



C 语言调试技巧

pluskid@MSTC

BUG 与 DEBUG

Bugs are by far the largest and most successful class of entity, with nearly a million known species. In this respect they outnumber all the other known creatures about four to one.

-Professor Snope's Encyclopedia of Animal Life

BUG 一词的由来

早在爱迪生时代（1878），Bug 已经是一个通用的“行话”了，专指意想不到的系统错误。

1947 年 9 月 9 日，一个工程师从 Harvard Mark II 计算机中拿出了一只真实的虫子——飞蛾，并把它贴到运行日志上，旁边写下注释：“1545 继电器#70 面板 F（飞蛾）在继电器内。发现了第一支真正的虫子。”






9/9

0800 Antan started
 1000 " stopped - antan ✓
 1300 (032) MP-MC ~~1.482160000~~ { 1.2700 9.037 847 025
 (033) PRO 2 ~~2.130476415~~ 9.037 846 895 consist
 consist 2.130476415 4.615925059(-2)
 2.130676415
 Relays 6-2 in 033 failed special speed test
 in relay .. 11.00 test.

1100 Started Cosine Tape (Sine check)
 1525 Started Multi Adder Test.
 Relays changed

1545  Relay #70 Panel F
 (moth) in relay.

1630 Antan started.
 1700 closed down.

First actual case of bug being found.

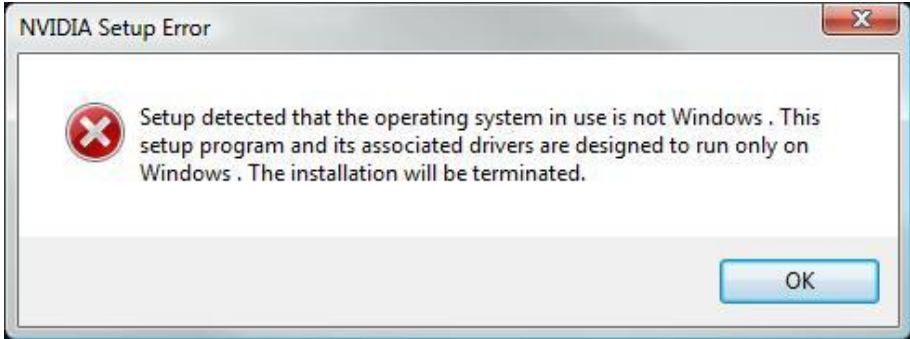
Relay 3145
 Relay 3376

有趣的 BUG

```
printf("Welcome back: %s\n",
      name);
return 0;
```

name | 0x001cfd4





更多有趣的 bug 请参考 The Daily WTF(Worth Than Failure):

<http://thedailywtf.com/>

代价高昂的 BUG

- 1985-1987 – Therac-25 medical accelerator: 医疗设备中的软件错误，造成至少 5 人死亡，多人重伤。





- 1996 – Ariane 5 火箭(十亿美元), 由于导航系统的 bug, 在启动后不到一分钟之内被销毁。
- 1962 – Marriner I 空间探测飞船, 由于程序员对一个手写的公式理解错误, 导致程序 bug, 火箭偏离轨道, 最后在大西洋上空被销毁。
-

参考资料

- Collection of Software Bugs:
<http://www5.in.tum.de/~huckle/bugse.html>
- Software Horror Stories:
<http://www.cs.tau.ac.il/~nachumd/verify/horror.html>

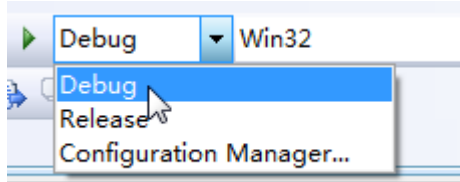
C 语言调试技术

VISUAL C++ 调试简介





Visual C++ Express(目前最新稳定版本是 2008)是由 Microsoft 专门提供给学生及非商业用户的一个轻便的集成开发环境,可以免费下载使用,学习 C/C++ 语言推荐使用的平台。



新建项目

1. File→New→Project (Ctrl+Shift+N)
2. Win32→Win32 Console Application→Enter *Project Name*→OK
3. Application Settings→check *Empty Project*→Finish

