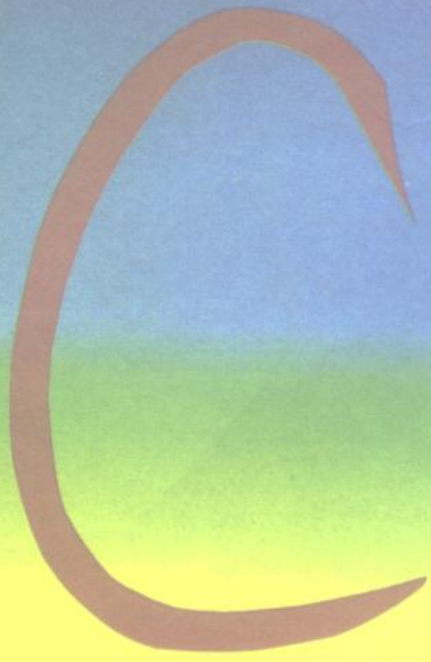
HOPE

微机高级C语言调试技巧

翟彤 译
李平 校



北京希望电脑公司

微机高级C语言调试技巧

微机高级C语言调试技巧

翟彬　　译
李平　　校

北京希望电脑公司

一九九一年六月

前　言

C 语言是一种随着 UNIX 发展起来的高级语言，它与 UNIX 系统有着互相配合的紧密关系．随着 UNIX 日益成为操作系统的标准，C 语言的应用也蓬勃发展起来．今天，使用 C 语言编程的程序员越来越多，有关 C 语言的书籍也十分丰富，但对 C 语言的调试技巧则很少有书论及到．

究其原因，第一是因为 C 语言本身是一种非常灵活而不规范的语言，它表达能力很强，但也带来了许多内在的弱点．例如，C 程序员都觉得 C 指针十分灵活好用，但正是 C 指针给程序带来了许多隐患．第二是 C 语言的调试是一门非常复杂的技术，离开对 C 语言精髓的透彻理解，没有多年的 C 编程经验，是很难对 C 语言调试评头品足的．PC 机的 C 程序员常会发现自己开发的程序经常莫名其妙的死机，既使您使用各种调试工具也是枉然，为了找出原因，许多程序员不得不花费数小时甚至数天的时间去寻找程序内的错误．

本书是一本有关 C 语言调试方面的专门论著，作者是 C 语言的专家．在本书中，作者详尽介绍了 PC 机上各种操作系统下的多种调试技巧．当您在调试 C 程序时遇到一个使您发呆的错误，与其坐在机器前东查西找，还不如认真研读本书，相信您一定会有很大的收获．

目　录

第一章　程序调试基础

开发符合设计说明的程序的代码应该包括:编辑和调试。编辑部分实现代码的输入，调试部分则使输入的程序代码逐步接近程序的设计说明。虽然要使程序完全符合设计说明是非常困难的，但没有谁能不经过调试就能编出符合设计说明的程序。这说明，程序的调试是开发程序不可缺少的一部分。

通俗地说，程序的调试就是查错和修改。我们可借用游击战比喻程序的查错，程序设计小组可能把排除程序错误描述成"搜寻与催毁"的任务。这个比喻是形容错误会隐避且不易辨识。排除程序错误首先要查找，程序调试阶段的目的在揭露错误的操作。当确定有错误存在(敌人出现)时就开始搜寻。发觉程序错误地执行某项工作后，程序员的任务是决定程序的那个部分引起错误。当然，在搜寻范围缩小到一行或一个表达式时，程序员就可以改正错误了。

然而，把调试程序说成是"搜寻与摧毁"的任务在某些方面不是很恰当。程序员搜寻了几个小时只找到一个放错位置的分号，把它说成摧毁任务是夸大了些。每一种错误都需要有特殊的搜寻技巧，因此必须个别处理。手榴弹和迫击炮在排除程序错误上是用不到的。最夸张的是把优秀的程序员与战斗中的士兵相比。相反的，好的程序员比较象实验室的技师，通过训练与练习能培养一些有效的查错所必须具有的有规律有组织的错误定位技巧。

