TURING 图灵程序设计丛书

The Art of Debugging with GDB, DDD, and Eclipse

软件调试的艺术

[美] Norman Matloff
Peter Jay Salzman 著
张云 译

GDB

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

The Art of Debugging with GDB, DDD, and Eclipse

软件调试的艺术

[美] Norman Matloff
Peter Jay Salzman 著
张云 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

软件调试的艺术 /（美）马特洛夫（Matloff, N.），（美）萨尔兹曼（Salzman, P. J.）著；张云译．—北京：人民邮电出版社，2009.11
（图灵程序设计丛书）
书名原文：The Art of Debugging with GDB, DDD, and Eclipse
ISBN 978-7-115-21396-9

Ⅰ. 软… Ⅱ. ①马…②萨…③张… Ⅲ. 软件－调试 Ⅳ. TP311.5

中国版本图书馆CIP数据核字（2009）第162737号

内 容 提 要

调试对于软件的成败至关重要，正确使用恰当的调试工具可以提高发现和改正错误的效率。本书详细介绍了3种调试器，GDB用于逐行跟踪程序、设置断点、检查变量以及查看特定时间程序的执行情况，DDD是流行的GDB的GUI前端，而Eclipse提供完整的集成开发环境。书中不但配合实例讨论了如何管理内存、理解转储内存、跟踪程序找出错误等内容，更涵盖了其他同类书忽略的主题，例如线程、客户/服务器、GUI和并行程序，以及如何躲开常见的调试陷阱。

本书适合各层次软件开发人员、管理人员和测试人员阅读。

图灵程序设计丛书

软件调试的艺术

◆ 著　[美] Norman Matloff　Peter Jay Salzman
译　张　云
责任编辑　傅志红
执行编辑　武　嘉

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京隆昌伟业印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：14.25
字数：337千字　2009年11月第1版
印数：1－3 000册　2009年11月北京第1次印刷

著作权合同登记号　图字：01-2008-5834号

ISBN 978-7-115-21396-9

定价：39.00元

读者服务热线：(010)51095186　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

Copyright © 2008 by Norman Matloff and Peter Jay Salzman. Title of English-language original: *The Art of Debugging with GDB, DDD, and Eclipse,* ISBN 978-1-59327-174-9, published by No Starch Press. Simplified Chinese-language edition copyright © 2009 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由No Starch Press授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

前　言

“嘿，还真不错呢！”我们的学生Andrew首次认真使用调试工具后惊叹道。Andrew三年前在大一上编程课时就学过了调试工具，不过那时他只是将调试工具当做应付期末考试的内容来打发的。现在他已经大四了，教授强烈要求他停止采用输出语句进行调试的方法，改为使用正式调试工具。让Andrew喜出望外的是，他发现使用恰当的工具可以大大缩短调试时间。

如今，在学生及已参加工作的程序员中，还有不少“Andrew”，但愿本书能帮助“Andrew们”发现调试工具的好处。但是，我们更希望本书能帮助那些已经使用了调试工具，但还无法确定在什么情况下该做什么事的程序员做出适当的判断。本书也适用于打算进一步学习调试工具及其幕后原理的人。

本书的编辑曾说过，很多调试知识以前都只是在圈子里口口相传，没有正式编著成书。本书将改变这一状况。本书回答了下列问题。

- 如何调试线程代码？
- 为什么有时候断点最终出现的位置与原来设置的位置略有偏差？
- GDB的`until`命令为什么有时会跳到意想不到的地方？
- 如何巧妙使用DDD和Eclipse？
- 在当今普遍使用GUI的时代，像GDB这样的基于文本的应用程序还有价值吗？
- 为什么当错误代码超出数组边界时不发生段错误？
- 为什么要将我们的一个示例数据结构命名为`nsp`？（不好意思，这只是与我们的出版商No Starch Press私下开的一个玩笑。）

本书既没有美其名曰“用户手册”，也不是关于调试过程认知理论的抽象论文。本书有点介于这两者之间。一方面，确实提供了如何操作GDB、DDD和Eclipse中特定命令的信息；另一方面，讲解并频繁使用了关于调试过程的一些通用原则。

我们之所以选择GDB、DDD和Eclipse，是因为它们在Linux/开源社区中比较流行。本书的示例有点偏向于GDB，不仅仅因为GDB基于文本的性质使得显示在一个页面中更紧凑，而且正如上面的问题所暗示的，我们发现基于文本的命令在调试过程中仍然起着举足轻重的作用。