启动调试


1. 选择 Debug 配置
2. Build→Build Solution (F7)
3. Debug→Start Debugging (F5)

设置断点

有许多方法可以在代码的某一行设置断点:





- Debug→Toggle Breakpoint
- 快捷键 F9
- 直接点击编辑器左侧的灰色区域，再次点击可以删除断点，右键可以得到更多断点操作选项：
 - 禁用断点：暂时禁用，但不删除。
 - **when hit**：可以设置当断点触发时自动执行一些简单的动作，例如打印出一句话。
 - **hit count**：可设置简单的断点条件，仅当断点触发数目满足要求时中断程序运行。
 - **condition**：可设置复杂条件，仅当条件满足时终端程序运行，例如：
`a==2043 && b==2044`。

```
size_t pos;  
if (a != t  
    tree[;  
    tree[;
```

调试程序

控制程序的运行

启动调试之后，工具栏上会出现一些控制程序执行流程相关的按钮，这些指令也可以在 **Debug** 菜单下找到，也可以使用相应的快捷键调用。执行过程中在编辑窗口左侧的灰色部分会有一个黄色的小箭头表示目前程序执行到了代码的哪一行。





- **Stop Debugging, Restart Debugging:** 顾名思义
- **Break All:** 可以手动暂停程序
- **Continue:** 继续执行程序，直到下一个断点或者程序退出
- **单步执行:** 一条语句一条语句地执行
 - **Step into:** 如果遇到函数调用则进入函数体
 - **Step over:** 不跟踪进入函数体，把函数调用当作一条完整的语句执行
 - **Step out:** 执行到跳出当前函数体为止

观察程序的状态

启动调试之后，在主窗口下方会出现一些显示程序当前执行状态的窗口，分列在各个 tab 里，如果没有显示，可以通过 **Debug→Windows** 里的菜单项令其显示出来，几个常用的信息窗口如下：

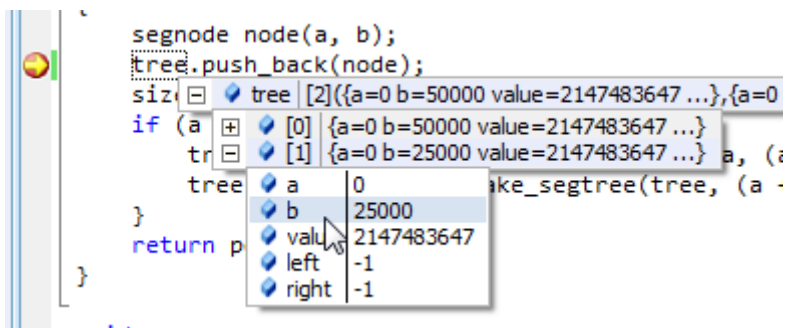
- **Locals:** 显示当前函数的局部变量值。
- **Call Stack:** 当前的调用栈信息，可以看到函数的调用关系。
- **Breakpoints:** 查看断点的信息。
- **Output:** 程序本身的输出会显示在这里。





- **Watch:** 可以输入任意表达式查看对应的值，几乎所有合法的 C 表达式都可以，甚至可以调用函数，但是要注意最好不要有副作用。在执行过程中发生了变化的值会以红色高亮显示出来。
- **Immediate Window:** 可以临时输入一个表达式来查看它的值，但是不会像 Watch 窗口那样在程序执行过程中一直保留着这个表达式。

另外，把鼠标放在代码里的变量名上，也可以直接查看变量的内容：



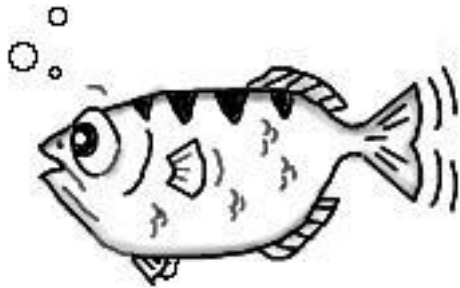
GDB 调试简介

gdb 是由 GNU 软件系统社区提供的调试器，同 gcc 配套组成了一套完整的开发环境，可移植性很好，支持非常多的体系结构并被移植到各种系统中（包括各种类 Unix 系统与 Windows 系统里的 MinGW 和 Cygwin 移植。此外，除了 C 语言之外，gcc/gdb 还支持包括 C++、Objective-C、Ada 和 Pascal 等各种语言后端的编译和调试。gcc/gdb 是 Linux 和许多类 Unix 系统中的标准开发环境，Linux 内核也是专门针对 gcc 进行编码的。





