关于程序设计中 DEBUG 的概念及其使用的问题

耿楠

西北农林科技大学信息工程学院，陕西 · 杨凌，712100

2016 年 10 月 13 日

摘要

DEBUG 是一个用于调试程序的计算机程序，在程序设计与开发中，是一个重要的分析程序、检查错误、分析错误和解决错误的工具。本文说明了 C 语言程序设计中 DEBUG 的基本概念、分别在 Code::Blocks 中及命令行实现 DEBUG 的基本方法。另外，DEBUG 不只是 C 语言程序设计中需要的技术和技能，在后续汇编语言、Java、C# 等其它程序开发过程中也是必不可少的技术和技能，本文也将为后续的学习和工作提供必要支持。

一、DEBUG 的概念

1937 年，美国青年霍华德 · 艾肯为 IBM 公司投资了 200 万美元研制计算机，并将第一台成品取名为：马克 1 号 (Harvard Mark I)。为马克 1 号编制程序的葛丽丝 · 霍波 (Grace Hopper) 是一位美国海军准将及计算机科学家，同时也是世界最早的一批程序设计师之一，有一天，她在调试设备时出现故障，拆开继电器后，发现有只飞蛾被夹扁在触点中间，从而 “卡” 住了机器的运行。于是，霍波诙谐的把程序故障统称为 “臭虫 (BUG)”，把排除程序故障的过程叫 DEBUG。而这奇怪的 “称呼”，竟成为后来计算机领域的专业行话。通常，通过 DEBUG 可以:

1）监视 “DEBUG 对象”[1]的状态;

2）修改和控制 “DEBUG 对象” 的状态;

3）以字节为单位查看和修改内存中的任何内容;

4）逐指令执行 “DEBUG 对象”;

5）追踪 “DEBUG 对象” 的执行过程;

6）查看 CPU 工作状态。

这些工作可以为 **“发现 DEBUG 对象中存在的问题”** 以及 **“提出解决问题的方案”** 提供有用的信息。

实现 DEBUG 的工具通常称为：Debugger(调试器) ，这个词按它的英文字面意思来讲是有这样一种 “装置 (er)”，这种装置可以 “消除 (De)” 系统中的 “缺陷 (bug)”。不同的编译器往往提供有不同的 Debugger，例如，GNU 提供了 gdb 调试器，Visual Studio 提供了 cdb 调试器。不同的调试器的主要功能是一致的，但在细节上也有区别，其具体功能和操作细节需要查阅其相关资料。无论是哪种 Debugger，其执行 DEBUG 的过程基本是一致的：

[1]被调试的对象，也就是被调试的程序。

1）启动 Debugger 并载入“DEBUG 对象”；

2）设置断点；

3）执行 Debugger 各个命令；

4）查看和修改变量 (内存) 状态；

5）分析数据；

6）定位出错位置；

7）修改错误；

8）重复调试，直至正确。

不同的程序开发 IDE(集成开发环境) 也为 Debugger 提供了不同的操作方法，这些 IDE 为用户实现 DEBUG 操作提供了更为便捷的方式。但无论是哪种 IDE，其本质仍然是对 Debugger 各种命令的一个封装。本文分别通过 Code::Blocks 中及命令行进行 DEBUG 以说明 C 语言程序设计中实现 DEBUG 的基本方法。

二、在 Code::Blocks 中进行 DEBUG

Code::Blocks 是一款免费开源、功能强悍的 C/C++ 开发 IDE，该工具小巧灵活，支持跨平台、SVN 版本管理、代码语法高亮、自动格式化、国际化等功能。Code::Blocks 同样也集成了 DEBUG 操作，以方便程序的调试过程。

1、为 Code::Blocks 配置 Debugger

为使用 Code::Blocks 中集成的 DEBUG，首先必须确保 Code::Blocks 配置了 Debugger，为此，需要对 Code::Blocks 进行配置。