Eclipse的用途越来越广泛，其远不止仅作为我们这里的调试角色，还提供了各种有吸引力的调试工具。另一方面，DDD占用空间小，而且包括了Eclipse中不具备的某些强大功能。

第1章是概览。很多经验丰富的程序员可能想跳过这一章，但是我们强烈建议大家通读一遍，因为这一章列出了我们针对调试过程推荐的一些简单却有用的通用准则。

试读结束：需要全本请在线购买： www.ertongbook.com

第2章着重介绍了调试必不可少的内容——断点，讨论了关于断点的所有细节，包括设置、删除和禁用断点，从一个断点移到另一个断点，查看关于断点的详细信息，等等。

到达断点后会出现什么情况呢？第3章回答了这一问题。我们这里采用的示例涉及遍历树的代码，重点是介绍当到达断点时如何方便地显示树中节点的内容。这里GDB相当出色，提供了一些非常灵活的功能，有助于在每次程序暂停时有效地显示感兴趣的信息。这一章还提供了一个特别优秀的用图形显示树和其他链接数据结构的DDD功能。

第4章包括了由于段错误而产生的致命运行时错误。我们首先介绍了崩溃时在底层发生了什么事，包括程序的内存分配以及硬件与操作系统的协同作用。对系统知识比较了解的读者可能会跳过这一章，但是我们相信，其他很多读者会得益于这一章介绍的基础知识。然后介绍了核心文件，包括如何创建核心文件，如何使用它们来完成“事后反思”。最后介绍了关于调试会话的一个扩展示例，其中有几个程序错误产生了段错误。

第5章不但介绍并行编程，而且包括网络代码。客户/服务器网络编程可算作并行处理，甚至我们的工具也是并行使用的。比如，在两个窗口中使用GDB，一个窗口用于客户机，另一个窗口用于服务器。由于网络代码涉及系统调用，因此我们用C/C++的`errno`变量和Linux的`strace`命令补充我们的调试工具。5.2节涉及线程编程。这里同样首先概述基本内容：分时、进程与线程、竞争条件等。这一章介绍了在GDB、DDD和Eclipse中使用线程的技术细节，并再次讨论了一些要记住的通用原则，比如发生线程上下文切换时的时间选择随机性。5.4节介绍了用流行的MPI和OpenMP程序包进行并行编程，并举了一个OpenMP的扩展示例。

第6章包括其他一些重要主题。如果代码不能编译，那么调试工具将无能为力，因此这一章讨论了处理这种问题的一些方法。然后处理由于缺少必要的库造成的连接失败问题，我们再一次觉得提供一些“理论”是有用的，比如库的类型以及如何将库与主要代码连接。如何调试GUI程序呢？为了简便起见，我们这里坚持采用“半GUI”设置——curses编程，并显示如何让GDB、DDD和Eclipse与curses窗口中的事件交互。

正如前面提到的，可以使用辅助工具使调试过程得到很大的增强，第7章介绍了部分辅助工具，还介绍了`errno`和`strace`、`lint`的一些内容，以及对于有效使用文本编辑器的一些提示。

全书主要介绍的是C/C++编程的调试，第8章则谈到了其他语言，包括Java、Python、Perl和汇编语言。

如果漏掉了读者喜欢的调试主题，我们感到抱歉，书中包括了我们自己编程时发现有用的全部内容。

感谢No Starch Press的诸位工作人员在这个项目上花了很长时间来协助我们。尤其感谢公司的创始人及总编辑Bill Pollock。他一开始就对我们这个“非常规”项目抱有信心，并宽容了我们的多次延误。

Daniel Jacobowitz对本书的初稿做了认真的审查，并提出了许多宝贵建议。还有Neil Ching，虽然他表面上是做文字编辑的，实际上却是拥有计算机学士学位的高材生。他在我们的技术讨论中提出了一些重要看法。本书质量的提高很大程度上得益于从Daniel和Neil处得到的反馈。当然，照例要做一下免责声明：如果还有错误，那一定是我们自己造成的。

Norm的致谢。感谢我的妻子Gamis和女儿Laura，这两个可爱的女子让我觉得非常幸福。尽管这本书她们一个字也没看过，但是她们解决问题的方法、幽默的性格和对生活的热爱深深地影响了我。还要感谢这么多年来我教过的学生，他们教会我的与我教会他们的一样多，是他们让我觉得我选对了职业。我一直致力于“有所作为”，但愿这本书能算作我的一个小小作为。