同 Visual Studio 不一样的是，gcc、gdb 并不是一个集成开发环境，而是提供了一系列命令行工具，可以通过一些编辑器（例如 Emacs）或者通用的 IDE（例如 Eclipse）以及专用的 GUI 前端（例如 ddd）来配套使用，也可以直接用命令行接口进行操作。



gdb 的吉祥物是专门捕杀 bug 的射手鱼：

For a fish, the archer fish is known to shoot down bugs from low hanging plants by spitting water at them.

为调试做准备

通常，在为调试而编译时，我们会（在尽量不影响程序行为的情况下）关掉编译器的优化选项(-O)，并打开调试选项(-g)。另外，-Wall 选项打开所有 warning，也可以发现许多问题，避免一些不必要的 bug：

```
gcc -g -Wall program.c -o program
```

编译好之后用 gdb 启动要调试的程序：

```
gdb program
```





之后会出现 `gdb` 的 `shell`，输入

```
run args
```

即可启动程序，`args` 是传递给程序的命令行参数。当然，在启动之前，通常会先设置断点，并熟悉一下相关的命令。

在 GDB SHELL 中调试

在 `gdb` 中最最重要的一个命令是 `help`，直接输入 `help` 即可查看帮助文档，`help` 后面跟某个具体的命令就可以得到该命令的使用说明。另外，在 `gdb shell` 中有一些快捷键可以使用（和通常的 `Linux shell` 是差不多的，因为都是使用 `GNU readline` 来实现的），部分列举如下：

- `Tab`：可以对命令或参数进行补全
- `Ctrl+P`：历史中的上一条命令
- `Ctrl+N`：历史中下一条命令
- `Ctrl+A`, `Ctrl+E`：跳到行首、行尾
- 输入命令后按回车键执行命令，如果不输入任何命令直接回车则重复前一个命令。





常用的命令通常会有缩写，例如 `backtrace` 命令可以用缩写 `bt` 代替，另外，在没有歧义的情况下，也可以只写一个前缀，例如，对于 `help` 命令来说，输入 `h`、`he` 或者 `hel` 都是可以的。

以下列出一些常用的命令（具体语法请参见帮助文档）：

- `quit`: 退出调试器
- `run`: 启动程序，可以给程序传递命令行参数，也支持重定向操作（两年前一位朋友告诉我在 `cygwin` 下的 `gdb` 里对重定向的支持有 `bug`，不知道现在这个问题修复了没有）。
- `break`: 设置断点，可以在某一个函数、源代码的某一行或者是某一个内存地址设置断点，还可以为断点指定一个条件。例如，`b main` 会在 `main` 函数设置一个断点。
- `condition`: 为断点指定条件。例如，`cond 1 argc==1` 为编号为 1 的断点设置条件 `argc==1`。
- `info`: `info` 是一系列命令，其中最常用的是 `info breakpoints`，可以列出现有的断点，在这里可以查到各个断点的编号和条件。
- `next`: 单步执行，对应 Visual Studio 的 Step over。
- `step`: 单步进入，对应 Visual Studio 的 Step into。
- `finish`: 执行到当前函数退出，对应 Visual Studio 的 Step out。





- **where:** 和 **backtrace** 是同一个命令，查看调用栈以及当前位置。
- **list:** 可以显示源代码。单步执行的时候 **gdb** 会自动显示下一行将要执行的代码，**list** 命令可以显示某一行代码前后 **N** 行的内容。
- **print:** 可以查看某一个表达式的值，相当于 **Visual Studio** 的 **Immediate Window**。
- **display:** 类似与 **Visual Studio** 的 **Watch** 窗口，在单步执行的时候每一步都会重新打印一下表达式的值。
- **watch:** 设置观察点，这里的 **watch** 和 **Visual Studio** 里的那个 **Watch** 窗口是不一样的，一个 **watchpoint** 相当于一个断点，不同的是它是设置在一个变量（或内存地址）上，当该变量的值改变是程序会像触发了断点一样被中断下来。