首先，使用如图 1所示的settings > Compiler...菜单打开 Compiler settings 编译器的全局设置对话框，如图 2所示。

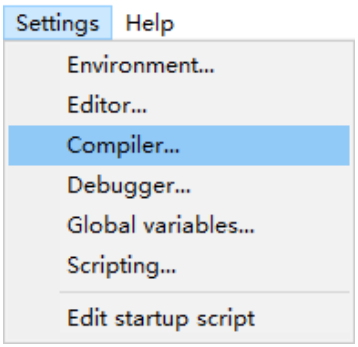

图 1　设置编译器菜单

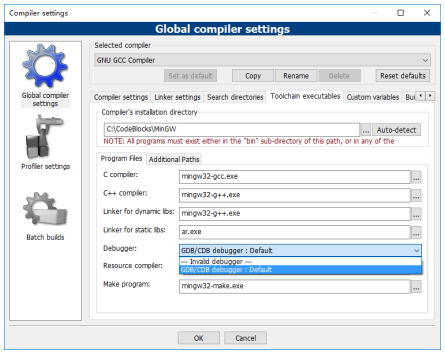

图 2　设置编译器参数

在此，将图 2的Toolchain executables标签中的 Debugger 设置为：“GDB/CDB debugger:Default”。

其次，使用如图 3所示的settings > Debugger...菜单打开 Debugger settings 编译器的全局设置对话框，如图 4所示。

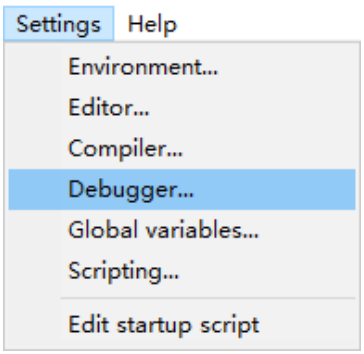

图 3　设置 Debugger 菜单

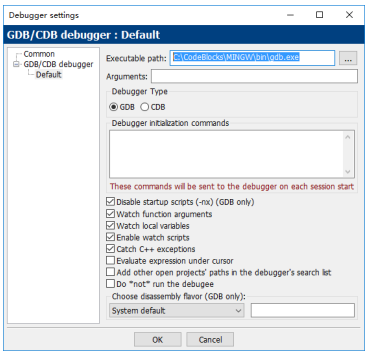

图 4　设置 Debugger 参数

选择图 4左侧“GDB/CDB debugger:Default”对其进行设置，注意选择“Executable path:”的**正确路径**，Debugger type 与“Executable path:”的**正确匹配**。

至此，便可完成 Code::Blocks 的 Debugger 设置，为后续的程序 DEBUG 提供正确的 Debugger 配置。

2、Code::Blocks 的 DEBUG 菜单与工具栏

在 Code::Blocks 中，与其它软件的基本操作类似，可以通过“Debug菜单”、“Debugger 工具栏”、“右键快捷菜单”、“快捷键”等方式执行 DEBUG 相关操作。

Debug菜单如图 5所示，其各菜单项的具体含义请读者参阅 Code::Blocks 的操作手册。

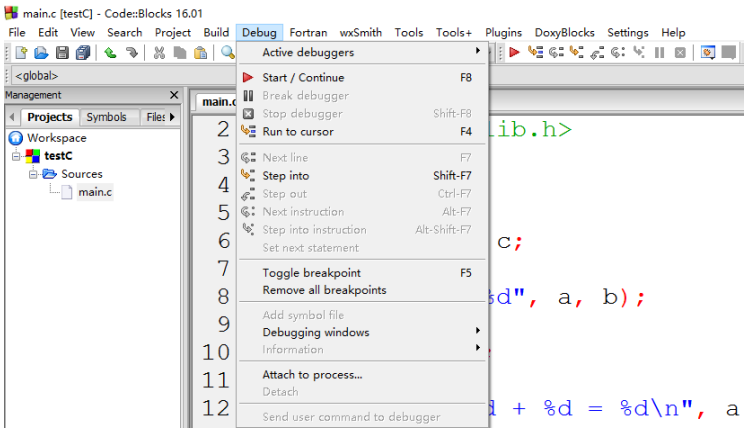

图 5　Debug菜单

Debugger 工具栏如图 6所示，从左到右，与 Debug菜单的各菜单项的对应关系分别是：Debug/Continue、Run to cursor、Next line、Step into、Step out、Next instruction、Step into instruction、Break debugger、Stop debugger、Debugging windows、Various info，具体含义请读者参阅 Code::Blocks 的操作手册。

图 6　Debugger 工具栏

3、添加程序运行断点

例如有如下程序代码：

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, c;

    scanf("%d%d", a, b);

    c = a + b;