Pete的致谢。我还要感谢Nicole Carlson、Mark Kim和Rhonda Salzma，花了不少时间来通读本书的各章，提出了更正与建议，感谢你们阅读本书。还要感谢Davis的Linux用户组上这么多年来回答我问题的人们，认识你们使我更聪明。感谢Evelyn提升了我生活的方方面面。特别值得一提的是小猫咪Geordi，它总是趴在我的稿纸上，以免稿子被吹散，还时常为我温暖座椅，并值守书房。我每天都在想念你们。小家伙，睡吧。妈妈，看儿子棒吗？

Norm Matloff与Pete Salzman
于2008年6月9日

目 录

第 1 章 预备知识 ········ 1

1.1 本书使用的调试工具 ········ 1

1.2 编程语言 ········ 2

1.3 调试的原则 ········ 2

1.3.1 调试的本质：确认原则 ········ 2

1.3.2 调试工具对于确认原则的价值所在 ········ 2

1.3.3 其他调试原则 ········ 3

1.4 对比基于文本的调试工具与基于GUI的调试工具，两者之间的折中方案 ········ 4

1.4.1 简要比较界面 ········ 4

1.4.2 折中方法 ········ 9

1.5 主要调试器操作 ········ 11

1.5.1 单步调试源代码 ········ 11

1.5.2 检查变量 ········ 12

1.5.3 在GDB、DDD和Eclipse中设置监视点以应对变量值的改变 ········ 14

1.5.4 上下移动调用栈 ········ 14

1.6 联机帮助 ········ 15

1.7 初涉调试会话 ········ 16

1.7.1 GDB方法 ········ 18

1.7.2 同样的会话在DDD中的情况 ········ 31

1.7.3 Eclipse中的会话 ········ 34

1.8 启动文件的使用 ········ 38

第 2 章 停下来环顾程序 ········ 39

2.1 暂停机制 ········ 39

2.2 断点概述 ········ 39

2.3 跟踪断点 ········ 40

2.3.1 GDB中的断点列表 ········ 40

2.3.2 DDD中的断点列表 ········ 41

2.3.3 Eclipse中的断点列表 ········ 42

2.4 设置断点 ········ 42

2.4.1 在GDB中设置断点 ········ 42

2.4.2 在DDD中设置断点 ········ 45

2.4.3 在Eclipse中设置断点 ········ 46

2.5 展开GDB示例 ········ 46

2.6 断点的持久性 ········ 48

2.7 删除和禁用断点 ········ 50

2.7.1 在GDB中删除断点 ········ 50

2.7.2 在GDB中禁用断点 ········ 51

2.7.3 在DDD中删除和禁用断点 ········ 51

2.7.4 在Eclipse中删除和禁用断点 ········ 53

2.7.5 在DDD中“移动”断点 ········ 53

2.7.6 DDD中的Undo/Redo断点动作 ········ 54

2.8 进一步介绍浏览断点属性 ········ 55

2.8.1 GDB ········ 55

2.8.2 DDD ········ 56

2.8.3 Eclipse ········ 56

2.9 恢复执行 ········ 56

2.9.1 在GDB中 ········ 57

2.9.2 在DDD中 ········ 64

2.9.3 在Eclipse中 ········ 66

2.10 条件断点 ········ 66

2.10.1 GDB ········ 67

2.10.2 DDD ······ 69
2.10.3 Eclipse ······ 69
2.11 断点命令列表 ······ 70
2.12 监视点 ······ 74
2.12.1 设置监视点 ······ 75
2.12.2 表达式 ······ 77
第3章 检查和设置变量 ······ 78
3.1 主要示例代码 ······ 78
3.2 变量的高级检查和设置 ······ 80
3.2.1 在GDB中检查 ······ 80
3.2.2 在DDD中检查 ······ 84
3.2.3 在Eclipse中检查 ······ 86
3.2.4 检查动态数组 ······ 88
3.2.5 C++代码的情况 ······ 90
3.2.6 监视局部变量 ······ 92
3.2.7 直接检查内存 ······ 92
3.2.8 print和display的高级选项 ······ 93
3.3 从GDB/DDD/Eclipse中设置变量 ······ 93
3.4 GDB自己的变量 ······ 94