一个典型的 **gdb** 调试的过程是启动程序，**run** 运行，程序出错退出，用 **where** 查到出错的位置，设置断点，重新运行程序，然后在出错的附件使用 **next** 和 **step** 单步跟踪，并查出问题所在。

在 EMACS 中使用 GDB

直接使用 **gdb** 有一些不太方便的地方，比如查看源代码很麻烦，而且断点的设置也有点痛苦（如果要具体到某一行的话，需要手工输入文件名、行号）。**Emacs** 是一个功能齐全，可扩展性很好的编辑器，它也对 **gdb** 有比较好的支持，可以





当作一个轻量级的 `gdb` 前端使用。这里简要介绍一下在 `Emacs` 里使用 `gdb` 的方法，有 `Emacs` 恐惧症者请自行跳过。--

在 `Emacs` 使用 `M-x gdb` 即可启动 `gdb` 调试器，启动之后工具栏会发生变化，并且菜单栏会出现相应的菜单项，与 `Visual Studio` 类似。一个典型的 `Emacs` 调试窗口如下图所示：

```
emacs@kid-cad
File Edit Options Buffers Tools Gud Complete In/Out Signals Help

p p* [gdb icons]

▲ Breakpoint 1 at 0x400651: file foo.c, line 40.
(gdb) r
Starting program: /tmp/foo
(gdb) n
(gdb)

-U:**- *gud-foo* Bot L17 (Debugger:run [end-stepping-range])-----
Breakpoints Threads
Num Type Disp Enb Addr Hits What
1 breakpoint keep y 0x000000000400651 1 in main of foo.c:40
-U:%*- *breakpoints of foo* All L2 (Breakpoints)-----

int main(int argc, char **argv) {
  Node *list = NULL;
  int i;
  for (i = 0; i < 5; ++i) {
    Node *new_node = (Node *)malloc(sizeof(Node));
    new_node->value = i;
  }
}

foo.c 62% L44 (C/l Abbrev)-----
```

虽然如此，通常调试时还是直接在图中的 `gdb shell` 中输入 `gdb` 命令，而不是使用工具栏。比直接使用 `gdb shell` 更好的地方在于现在可以直接同步地看到带语法高亮的正在执行的源代码（黑色的小箭头），并且可以像 `Visual Studio` 那





样通过直接在代码左边点击来添加和删除断点（如果是在不支持鼠标操作的终端中，可以使用 **C-x C-a C-b** 和 **C-x C-a C-d** 来做同样的操作）。