    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

编译此代码，虽有警告，但无语法错误，可以运行。在程序运行时，当输入数据后，继续执行程序，则会出现如图 7所示的错误，程序出现崩溃。

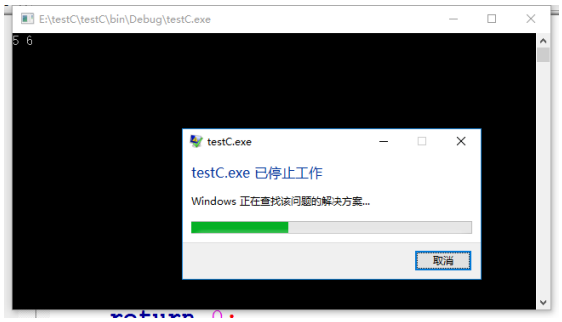

图 7　程序运行崩溃

类似这种编译中无错误，运行出错的现象一般称为“**逻辑错误**”，解决逻辑错误常用的技术便是“DEBUG”。

在 DEBUG 时，首先要做的是在程序的合适位置插入“断点”，以使程序执行到此**断点**时，程序可以**暂停**在断点处，以便查看程序执行的状态。

由图 7可以初步推断，可能在执行第 8 句 scanf() 的时候出错了，为此，可以第 8 句插入断点。

在编辑区，定位到程序第 8 句中，可以通过以下任何方式插入断点：

1）Debug > Toggle breakpoint 菜单，如图 5；

2）F5 快捷键；

3）在当前行右击的快捷菜单的 Toggle breakpoint，如 图 8；

4）在编辑区行号右侧的浅灰色区域单击鼠标左键 (红框标的区域)，如 图 9。

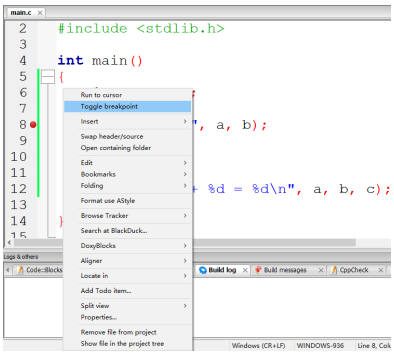

图 8　右击插入断点

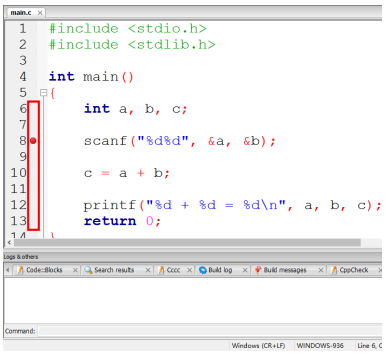

图 9　鼠标左击插入断点

注意：“Toggle breakpoint” 是一个**开关操作**，也就说，如果已经有断点时，执行 Toggle breakpoint 操作会删除该断点，如果此处没有断点，执行 Toggle breakpoint 操作会添加断点。

当插入一个断点后，在编辑区行号右侧的浅灰色区域与当前行对应的位置会有一个红色的圆点标志，在该圆点标志上右击，会出现图 10所示的断点编辑快捷菜单，可以对该断点进行需要的编辑，详情请参阅 Code::Blocks 使用说明书。

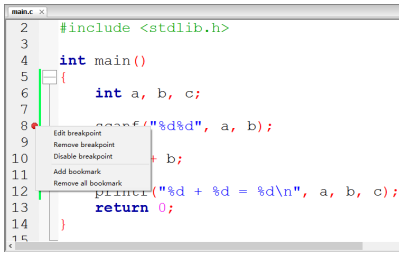

图 10　断点编辑菜单

在加入断点后，便可以进行程序的 DEBUG 过程了。

4、DEBUG 窗口

选择图 5中的Debug》Start/Continue菜单项，或是单击图 6工具栏中的红色右箭头按钮 (Debug/Continue)，便可以启动 Debugger 调试程序。

启动 Debugger 后，程序暂停在所设置的第 1 个断点处，并在断点上会叠加一个黄色的右箭头标志，以表示程序执行到了此处，如图 11所示。

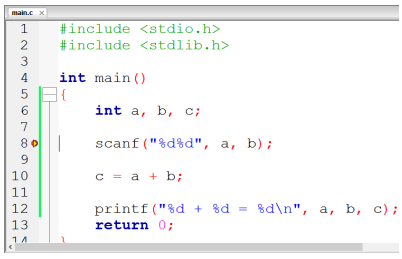

图 11　启动调试程序

程序暂停后，便可以查看 “DEBUG 对象”(被调试程序) 的当前运行状态，为此，需要打开相应的 “Debugging windows”。在此，可按图 12的菜单或图 13的工具栏按钮方式分别打开 “Memory” 窗口查看内存数据和 “Watchs” 窗口查看变量值。这些窗口可以如图 14所示，以浮动的方式布置在屏幕上任何位置，也可以如图 15所示，停靠在 Code::Blocks 的边栏中。