3.4.1 使用值历史 ······ 94
3.4.2 方便变量 ······ 94
第4章 程序崩溃处理 ······ 96
4.1 背景资料：内存管理 ······ 96
4.1.1 为什么程序会崩溃 ······ 96
4.1.2 内存中的程序布局 ······ 97
4.1.3 页的概念 ······ 99
4.1.4 页的角色细节 ······ 99
4.1.5 轻微的内存访问程序错误可能不会导致段错误 ······ 101
4.1.6 段错误与Unix信号 ······ 102
4.1.7 其他类型的异常 ······ 105
4.2 核心文件 ······ 106
4.2.1 核心文件的创建方式 ······ 106
4.2.2 某些shell可能禁止创建核心文件 ······ 107
4.3 扩展示例 ······ 108
4.3.1 第一个程序错误 ······ 111
4.3.2 在调试会话期间不要退出GDB ······ 113
4.3.3 第二个和第三个程序错误 ······ 113
4.3.4 第四个程序错误 ······ 115
4.3.5 第五个和第六个程序错误 ······ 116
第5章 多活动上下文中的调试 ······ 120
5.1 调试客户/服务器网络程序 ······ 120
5.2 调试多线程代码 ······ 125
5.2.1 进程与线程回顾 ······ 125
5.2.2 基本示例 ······ 127
5.2.3 变体 ······ 132
5.2.4 GDB线程命令汇总 ······ 133
5.2.5 DDD中的线程命令 ······ 134
5.2.6 Eclipse中的线程命令 ······ 134
5.3 调试并行应用程序 ······ 136
5.3.1 消息传递系统 ······ 136
5.3.2 共享内存系统 ······ 141
5.4 扩展示例 ······ 143
5.4.1 OpenMP概述 ······ 143
5.4.2 OpenMP示例程序 ······ 144
第6章 特殊主题 ······ 155
6.1 根本无法编译或加载 ······ 155
6.1.1 语法错误消息中的“幽灵”行号 ······ 155
6.1.2 缺少库 ······ 160
6.2 调试GUI程序 ······ 162
第7章 其他工具 ······ 172
7.1 充分利用文本编辑器 ······ 172
7.1.1 语法突出显示 ······ 172
7.1.2 匹配括号 ······ 174
7.1.3 Vim与makefile ······ 175
7.1.4 makefile和编译器警告 ······ 176
7.1.5 关于将文本编辑器作为IDE的最后一个考虑事项 ······ 177
7.2 充分利用编译器 ······ 178

7.3 C语言中的错误报告 …… 178
7.4 更好地使用strace和ltrace …… 182
7.5 静态代码检查器：lint与其衍生 …… 184
7.5.1 如何使用splint …… 185
7.5.2 本节最后注意事项 …… 185
7.6 调试动态分配的内存 …… 185
7.6.1 检测DAM问题的策略 …… 188
7.6.2 Electric Fence …… 188
7.6.3 用GNU C库工具调试DAM问题 …… 190
第8章 对其他语言使用GDB/DDD/Eclipse …… 196
8.1 Java …… 196
8.1.1 直接使用GDB调试Java …… 198
8.1.2 使用DDD与GDB调试Java …… 201
8.1.3 使用DDD作为JDB的GUI …… 201
8.1.4 用Eclipse调试Java …… 201
8.2 Perl …… 202
8.2.1 通过DDD调试Perl …… 204
8.2.2 在Eclipse中调试Perl …… 206
8.3 Python …… 207
8.3.1 在DDD中调试Python …… 208
8.3.2 在Eclipse中调试Python …… 209
8.4 调试SWIG代码 …… 210
8.5 汇编语言 …… 213

第1章

预备知识

有些读者，特别是专业人士，可能想要跳过本章内容。然而，我们建议每个人都应该至少简单浏览一下。许多专业人士会从中发现一些对于他们来说是全新的内容。无论如何，所有读者都应该熟悉这些内容，这一点很重要，因为其后的各章都将用到这些内容。当然，初学者更应该仔细地阅读本章。

本章前几节将概述调试过程并介绍调试工具的作用，然后在1.7节给出一个扩展性示例。

1.1 本书使用的调试工具

本书中，我们将阐明调试的基本原则，并且在如下调试工具的环境中说明这些基本原则。

- GDB

Unix程序员最常用的调试工具是GDB，这是由Richard Stallman（开源软件运动的领路人）开发的GNU项目调试器（GNU Project Debugger），该工具在Linux开发中扮演了关键的角色。