程序查错和科学方法

许多程序员把自己当成工程师和工匠。这两种职业需要训练和技巧。设计与编写有用的程序同样需要熟练的技巧。程序员也和工程师一样采用仔细计划和专业知识从事设计，就可以产生构思巧妙，成本效益很高的程序。以工匠的熟练加上小心仔细。程序员就能设计出许多实用且耐久的程序。

排除程序错误是个不同的领域。程序员在排除程序错误过程中得到最多的就是经验。在这个领域，程序员就像个联邦调查局的侦探。仔细研究每个地方，找出程序出错的原因，并且估计修理维护的可能性。评估结果值，研究程序的运行，找出任何与规格不符或不稳定的地方。

程序设计期间程序员试着建立有效且易懂的程序。调试程序期间，程序员可能会面对一连串未知数。这个工作并不是创造性的综合体而是测试与分析。排除程序错误过程中，程序员在一些不一定有意义的信息中找寻有用信息。程序员首先尝试发掘错误所在，然后找出其原因。程序员在排除程序错误过程不仅要是个优秀观察者和记录者，而且必须有熟练测试和设计经验，并且对因果关系有相当的了解。

以科学方法为基础的实验做为程序调试工作的模式是相当有意义的。

如果考虑

- 以科学方法去了解未知事物，解释观察到的自然现象并且通过反复实验找出因果关系。
- 错误是个未知的(原始的)现象，必须了解发生原因。

科学方法与查错之间的配合就如工程与软件设计的密切关系一样．最近，软件工程的应用对软件设计有很大助益．同样的，采用科学的方法对有计划的排除程序错误应该有帮助．

排除程序错误与科学实验之关系并没有改变程序员在对程序排除错误时所应遵循的步骤，而只是加强某些手段而已．

第一　程序员必须是独立的观察员而不是多情的创造者，一旦草拟完成就应该编写、检查与了解程序代码(我们写程序，但因为程序很少能照我们的意思做，所以必须了解它真正情况)，这种不随便被激怒的态度或是选择性的遗忘不该记住的事情，对成功有决定性影响．

第二　程序员必须记录多方面的详细信息．机器可以当作秘书，但记录保存的工作必须周密的完成．

第三　排除程序错误和设计是不同形式的工作，需要不同的思考方式．轻启智慧之门要比埋头苦干更有用．

程序调试的过程

程序调试的过程包含四个阶段：测试(testing)，固定(stabilization 固定是要控制环境变量，使错误不会动态的变化)，局部化(localization)和更正(correction)．有时无法清楚区分这些阶段．例如，大多数高级语言固定一个错误通常是不重要的步骤．但是，C 语言几乎是需要集中心思于每个阶段．虽然各个阶段都不一样，但不一定要结束一个阶段才能开始另一个阶段．通常，排除程序错误是个循环的，随机性的工作．因为改正每个错误都要重复这四个阶段，所以排除程序错误是循环性．当错误改正后常常可以发现另一个新的错误，因此排除程序错误具随机性．虽然大概描述这些阶段是适当且有帮助，但是在实际工作环境中很少能把这四个阶段分辨得很清楚．

阶段一：测试

测试是输入各种不同的值来观察新程序的能力．测试阶段有二个目标：

1. 找出新程序的限制．
2. 证实设计与说明文件相符合．

这个过程类似飞机的飞行试验．虽然主要目标是证实设计与说明文件相符，但通常试飞员还会去试试其性能．换句话说，借助程序中断点(break point)的发现，测试该程序是否如我们希望的一样．

要产生完整的结果，测试必须在不同条件下检查程序能力．为了有可靠的结果，测试应该在良好控制的环境下进行．C 以外的其他语言，控制环境就是控制程序的输入即可．在 C 语言，控制环境可能需要大费周折．所有测试结果必须加以记录并且与程序的预期结果相比较，任何与预期结果不相符合的地方，可能就是错误所在，须进一步检查．

阶段二：固定

固定就是企图控制情况使得即使在程序加入某些调整指令之后，仍能在控制下重复产生错误．这种情况，调整是指程序里用来报告与程序执行的相关情况的指令，而不属于完成程序主要工作的指令。

调整通常就是指"追踪"或"查错"指令．由一些打印指令(Printf)组成，用来输出追踪结果以便知道程序执行情况．追踪指令通常在排除程序错误完成后除去．C 以外的高级语言，错误不会随着调整或追踪指令的加入而改变或消失．在 C 则不是这样。原始程序或联结过程的任何改变都会大大地改变错误出现的情况——甚至使错误消失不见。