高级调试技术

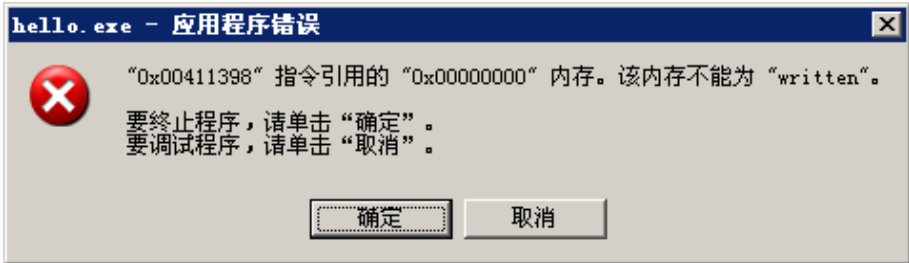
内存 BUG 调试

内存 Bug 简介

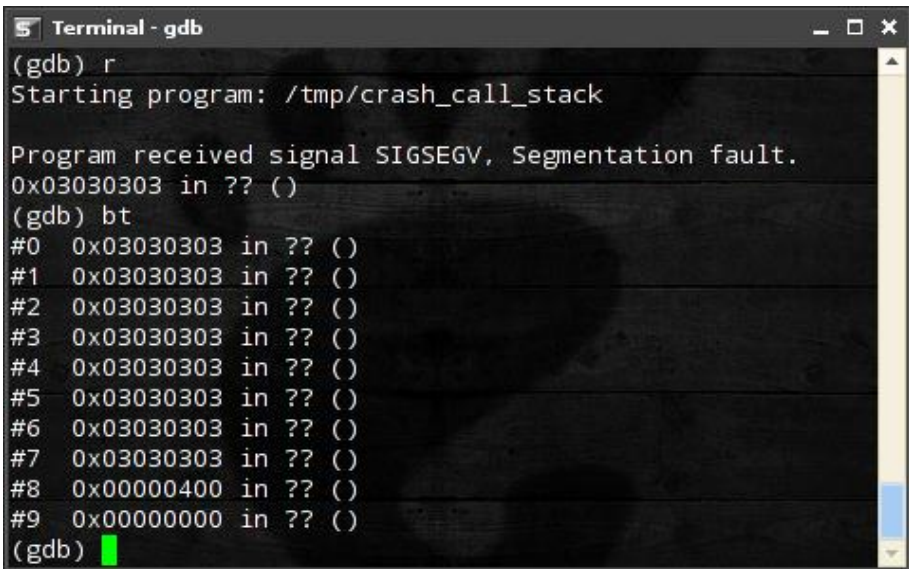
C 语言的其中一个强大的地方就在于可以直接对内存进行许多底层的操作，同时这个特性也成为了一个相当繁荣的 **bug** 家族——内存 **bug** 滋生的温床。常见的内存 **bug** 通常包括：

- ✧ 内存泄漏：这是需要长期运行的服务程序的最致命的杀手之一。
- ✧ 缓冲区溢出：众所周知的安全隐患，非常大一部分系统安全问题（包括入侵、感染等）都是由缓冲区溢出引起的。结果现在 Visual C++ 编译器对 C 标准库里的诸如 `gets`、`strcpy` 等函数一律报 **C4996** 警告，可见微软对缓冲区溢出的问题也是咬牙切齿啊。
- ✧ 野指针，或者是非法内存访问。是另一类非常常见的错误，如果运气好，被操作系统及时捕捉到了异常行为，则情况还比较乐观（例如示例代码 **01**）：在 Linux 下通常会表现为程序出现 **segmentation fault**(段错误)而退出，而 Windows 下则通常会弹出一个类似这样的对话框：





如果运气不好，则有可能将整个调用栈信息全部损坏掉，bug 与程序同归于尽，除非调试器保护现场的能力非常强悍，否则估计连骨灰都找不到了，就像下面这个样子：





- ◇ 内存未初始化，相比它的前面几位兄弟，这个 **bug** 似乎名气要小一点，但是绝对也是无孔不入，不太出名是因为它通常没有自己独特的表现形式，但是他善于制造混乱，产生各种其他的诡异问题，并且由于未初始化的内容通常是随机的，因此在不同的时间、地点运行有可能会产生不同的结果，不易重现，很难调试。

内存调试工具 **valgrind**



Valgrind 是一套强大的内存调试和剖析(**profiling**)工具，内存调试器能够帮助我们查出许多普通调试器很难找到的 **bug**，包括：





- ◇ 内存泄漏
- ◇ 读写已经释放过的内存
- ◇ 多次释放同一块内存或者试图释放压根没有申请过的内存
- ◇ 数组越界
- ◇ 试图访问未分配的内存
- ◇ 读取未初始化的内存
- ◇ 空指针读写

valgrind 使用非常简单，首先用 gcc 带-g 选项对程序进行编译，然后运行：

```
valgrind --tool=memcheck --leak-check=yes ./program
```

valgrind 会运行程序，并把生成的报告打印到终端上，输出类似于这样：

```
==19389== Invalid write of size 1
==19389==    at 0x4C22F0F: strcpy (in ...)
==19389==    by 0x400692: copy_string (buggy.c:11)
==19389==    by 0x4006B6: main (buggy.c:16)
==19389== Address 0x5179053 is 0 bytes after a block of size
19 alloc'd
==19389==    at 0x4C21E03: malloc (in ...)
==19389==    by 0x400663: copy_string (buggy.c:6)
```





```
==19389== by 0x4006B6: main (buggy.c:16)
```