大多数Linux系统应该预先安装了GDB。如果没有预先安装该工具，则必须下载GCC编译器程序包。

- DDD

随着GUI（图形用户界面）越来越流行，大量在Unix环境下运行的基于GUI的调试器被开发出来。其中的大多数工具都是GDB的GUI前端：用户通过GUI发出命令，GUI将这些命令传递给GDB。DDD（Data Display Debugger，数据显示调试器）就是其中的一种工具。

如果你的系统还没有安装DDD，则可以下载该工具。例如，在Fedora Linux系统上，命令

```
yum install ddd
```

将自动处理整个安装过程。在Ubuntu Linux上，可以使用命令`apt-get`。

- Eclipse

有些读者可能使用过IDE（集成开发环境）。IDE也是一种调试工具，它在一个程序包中集成了编辑器、生成工具、调试器和其他开发辅助程序。本书中的示例IDE是非常流行的Eclipse系统。与DDD一样，Eclipse在GDB或其他一些调试器的基础上运行。

可以通过如上所述的`yum`或`apt-get`命令安装Eclipse，也可以直接下载相应的*.zip*文件，在适

试读结束：需要全本请在线购买： www.ertongbook.com

当的目录（例如*/user/local*）中解压缩该文件。

本书使用3.3版本的Eclipse系统。

1.2　编程语言

本书主要面向C/C++编程，并且将在该环境中完成大多数示例。不过，第8章也会讨论其他语言。

1.3　调试的原则

虽然调试是一门艺术而非科学，但是仍然有一些明确的原则来指导调试的实践。本节将讨论其中一些原则。

至少其中的一个规则，即确认的基本原则（Fundamental Principle of Confirmation）在本质上是相当正式的原则。

1.3.1　调试的本质：确认原则

下面的规则是调试的本质。

- *确认的基本原则*

修正充满错误的程序，就是逐个确认，你自认为正确的许多事情所对应的代码确实是正确的。当你发现其中某个假设不成立时，就表示已经找到了关于程序错误所在位置（可能并不是准确的位置）的线索。

换一种表达方式来说：惊讶是好事！

当你认为关于程序的某件事情是正确的，而在确认它的过程中却失败了，你就会感到惊讶。但这种惊讶是好事，因为这种发现会引导你找到程序错误所在的位置。

1.3.2　调试工具对于确认原则的价值所在

传统的调试技术只是向程序中添加跟踪代码以在程序执行时输出变量的值，例如使用`printf()`或`cout`语句输出变量的值。你可能会问："这样操作够吗？为什么要使用GDB、DDD或Eclipse这样的调试工具？"

首先，这种方法要求有策略地持续添加跟踪代码，重新编译程序，运行程序并分析跟踪代码的输出，在修正程序错误之后删除跟踪代码，并且针对发现的每个新的程序错误重复上述这些步骤。这种工作过程非常耗时，并且容易令人疲劳。最为重要的是，这些操作将你的注意力从实际任务转移开，并且降低了集中精力查找程序错误所需的推理过程的能力。

相反，通过使用DDD和Eclipse这样的图形调试工具，只需要将鼠标指针移动到代码显示中的变量实例上方就可以检查该变量的值，并且此时会显示该变量的当前值。为什么还要在整夜的调试中使用`printf()`语句来输出变量的值，使自己更加疲劳，等待更长的时间呢？放松心情，使用调试工具可以减少所花费的时间和需要忍受的厌烦感。

使用调试工具不仅仅能够查看变量。在许多情况下，调试器会指出程序错误所在的大概位置。

例如，程序由于段错误（即内存访问错误）而崩溃。在本章后面的样本调试会话中可以看到，GDB/DDD/Eclipse可以立刻指出段错误所在的位置，这一般就是（或接近于）程序错误所在的位置。

类似地，调试器要求用户设置监视点（watchpoint），通过监视点可以了解在程序运行期间某个变量的值到达可疑值或范围的具体位置，而通过查看调用`printf()`获得的输出很难推断出这种信息。

1.3.3 其他调试原则

- 从简单工作开始调试

在调试过程的开始阶段，应该从容易、简单的情况开始运行程序。这样做也许无法揭示所有的程序错误，但是很有可能发现其中的部分错误。例如，如果代码由大型循环组成，则最容易发现的程序错误是在第一次或第二次迭代期间引发的程序错误。

- 使用自顶向下的方法

你可能已经了解如何使用自顶向下或模块化的方法来编写代码：主程序不应该过长，并且它应该主要包含对执行实际工作的函数的调用。如果某个执行实际工作的函数较长，则应该考虑将该函数依次分解为多个较小的模块。