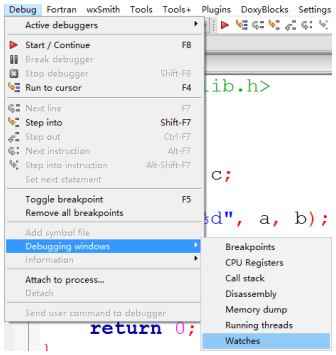

图 12　菜单操作

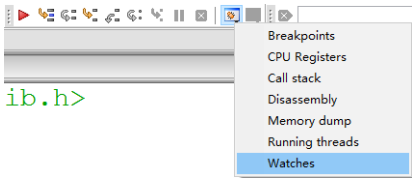

图 13　工具栏操作

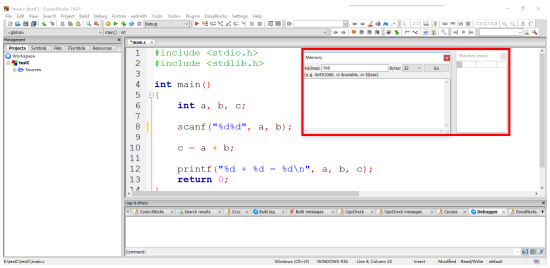

图 14　浮动窗口

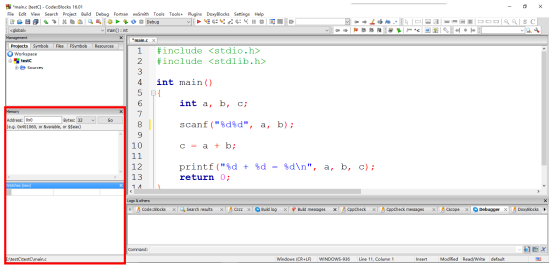

图 15　停靠窗口

5、查看程序运行状态

程序暂停后，通过图 16所示的 “Memory” 窗口可以查看内存中的数据，通过图 17所示的 “Watchs” 窗口可以查看程序中的各变量值。

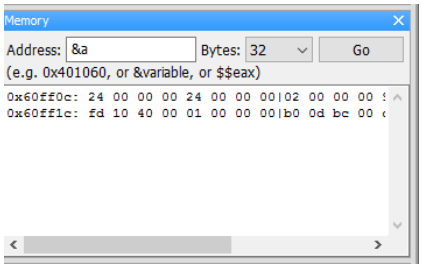

图 16　Memory 窗口

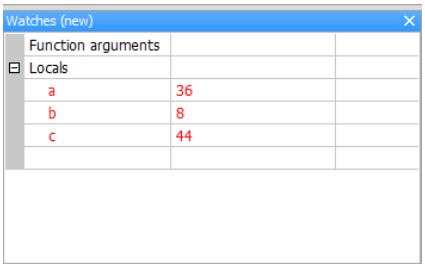

图 17　Watches 窗口

在此，需要注意的是 “Memory” 窗口应指定需要**显示内存区域的首地址**和需要**显示的字节数**。本例中用&a 取得变量 a 的地址，显示 32 个字节的内存数据。

另外需要**特别强调**的是：内存中的数据应该反向读，如 a 的类型是**int**类型的，占 4 个字节，则内存中 a 的数据是0X00 00 00 24(十六进制)。

当然，也可以使用 gdb 调试器的调试命令进行相应的操作，可以在 Logs&others 窗格的 Command: 命令行中执行这些命令，如图 18所示的“p[2]”命令，使用 p 命令也可以显示变量 a 的值和地址。

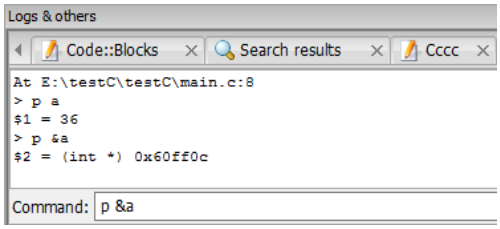

图 18 “p”命令显示变量值和地址

当然，还有更多的 gdb 调试器命令，使用这些命令可以跟踪程序运行过程，剖析程序运行机制。

6、单步执行程序

执行 Debug 》 Next line 或按 F7 快捷键或是单击如图 19所示的工具栏中的 Next line 按钮，便可以执行当前行程序代码。

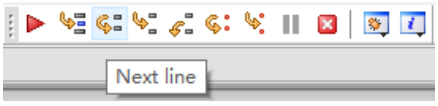

图 19 Next line 命令

观察此时代码的状态，如图 20所示，可以看到，当前第 8 行前的红色圆点断点标志上的黄色右箭头已消失，调试工具栏中除了暂停和停止按钮外，全为灰色不可用状态。

此时，程序执行了 scanf("%d%d", a, b);，这是一个输入操作，需要完成输入操作后程序才可以执行后续代码。在打开的如图 21所示的命令行窗口中，输入需要的数据。