程序员在固定阶段遭遇到最大的问题是如何紧紧控制测试条件与如何改善测试计划。使用在固定的技巧与使用在控制测试环境的技巧相类似．因此，有人也许把固定定义成详细测试．然而测试阶段和固定阶段是不同的．在测试阶段，程序员没有特定的目标(像猎人悄悄走近躲闪的猎物)．在固定阶段，猎物已经发现．在对目标开火前先要瞄准目标。

即使是使用现代化的武器装备，猎人在每次发射前都要调整"偏差"因素．因为 C 语言似乎常需要这种"人工调整"，所以许多批评者称它是旧式的或原始的。

对这个情况，典型的排除程序错误技巧更可以称得上是旧式或原始的．这里所介绍的排除程序错误技巧稍加扩充就足以应付一般的排除程序错误工作．人工调整变得很不需要．换句话说，如果排除程序错误能象设计和编码一样受重视，固定阶段(即使是 C)可以并入测试阶段里面。

阶段三：局部化

只有在错误确实存在并会重复产生时，才能开始找寻错误的原因．搜寻的进行是利用缩小可能的范围，把错误局限在特定程序段或数据变量上。

局部化阶段具有集中信息分析的特征，程序员象科学家一样研究信息，建立有关如何产生这些信息的假设，并修改实验以测试假设的可行性．有时候，因为修改后的实验没有产生可靠的结果，所以程序员必须回到固定阶段．每一个经过适当设计的实验能提供信息，便能缩小错误位置的可能范围。

发现错误(辨识错误的程序行号或变量名称)是这个阶段主要目标的副产品．主要目标是指出假设与验证假设，以便解释所有观察到错误的特征。

对具有科学精神的程序员而言，不以充分了解错误结构为基础("那种错误如何产生这个输出")的改正是不能令人满意的．实际上，这种隋性的"修理"程式并不能算是改正；只是种会破坏先前工作的瞎猜。

一般来说，局部化是个单调的过程．正确操作时就能引导程序员更接近答案．如稍后章节解释的邻近和局部化原则主要依赖三种独立的邻近形式．要正确的解释测试结果以及设计有用的实验，程序员必须对每个线索有清楚的了解。

阶段四：改正

找出错误后必须加以改正。通常改正是最简单的．但是如果是设计上的错误，改正的

代价就相当高。但是无论是改正简单错误或是改正设计上的错误，都不在本书所谈的程序调试技巧范围内。

证实改正的正确性是很重要的。支持改正的实验必须在程序修改之后重做一次。此外，为了证实改正没有搅乱任何东西，细心的程序员必须准备一份无错误的测试报告。

邻近(proximity)原理

每个领域都有一些基本原则。电子领域里就是欧姆定律。物理学上就是物质不灭定理。排除程序错误过程也要依赖一些基本的法则和性质。最主要却也最无助益的定理是小心谨慎。程序员必须相信每个结果必有原因。否则为什么会出现困扰？错误发生是没有任何规律性的。

更实际一点，此定理的限制形式是邻近原理。每个"果"在某些尺度上接近它的"因"。程序有三种形式的接近：语句，时间和空间。所以排除程序错误的第一个原理是：每个"果"都接近它的"因"，不是语句，就是时间或空间性的。

在搜寻错误时，我们靠这些邻近定理的引导帮助我们知道何时该动手了。

语句的邻近

如果两段程序出现在程序列表的相邻位置，他们就是语句相邻。

在这个例子中，语句是指编写的方式。语句邻近是在打印的原始列表上实际相邻的二行。程序员靠语句邻近，因为时间上和空间上的邻近只有在执行阶段才存在。编译器会在错误信息里提供行号帮助程序员。

```
1:      main()
2:
3:      {
4:      int i;
5:
6:      for ( i = 0; i < 5; i++ )
7:         printf( "\nHello World" )
8:      printf( "\nGoodbye World\n" );
9:      }

******** Fatal Error in line 8. Expecting ';' ********
```