前面的 19389 是程序的进程标识符(Pid)，在同时跟踪多个程序时可以用于区分，这里可以忽略。输出的内容非常详细，可读性很好，这里的输出大致是说分配了 19 字节的数据，但是却多写了一个字节。同时还给出了出错的源文件以及行号，一般定位之后就很容易查出问题了。

其他的一些输出就不一一列在这里了，一般都能看得懂，另外，valgrind 的一些高级用法以及如何用来做 profiling，可以参考官方文档。

GDB 高级功能

gdb 有一些很强大的高级功能，特别是在新发布的 7.0 里加入了逆向调试和 Python 脚本的支持。

逆向调试，顾名思义就是逆向地执行程序，相关指令包括 reverse-continue、reverse-next 和 reverse-step 等。逆向执行实际并不是在“执行”程序，而是把程序“回滚”到之前的状态，考虑最简单的情况，一个变量赋值，要逆向执行，就是要恢复其被赋值之前的值，如果对于整个程序而言的话，那么为了能够回滚，程序每执行一步的所有状态都需要记录下来，即便是通过一些增量式的存储技术，也是相当大的数据量，而且程序的执行速度可能会被进一步拖慢。

逆向调试在某些情况下也许会比较有用，比如一个很难跟踪的 bug，单步执行的时候不小心多按了一下 step，这个时候就希望能够 reverse-step 一下，否则就又要从头跟踪一遍了。另一方面，逆向调试并不是万金油（且不说目前 gdb





的这个技术还处在比较初步的阶段)，设想，如果一条程序语句的功能是发送一封 email，那 `reverse-next` 一下难道还能把它再撤回来吗？

关于 Python 脚本支持，指的是可以在 `gdb` 里写一些 Python 脚本来控制调试过程，例如，考虑一个复杂的断点条件“某个变量等于特定值并且栈的前一帧的函数名是 `foobar`”之类的，没法用 C 语言表达式表达出来的，就可以写一个 Python 函数来测试断点条件是否满足。

不过，这些浮云的功能在平时的调试中通常并不会用得特别多，因此并不是我们这次的重点，感兴趣的同学可以事后自行查阅相关文档。

无调试器调试

调试器的出现固然极大地改善了可怜的程序员们的生活水平，然而调试器也并不总是扮演救世主的角色，例如，在有复杂竞争条件的多线程程序或者分布式程序中，调试器所能起的作用通常都不大。另外，调试运行和正常运行的程序实际上是有一定的差异的，有些神奇的 `bug`，当你以正常方式运行程序时，它跑出来作威作福，可以当你以调试模式运行程序的时候，它就躲得无影无踪了。更为极端的情况是没有调试器可以用，如果 `gdb` 的开发人员需要用 `gdb` 来调试 `gdb` 都还可以接受的话，那么 Linux kernel 的开发人员就真的是悲剧了。

因此，很多时候，我们需要在没有调试器的情况下进行调试，幸运的是，在这样的情况下，也是有一些约定的方法可以遵循的。





core dump

在 Linux 下，程序如果出现段错误退出，会产生 **core dump** 文件，默认情况下被 **ulimit** 禁用了这个功能，运行下面这个命令：

```
ulimit -c 5000
```

将允许系统产生 **5kB** 以内的 **core dump** 文件，可以根据自己的需求调整大小，并写到 **shell** 的启动脚本里。系统生成的 **core dump** 文件通常就是叫做 **core**，包含了程序出错时的整个状态，用 **gdb** 加载 **core** 文件：

```
gdb program core
```

就可以进行一些事后分析，例如，可以通过 **backtrace** 命令查出出错时的调用栈，并查看一些变量的值等，通常对于定位 **bug** 有很大的帮助。

printf 调试

printf 调试泛指通过记录程序执行状态来做调试的方法。具体来说，通常我们对于程序的行为和状态都有一些期望的值，通过将程序运行时的实际值打印出来，与期望的情况进行对比，就可以逐渐找到问题的所在。

然而这种方法操作起来却有一些相当繁琐的地方：

✧ 首先，由于不知道问题出在哪里，又不能把所有的地方都添加输出语句（输出信息太多的话，要找到问题就变得困难了），所以通常会在可疑的地方





添加输出语句，如果结果发现猜错了，就需要换一个地方或者扩大范围，修改代码，重新编译，运行，再查看新的输出结果。对于编译时间很长（例如，有很多模版代码的 C++ 程序）的情况，整个过程会变得相当痛苦，因为可能需要重复很多次，并且许多时间都是在做无聊的等待。