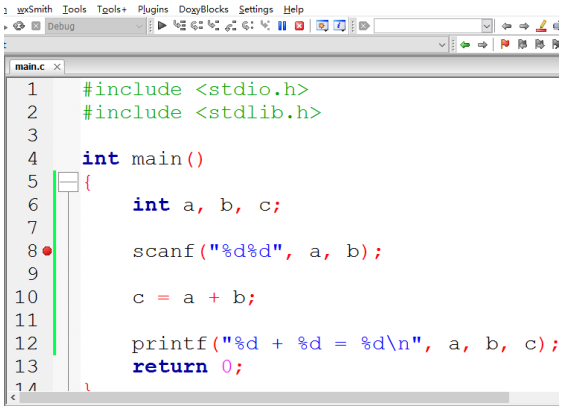

图 20 执行下一行代码

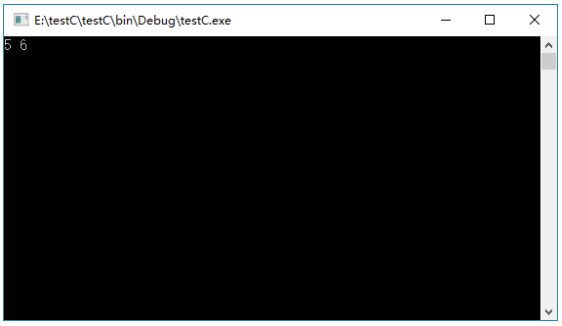

图 21 输入数据命令行窗口

在图 21命令行窗口中输入数据后，回车，此时会出现 图 22和图 23的错误结果。

至此，显然可以得出结论：**程序在执行到 scanf(...); 时，出现了错误**。

[2]也可以是“print”，“p”是其首字母缩写。

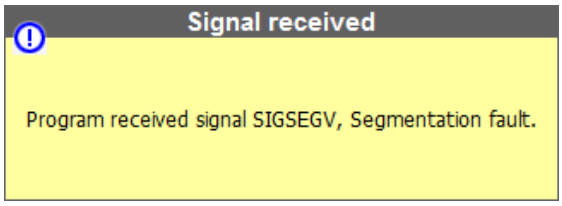

图 22　错误信息窗口

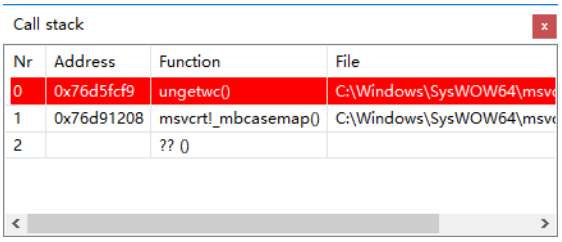

图 23　调用栈错误窗口

7、修改并继续调试程序

仔细分析第 8 行的代码，显然是由于 `scanf("%d%d", a, b);` 中的变量 a 和 b 前少了&的原因。完成代码修改后，继续调试程序，如图 24所示。

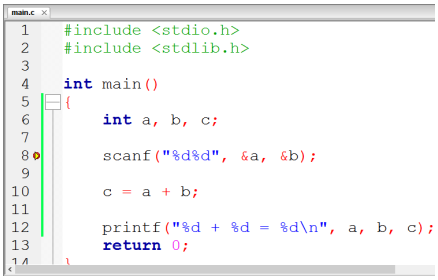

图 24　修改并调试的代码

此时程序暂停，通过图 25所示的“Memory”窗口可以查看内存中的数据，通过图 26所示的“Watchs”窗口可以查看程序中的各变量值。

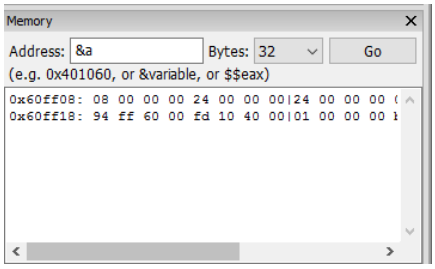

图 25　修改后的 Memory 窗口

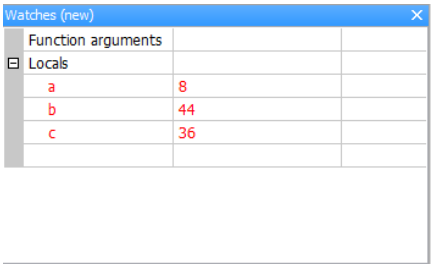

图 26　修改后的 Watches 窗口

值得注意的是：图 17与图 26中的值都是变量的初始值，但并不相同。**也就是说，未初始化的变量的值是不确定的**。

同样，也可以使用 gdb 调试器的调试命令进行相应的操作，如图 27所示。要强调的是：gdb 调试器的“p”命令不仅可以显示变量的值和地址，还可显示表达式的值，如 `p sizeof(a)` 显示表达式`sizeof(a)` 的结果 (**`sizeof()` 是运算符**)，当然也可以使用 `p a + b` 显示表达式 `a + b` 的结果。

在 gdb 调试器的命令行不仅可以用 `p` 命令显示程序运行中的各种状态，还可以使用其它命令实现对程序运行状态的控制，例如，可以使用“`set var`”命令改变变量当前的值，如图 28所示。