表 1.1　编译阶段的错误

错误信息指出第 8 行，所以程序员估计在第 7 行或第 8 行可以找到错误。这个估计对许多例子，包括此例子而言是很合理的。然而，请记住编译器指出的行号只有在最后一行能够产生错误时才保证有用，真正的错误可能隐藏在第一行到编译器所指定之间的任何地方。极端的例子里，在一个只有 742 行的程序里，编译器产生的错误信息可能会是：

```
* * * * * * * fatal error:at or before line 742 * * * * *
```

这个信息告诉你程序的第一行到最后一行之间的某个地方有语法错误。第三章介绍一些通用的方法强迫编译器产生较有用的信息。

时间上的邻近

如果二个程序段，一个紧接在另一个之后执行就是时间上的邻近。

时间上的相邻是执行阶段才有的特性。也就是说，时间相邻只有在程序真正执行时才会发现。循环的程序段里，语句和执行时间的次序是相同的。当执行到控制结构时(诸如分支，循环或函数调用)，程序执行次序将与语句次序不同。

```
1:      main()
2:
3:      {
          .
          .
          .
40:    while ( k < 7 ){
41:       dotest();
          .
          .
          .
          .
234:      proc1();
235:    }
236: }
```

表 1.2　时间上邻近的例子

在这个例子里，41 行和 234 行语句上是远离的。然而，在 K 小于 7 的情形下，当执行到 234 行时，234 行与 40 行变成执行时间上的相邻。换句话说，执行时受 while 控制结构的影响，41 行紧跟在 234 行之后。

上面的例子说明语句相邻和时间相邻之间的差异。语句邻近是静态的而且在程式产生时就决定了。时间上的邻近是动态的，由执行阶段的状况决定。如果 K 在第 40 行时等于 6，但在第 234 行之前大于 7，则循环不再重复，并且 detest()和 Proc1()之间维持时间上的疏远。传统追踪方法是有用的，因为他们不是表现出时间上邻近关系就是表现出空间上的相邻近。

空间上的邻近

如果二个程序段共同访问(reference)一个变量，而且在他们访问之间没有其他程序再访问到该变量，则称此二段程序对该变量的访问具有空间上的邻近。

当程序员怀疑程序的某一行时，就会对空间访问邻近原理有反应。因为"它是改变此变量的最后一个操作"(变量现在的值是错的)。

在表 1.3 里，第 5 行和第 13 行在语句和时间上是远离的关系。但在空间上变量访问是相邻近。他们是访问变量 i 最邻近的两行(执行时)。

```
1:    main()
2:    {
3:    int i, j, k;
4:
5:    i = 0;
6:    for ( j = 0; j < 19; j++ ) {
7:      for (k = 0; k < j; k++ ) {
8:         putchar( " " );
9:         }
10:     printf( "*\n" );
12:     }
13:   printf( "%d\n", i );
14:   }
```

表 1.3　第 5 行和 13 行具空间访问性的邻近

在单用户的机器环境下，某些指针(Pointer)错误会破坏语句邻近的性质．某些环境里，有些程序行会没有出现访问指令就改变变量内容，至少在原始程序中没有发现该访问(通常发生在使用指针时)．这种情况发生时，空间访问邻近原理就会从"基本的辅助排除程序错误性质"中除去，造成许多传给排除程序错误方法变成不可靠及无效率．稍后几章将会介绍这种举动的因果关系，并且提供使用在传统方法场合上的排除程序错误工具和追踪技巧。

调试工具和技巧

聪明的程序员会从各种硬件与软件的支持中选择工具和追踪技巧——工具从笨拙到精巧都有。事实上，主要的工具中，即使是笨拙的追踪指令也有许多种变化。追踪指令可以用来追踪时间或空间上邻近的关系，用来维护或分解程序结构，监督虚拟或物理资源。

虚拟资源是指某一特定程序语言里机器所拥有的资源。物理资源是真实的物理设备。

例如，一个以 8086-8087 为基础，没有数学协处理器(Coprocessor: 指受到和 CPU 同等待遇的专用处理器)的系统，浮点数运算就是虚拟资源。若是具有协处理器，浮点数运算则是物理资源。这两个例子，C 所表现出来的计算函数能力就是虚拟资源。