- ◇ 其次，如果找到了问题所在，是不是要删除那些状态输出语句呢？过多的输出是会影响程序运行性能的（例如水寒自己写的一个 HTTP proxy，后来发现性能瓶颈居然在于记录了太多的日志信息），特别是打印到终端上。这些输出语句可能遍布代码的各个角落，要全部清除也不容易，而且，万一以后遇到了类似的 bug 呢？可能还要再写一遍这些类似的输出语句。另一个选择就是把他们注释掉。但是，无论如何，代码会被改得越来越乱。

避免让代码变乱的解决方案是使用标准化的工具，例如，最简单的情况，可以使用下面这样的宏：

```
#ifdef DEBUG
#define LOG(args) printf args
#else
#define LOG(args) ((void) 0)
#endif /* DEBUG */
```

需要记录信息的时候，使用

```
LOG(("a = %d, b = %d\n", a, b));
```





就可以了（注意双重括号是必要的），需要调试的时候，只要定义 `DEBUG`，就可以得到调试输出，而调试结束之后可以直接去掉 `DEBUG` 的定义，这样 `LOG` 宏在编译的时候就会变成空语句，也就不会产生任何输出了。即使想要移除这些调试语句，由于它们都有统一的格式，因此也可以方便地进行自动化处理。

对于更为复杂的项目，可以使用一些第三方的成熟的日志库来满足更复杂的需求，实现更灵活的控制。

总的来说，`printf` 调试主要用在两种情况下：

- ◇ 过于简单的情况：懒得启动调试器了 --
- ◇ 过于复杂的情况：调试器已经无能为力了，例如一些分布式的程序

assert 断言

`assert` 即断言，它同 `printf` 调试类似，也是我们对程序的状态有一些预期的值，通过在一些特定的地方插入断言，可以检查程序的真实状态和预期状态是否相符。不过断言与 `printf` 还有一些不同之处：

- ◇ C 标准库里就有断言支持，即 `assert` 宏
- ◇ 在程序里许多地方插入断言也没有关系，断言在正常的时候并不会产生输出，而且在去掉调试选项之后，断言会编译为空语句，不会影响最终程序的性能。另外，断言通常是对程序状态的一个客观描述，还可以起到注释的作用。因此在代码中保留合适的断言是比较推荐的做法。





✧ 断言出错之后立即退出，而 `printf` 则需要事后再去分析和寻找问题。然而太过于暴力也算是断言的一个缺点，因为 `bug` 有大小疾缓，有时候让程序能持续稳定地运行也是很重要的，因此除非特别严重的时候，人们通常会倾向于使用更加温和的记录日志的方式来记录下潜在的 `bug`，而不是直接结束程序。

终极调试技术

如果把调程序比作是玩一个 `RPG` 游戏，那么玩家的最强必杀技应该就是“分析源代码”了，这招威力强大，金木水火土各种属性相生相克以及各种道具魔法（比如“隐身”、“魔防”）通通无视，无人能挡，然而有几个缺点：

- 需要练级练到一定程度，积累了相当多的经验值才能施展出来
- 对 `MP` 的消耗过大，几乎一次性耗光，如果碰到小 `boss` 就随便使用的话，再碰到大 `boss` 就只有等死了……

就是这样，虽然听起来很华丽，但是这其实算是没有办法时的办法。有一些情况调试器无法处理，例如由于多线程条件竞争引发的 `bug`，调试器通常都会感到力不从心；还有一些 `bug` 相当暴力，如果你还准备拿起大刀和它对砍的话，它会一颗核弹让你一秒回到解放前，比如这几天正在折磨我的一个 `bug`：一段 `Matlab` 程序，运行数小时（因为我不可能一直盯着它看，所以我也不知道是几个小时）之后会导致系统死机，`Linux` 下是系统直接 `freeze`，`Windows` 则是经典蓝屏，对此我除了分析源代码之外看来也别无它法了；此外，还有一些 `bug`