在程序后续代码中，将使用“`set var`”命令修改后的变量值，如图 29所示。

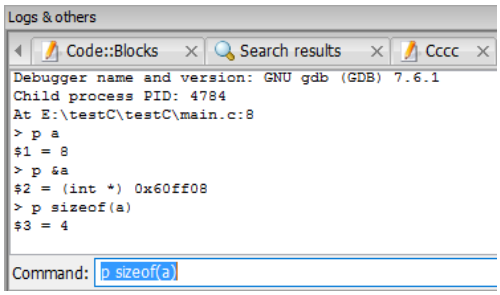

图 27 “p”命令显示表达式的值

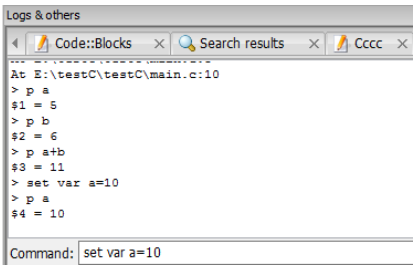

图 28 set var 的命令

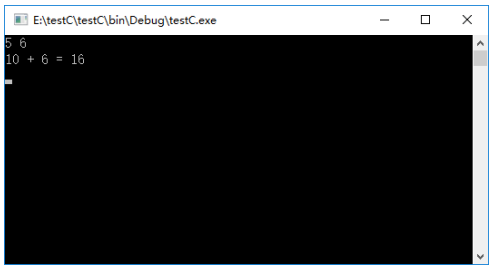

图 29 set var 的执行结果

8、结束程序调试

继续执行Next line，便可以跟踪和控制程序每一步运行状态，从而剖析“DEBUG 对象”，为**“发现 DEBUG 对象中存在的问题”**以及为**“提出解决问题的方案”**提供有用的信息。本例后续的调试结果如图 30和图 31所示。

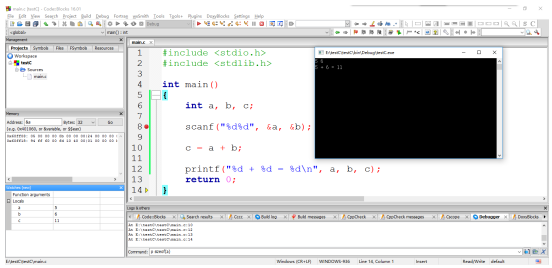

图 30 调试运行结果

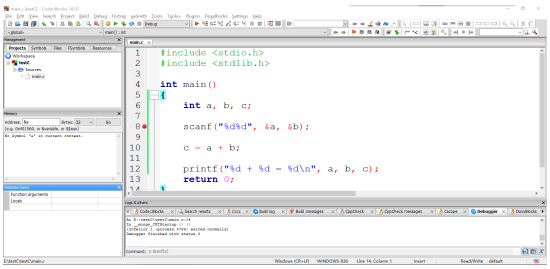

图 31 结束调试

9、调试操作失效的处理

程序设计过程中，难免会出现各式各样的错误，严重的情况下，**运行中的程序甚至不能够正常结束**，此时，调试工具栏会出现图 32所示的全为灰色的情况，无法进行调试操作。

图 32 禁用的 DEBUG 工具栏

在出现无法正常结束程序时，可以打开“管理器”，找到当前程序的进程，强行结束该程序进程，便可以执行后续的调试工作，如图 33所示。

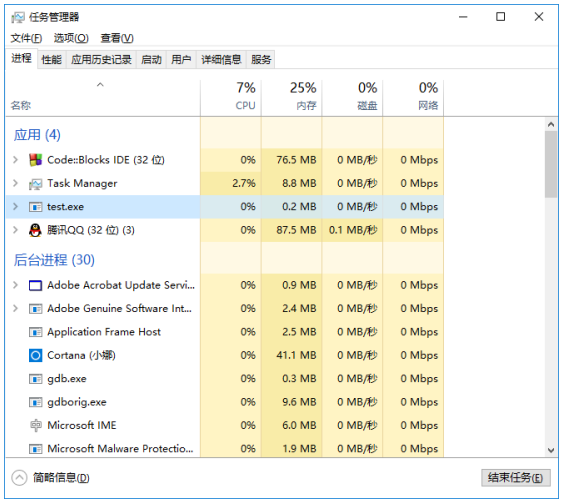

图 33　结束进程

三、在命令行 DEBUG 程序

虽然可以在 Code::Blocks 中借助各种可视化的工具进行程序的 DEBUG，但 Code::Blocks 实际上是对 gdb 调试器的各种调试命令进行了封装，不使用 Code::Blocks，直接在命令行使用 gdb 调试器的调试命令对程序进行调试也是一种常见的操作。

命令行窗口可以通过开始菜单中启动，或在 Win7 中用 ctrl + R 后输入 cmd 命令启动，或 Win10 中在小娜窗格中输入 cmd 命令启动 (命令行窗口俗称 “**黑窗口**”)。

1、在命令行编译链接程序

C 语言程序代码也可以用 gcc 命令在命令行进行编译和链接：

在命令行窗口中执行 “gcc -Wall -g -c addition.c -o addition.o” 命令可执行源代码的编译 (图 34)，执行 “gcc -o addition.exe addition.o” 命令可执行目标程序的链接 (图 35)[3]。