除了应该以自顶向下的方式编写代码之外，也应该以这种方式调试代码。

例如，假设程序使用函数`f()`。在使用调试工具单步调试代码并且遇到对`f()`的调用时，调试器将为你提供选择执行过程中下一次暂停位置的机会——是在调用函数的第一行还是在跟在函数调用后的语句。在许多情况下，后一种选择是较好的初始选择：执行调用，然后检查依赖于调用结果的变量的值，从而了解该函数是否正确运行。如果该函数正确运行，就可以避免单步调试函数中代码这样既浪费时间又不必要的工作，这并不是错误的行为（在该示例中）。

- 使用调试工具确定段错误的位置

当发生段错误时，执行的第一步操作应该是在调试器中运行程序并重新产生段错误。调试器将指出发生这种错误的代码行。然后，可以通过调用调试器的反向跟踪（backtrace）功能获得其他有用信息，该功能显示导致调用引发错误的函数的函数调用序列。

在某些情况下，可能很难重新产生段错误，但是如果有核心文件（core file），则仍然可以执行反向跟踪以确定产生段错误的情况，第4章将讨论这些。

- 通过发出中断确定无限循环的位置

如果怀疑程序中存在无限循环，则进入调试器并再次运行程序，让该程序执行足够长的时间以进入循环。然后，使用调试器的中断命令挂起该程序，并且执行反向跟踪，了解已到达循环主体的哪个位置以及程序是如何到达该位置的。（该程序还没有被取消执行，可以根据需要恢复执行。）

- 使用二分搜索

你可能已经在有序列表的环境中看到过二分搜索。例如，假设你有一个包含按升序排列的500个数字的数组`x[]`，并且希望确定在何处插入新的数字`y`。首先将`y`与元素`x[250]`进行比较，如果结果是`y`小于该元素，接下来就将`y`与元素`x[125]`进行比较；但是如果`y`大于`x[250]`，则下一次就将`y`与`x[375]`进行比较。在后一种情况中，如果`y`小于`x[375]`，则将其与`x[312]`进行比较，`x[312]`

是x[250]和x[375]之间的中间元素，以此类推。在每次迭代过程中保持将搜索范围减半，从而快速查找到插入点。

在调试过程中也可以应用二分搜索的原理。假设你知道在最初的1 000次循环迭代期间，存储在某个变量中的值在某个时刻出现问题。可以帮助跟踪该值第一次出现问题时所在迭代过程的一种方法是使用监视点，1.5.3节将讨论这种高级技术。另一种方法则是使用二分搜索，采用这种方法可以节省时间而非空间。首先检查该变量在第500次迭代期间的值；如果此时该值仍然良好，则下一次检查位于第750次迭代期间的该值，以此类推。

再举另一个示例，假设程序中的某个源文件根本无法编译。由语法错误产生的编译器消息中，提及的代码行有时与实际的错误位置相去甚远，因此可能很难确定该位置。在这种情况下，二分搜索就可以派上用场：删除（或注释掉）编译单元中的一半代码，重新编译剩余的代码并查看错误消息是否继续存在。如果错误消息仍然存在，则错误就位于另一半代码中；如果错误消息不再出现，则错误就位于删除的那一半代码中。一旦确定了包含程序错误的一半代码，就可以进一步将程序错误限制在这半部分代码的一半，继续执行相同的操作直到确定问题所在位置。当然，在开始该过程之前应该建立原始代码的副本，或者更好的方法是使用文本编辑器的撤销功能。参见第7章，了解在编程时适当地使用编辑器的技巧。

1.4　对比基于文本的调试工具与基于 GUI 的调试工具，两者之间的折中方案

本书讨论的GUI（DDD和Eclipse）充当用于C和C++的GDB以及其他调试器的前端。虽然GUI具有显眼的外观并且使用起来也比基于文本的GDB更为方便，但是本书中的观点是基于文本的调试器和基于GUI的调试器（包括IDE）在不同环境下分别有其用途。

1.4.1　简要比较界面

为了快速了解基于文本的调试工具和基于GUI的调试工具之间的区别，来看看将它们用做本章中运行示例的情况。该示例中的程序是*insert_sort*，通过源文件*ins.c*编译该程序，其完成的是插入排序。