许多集成的程序调试工具用来取代追踪指令。机器码级的断点程序调试器允许在执行时以机器(物理)资源的观点不改变程序区域来追踪。同样的，符号调试器允许以逻辑(虚拟)资源的观点，不必在程序内加入追踪指令来追踪程序。良好的断点程序调试器允许初设追踪和动态的改变追踪。程序解释器在自动监督逻辑设施使用时，在符号阶层提供追踪能力以简化程序错误的测试和局部化阶段。

并非所有的程序调试工具都跟追踪有关。交叉访问(cross reference)程序指出访问到某些变量或函数名称的指令码，还有一些其他程序补足编译器上的语法检查。语法检查程序的例子从简单的括号检查到复杂的语法分析器都有。

所有硬件程序调试器都有助于与追踪相关的函数。逻辑分析器不是真正分析逻辑而是掌握机器层次的活动。硬件模拟器(emulator)能完成类似工作，但允许控制更多的测试条件。

结 论

程序的调试不一定是冒险的过程。因为排除程序错误拥有根植于可说明的原理和性质的统一结构。排除程序错误的过程值得归类成一门科学，而不是一种神秘的技术。

本章介绍程序调试过程的结构并且把排除程序错误过程的原理加以形式化。有了这些原理做基础，以后各章节再解释程序调试时如何调合这些过程与邻近原理的关系。尤其以这些基础做为程序调试技巧的引导，以适合 C 语言的各种复杂情况。

第二章　C语言及C程序的测试

C语言历史和特点

C是UNIX系统的主力语言，它与UNIX系统有着互相依存、休戚与共的紧密关系。UNIX系统是美国贝尔实验室的K.Thompson和D.M.Ritchie从1969年开始，用不到两个人年的时间研制成的。该系统最早运行在DEC的PDP-7上的。在UNIX上实现了汇编语言后，UNIX系统又以汇编语言编写。由于汇编语言的不可移植，并且描述问题的效率不如高级语言。特别是可读性差。所以K.Thompson决定开发一种高级语言来描述UNIX系统。1970年K.Thompson在PDP-11/20上实现了B语言并用B写了UNIX操作系统和绝大多数实用程序。B语言主要思想源于BCPL语言。BCPL是M.Richards基于CPL语言在1969年发表的一种语言，它是一种单一数据型语言，但即使现在，仍显示它足够的生命力。不过，由于下列几点原因，B没有盛行起来而导致D.M.Ritchie于1971年开始开发C语言：第一，PDP-11是字节编址的，而B却面向字，这样就妨害了对单个字节进行存取的能力。第二，现代的高级语言都要求有强有力的类型结构，而BCPL和B语言无类型性(或者说只是一种机器字类型)在描述客观世界许多事物时会遇到相当多困难。Algd168和Pascal语言是强类型型语言，它们，特别是后者数据类型丰富是其最大的优点之一。第三，B编译程序产生的是解释执行的代码，运行速度比较慢。当然还有一些次要的原因。总之，C语言的开发已是势在必行的。1972年C已经投入使用。

1973年UNIX系统用C重写了一遍。系统的代码量比以前的版本大了三分之一，加进了多道程序设计功能，特别是整个系统(包括C语言编译程序本身)建立在C语言基础上，而C语言又具有良好的可移植性，所以第五版的UNIX系统(UNIX V)就奠定了UNIX系统的基础。以后UNIX V6,V7,SYStemⅢ和最新的SYSTEMV都是在V，基础上发展，扩充的。

今天UNIX系统已在全世界范围内取得了巨大的成功，它几乎成了16位微型机的标准操作系统，在PDP-11小型机，VAX-11超级小型机，甚至象IBM 370，4300，UNIXVAC，Honeywell等大型机上也有它的踪迹。从某种角度可以说，没有C语言就没有UNIX今天的巨大成功。当然，没有UNIX，C语言也不会有今天。

虽然最初的C语言是附属于UNIX系统且在PDP-11上实现的，但是目前C语言却独立于UNIX系统，独立于PDP-11机而蓬勃发展，它适应的机种从8位微型机(比如以Z80为CUP的Cromemco)到Cray-Ⅰ巨型机。它附着的操作系统从8位微型机上的单用户CP/M到大型机的IBM VS/370.它与FORTRAN Pascal等语言一样已经成了微、小、超小、大、超大和巨型机上共同使用的语言。

人们喜欢用Pascal与C比较，其实它们主要都是Algol语言系统的发展，并各有其特点。