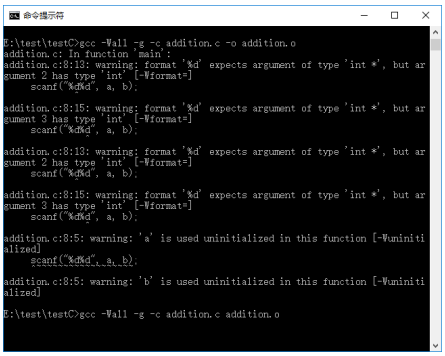

图 34　命令行编译程序

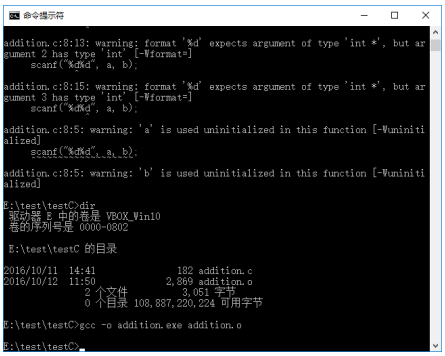

图 35　命令行链接程序

由图 34可以看出，编译中有警告信息，表明该程序可能会有错，但不存在语法错误，程序可以链接并执行，链接后可生成 “addition.exe” 可执行程序。执行 addition.exe，则会出现程序崩溃，如图 36所示。

[3]如果出现 “gcc **不是内部或外部命令，也不是可执行的程序或批处理文件**” 的错误，请检查环境变量的配置。

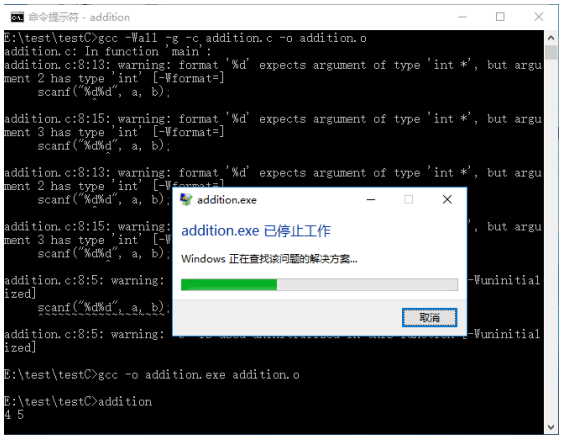

图 36　程序崩溃

2、在命令行启动 gdb 调试器调试程序

由于在编译中使用了-g 参数，因此，该可执行程序中具有调试信息，可以用 gdb 调试器进行 DEBUG 调试。

在命令行启动调试的命令是：“gdb addition[4]”命令，启动 gdb 调试器后的结果如图 37所示，其中，行首的“(gdb)”表示现在处理 gdb 调试器调试状态。

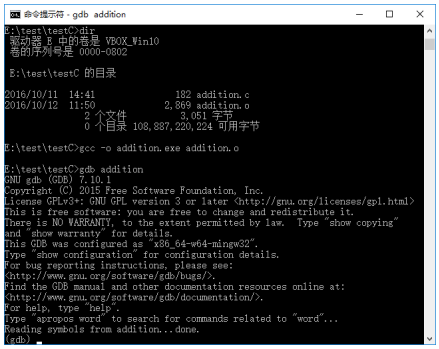

图 37　启动 gdb 调试器

在（gdb）后输入“l[5]”，便可以显示对应的源代码，默认每次显示 10 行代码，然后直接在（gdb）后输入回车表示重复上一次命令，代码显示如图 38所示。

使用“break 行号”，便可以在指定的行添加**断点**，使用“info break”，则可以显示已有**断点的状态**，如图 39所示。

使用“r[6]”，可以启动运行被调试程序。使用“p”命令可以显示代码中变量的值、变量的地址或是表达式的值。使用“x 地址”可以显示指定内存中的值，如图 40所示示 (**这些命令的细节，请参阅 gdb 调试器的相关资料，下同**)。