1. GDB：纯文本

为了使用GDB启动针对这个程序的调试会话，可以在Unix命令行中输入如下命令：

```
$ gdb insert_sort
```

在此之后，GDB会通过显示如下提示符来邀请你提交命令：

```
(gdb)
```

2. DDD：GUI调试工具

使用DDD时，通过在Unix命令行中输入如下命令来开始调试会话：

```
$ ddd insert_sort
```

此时将打开DDD窗口，在此之后就可以通过GUI提交命令。

DDD窗口的一般外观如图1-1所示。可以看到，DDD窗口在各个子窗口中安排信息。

- 源文本窗口显示源代码。DDD从`main()`函数处开始显示，但是可以通过使用窗口右侧的滚动条移动到源文件的其他部分。
- 菜单栏提供各种菜单类别，包括File（文件）、Edit（编辑）和View（视图）。
- 命令工具列出最常见的DDD命令（例如Run、Interrupt、Step和Next），从而可以快速访问这些命令。
- 控制台：回顾一下，DDD只是GDB（和其他调试器）的GUI前端。DDD将通过鼠标执行的选择转换为对应的GDB命令。这些命令和它们的输出显示在控制台窗口中。此外，可以直接通过控制台窗口提交命令给GDB，这是非常方便的功能，因为并不是所有的GDB命令都有对应的DDD组件。
- 数据窗口显示已经请求连续显示的变量的值。在执行这样的请求之前，这个子窗口不会出现，因此它没有出现在图1-1中。

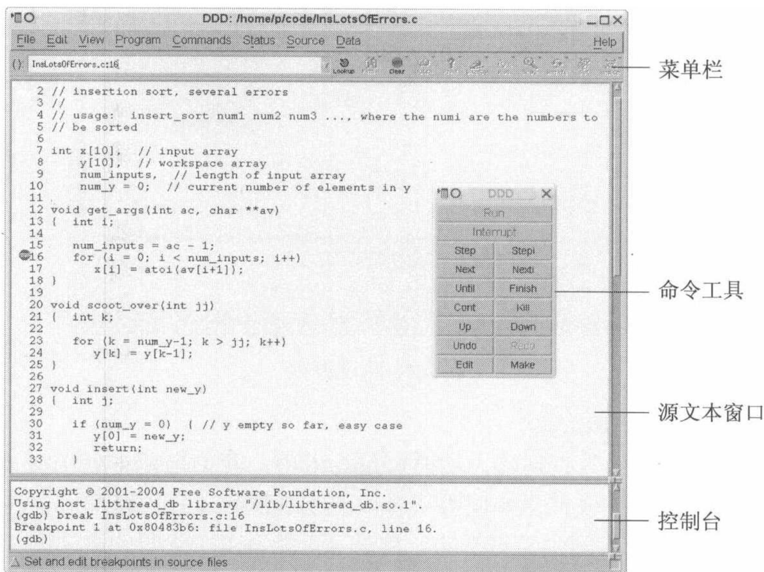


图1-1　DDD布局

下面是在每种类型的用户界面下如何将典型的调试命令提交给调试器的快速示例。在调试*insert_sort*程序时，你可能希望暂停该程序的执行，从而在函数`get_args()`的第16行（假设）处设置断点（在1.7节中将看到*insert_sort*的完整源代码）。为了在GDB中设置断点，可以在GDB提示符中输入如下命令：

```
(gdb) break 16
```

完整的命令名是break，但是GDB允许在不产生歧义的情况下使用缩写，并且大多数GDB用户会在此处输入b 16。为了帮助GDB的初学者理解这些缩写，我们将先使用完整的命令名，在本书的后面，当用户已经较为熟悉这些命令时改为使用缩写的命令名。

使用DDD，将查看源文本窗口，单击第16行的开始位置，然后单击DDD屏幕顶端的Break（中断）图标。也可以在第16行的开始位置右击，然后选择Set Breakpoint（设置断点）。另一种方法是在代码行开始位置左侧的任意位置双击该代码行。无论采用何种方法，DDD都会通过在该行中显示小的停止符号来确认选择，如图1-2所示。通过这种方式，可以快速浏览断点。

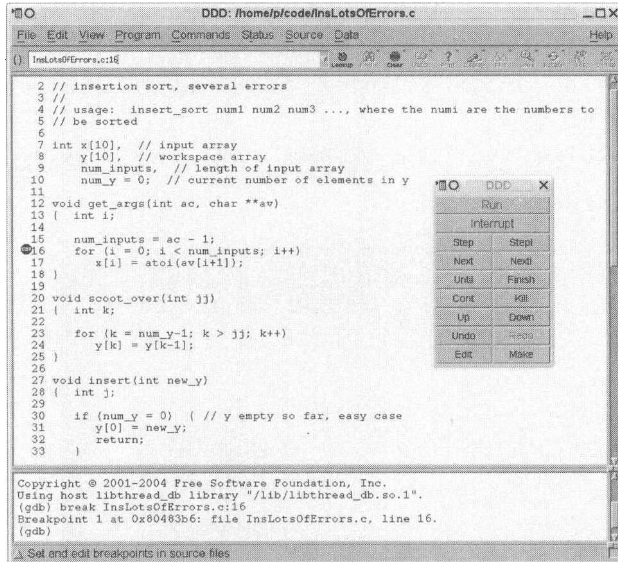


图1-2 断点设置

3. Eclipse：不只是GUI调试器

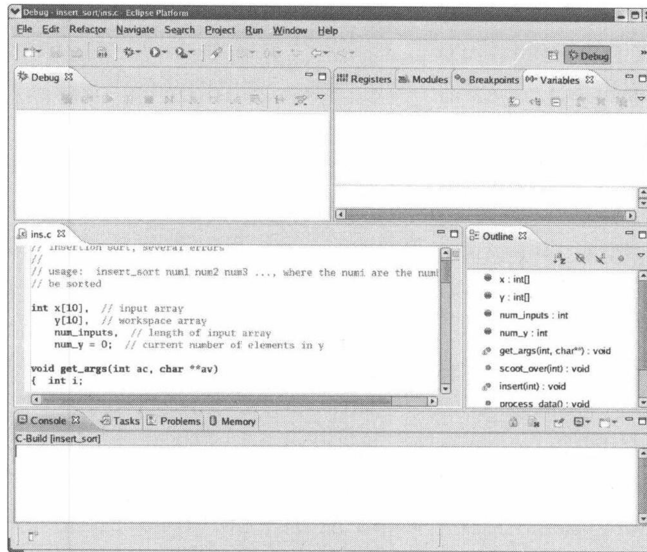
图1-3显示了Eclipse中的常规环境，按照Eclipse术语，我们当前正处于调试透视图（Debug Perspective）中。Eclipse是开发许多不同类型软件的常规框架。每种编程语言在Eclipse中都有自己的插件GUI，即透视图。实际上，对于同一种语言可能有多个竞争的透视图。在本书的Eclipse操作中，我们将使用用于C/C++开发的C/C++透视图、用于编写Python程序的Pydev透视图等。也有用于执行实际调试工作的调试透视图（带有一些语言特有的功能），图1-3就显示了该透视图。

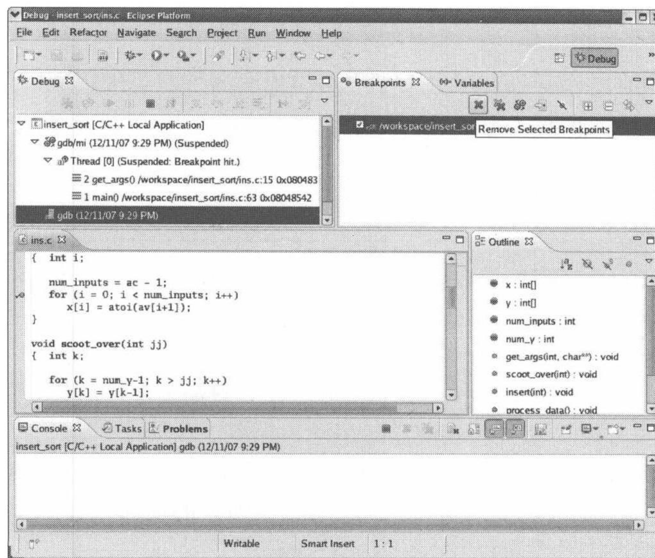
C/C++透视图是CDT插件的一部分。类似于DDD的情况，CDT在后台调用GDB。

透视图的相关组件一般类似于上面所描述的DDD的窗口组件。透视图被分解为多个称为视图的选项卡式窗口。位于透视图左侧的是源文件*ins.c*的视图，也有用于检查变量值的变量视图（到目前为止，该视图中还没有任何变量值），此外还有控制台视图，其功能非常类似于DDD中具有相同名称的子窗口，另外还有其他的视图。

可以按照与在DDD中相同的操作来可视化地设置断点。例如，在图1-4中，源文件窗口中的

代码行





试读结束：需要全本请在线购买： www.ertongbook.com