使用“n[7]”，可以执行下一行代码，如图 41所。

[4]可省略后缀名“.exe”。

[5]也可以是“list”，“l”是其首字母缩写。

[6]也可以是“run”，“r”是其首字母缩写。

[7]也可以是“next line”，“n”是其首字母缩写。

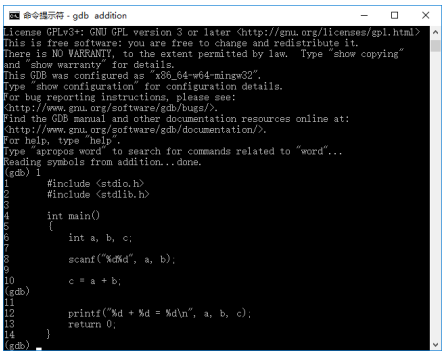

图 38　显示源代码命令

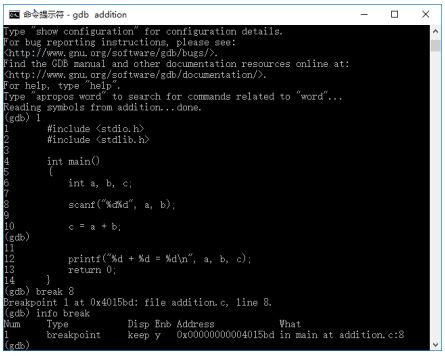

图 39　“break” 和 “info break” 命令

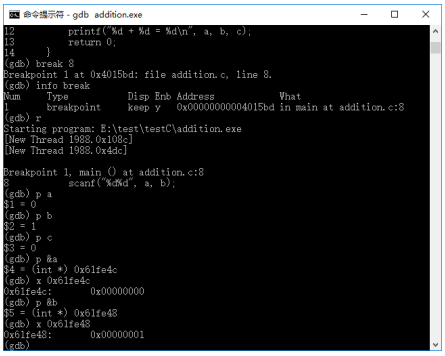

图 40　“r”、“p” 和 “x” 命令

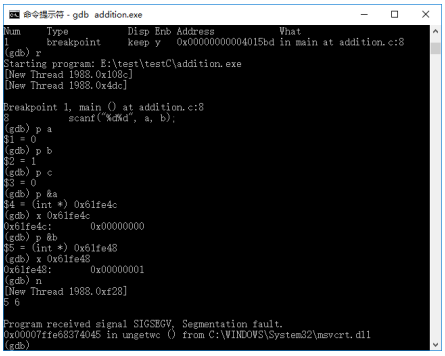

图 41　“n” 命令

在图 41，用 “n” 命令执行的是 `scanf(...)` 这一行代码，输入`5 6`，然后回车，显然，此时程序出错，因此，可以定位程序崩溃的原因就是这一句话。

执行 “q[8]” 便可以退出 gdb 调试器，如图 42所示。

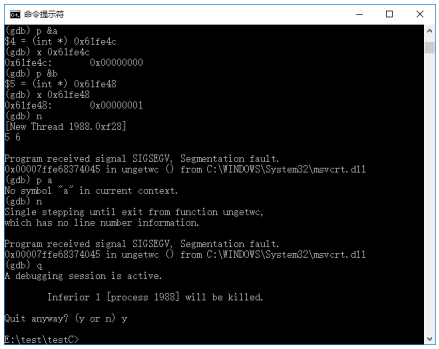

图 42　“q” 命令

修改代码，将第 8 句代码改为 `scanf("%d%d", &a, &b);`，继续调试程序。可以执行 “p” 命令显示表达式的值，也可以执行 “set var” 命令改变变量的值，调试结果分别如 图 43和图 44所示。

[8]也可以是 “quit”，“q” 是其首字母缩写。

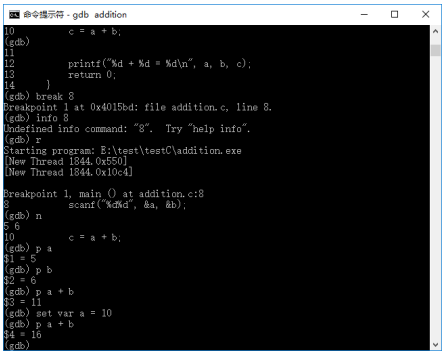

图 43 “p”和“set var”命令

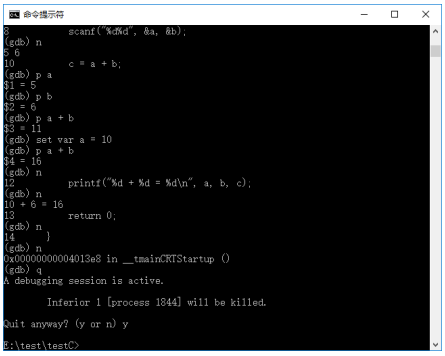

图 44 完成调试和纠错

四、结论

综上所述，使用 Debugger 进行程序的调试，在程序设计与开发中，是一个重要的分析程序、检查错误、分析错误和解决错误的 DEBUG 过程。DEBUG 可以在 Code::Blocks 中以可视化的方式实现，也可以在命令行窗口通过 gdb 调试器的调试命令实现。DEBUG 不只是 C 语言程序设计中需要的技术和技能，在后续汇编语言、Java、C# 等其它程序开发过程中也是必不可少的技术和技能，本文也将为后续的学习和工作提供必要支持。

在此，借用高适的《别董大》和李清照的《如梦令》填词一首，以期大家能领悟“**大学就是不断发现 BUG、解决 BUG，无穷无尽 DEBUG 的过程**”。

莫愁前路无知己，
总有 BUG 跟着你。
DEBUG，无尽头，
误入代码深处，
单步，单步，发现 BUG 无数。