不暴力，但是会玩隐形，随机出现，难以准确重现，或者是每当你启动调试器或者甚至只是添加一个 `printf` 语句时它也会立马躲起来，这类 `bug` 中最极品的一种有一个很形象的名字：`Heisenbug`，在后面的 `bug` 分类时还会讲到(update: 后来打印之前忘记列出 `heisenbug` 了，这算是这个小册子的 `bug` 了-.-)。

不过，分析源代码几乎就没有什么技巧可言了，或者是有太多的技巧，讲都讲不完了，这是比普通的 `debug` 要更看经验的东西，所以需要练级呀，多读精彩的代码是会受用终身的。此外，分析源代码也并不一定是纯手工的事情，目前有许多商业的和科研的静态源代码分析工具都是试图通过自动分析源代码的方式来找到潜在的 `bug`，其实将编译器警告全开(-Wall)来编译程序也算是类似的做法了。

最后，虽然没有什么具体的技巧可以在这里讲，但是有一点必须明确，分析源代码的时候一定要保持头脑清醒，并且一定要事先明白程序本来的意图。有一个通常效果很好的方法：如果可能的话，找一个朋友，把你想要完成的功能说给他听，然后在把你写的代码——你如何实现这个功能的，解释给他听，不过，目的并不是希望对方听了之后能立即给你指出错误所在，除非是相当低级的错误或者对方对这个东西非常熟悉，真正的意义在于，有时候你自己认为自己“明白”了一个东西，但其实是相当混乱的，然而如果要讲给别人听的话，就不一样了，需要对思路进行组织和整理，这个时候头脑会渐渐清醒起来，通常自己就会发





现问题的所在了。所以，如果一时找不到听众，对着一个布偶讲也是一个不错的选择。☺

调试技巧

7 GOLDEN RULES OF DEBUGGING

原本是在《The Developer's Guide to Debugging》一书中的 13 Golden Rules of Debugging，我把它们精炼了一下，变成 7 条，加上了我自己的理解，列在这里。特别是在调试到大脑一片混乱不堪的时候可以看一看，理一下思路：

1. 理解需求，亦即程序的目的，首先要明白自己“想要做什么”。
2. 重现 bug，这是很重要的一步，如果 bug 不能重现，几乎就只能靠前面说的“终极调试技术”来调试了。
3. 隔离 bug，例如，如果你已经能成功地在输入 11 的时候重现 bug，那么可以开始尽量缩小 bug 的活动范围：是不是只有在正数的时候会挂掉？是不是只有奇数的时候会挂掉？是不是自有素数的时候会挂掉？是不是只有小于 100 的数字会挂掉？建立包围圈，让 bug 无所遁形。
4. 检查电源是否插好了。嗯，当然，这是开玩笑的，不过也不是开玩笑，如果有朋友说他的电脑突然黑屏了，问你怎么回事，也许你会想到问他一下电源是否插好，但是如果是程序出 bug 了，你大概就不会想到去检查电源了。然而，事实上，还是有必要检查一下的，《Why Program Fails》的作





者就曾经遇到过一个程序 **bug**，在笔记本使用外接电源的时候可以正常工作，在使用电池供电时却会出问题；还有一次，湖边同学的电脑突然无故死机，然后再也无法启动了，我当时建议拔掉电源试试，不过最后是把电池卸掉接上电源才把问题解决了。当然，也许你仍然不会屑于在程序出错的时候去检查电源，而我这里当然也只是打一个比方而已，总而言之，生活中有许多你习以为常的东西，它们一直一来都是那个样子，但是那并不代表它们不会发生变化。有一次 **quark** 同学的电脑突然出现诡异的错误，各种程序以各种原因崩溃掉，最后发现原因是磁盘空间耗尽了。所以，在检查自己的代码时，也要细心观察外部环境的变化。

5. 每次只做一个改动，这个适用于代码量比较多的情况，因为在现实生活中代码中往往还隐藏着其他的 **bug**，如果一次改动太多有可能会让其他 **bug** 趁乱来搅浑水。当然，“一个”的概念比较模糊，并不是指一行代码，总之，不要让你的改动引入新的 **bug**，否则辛苦建立起来的包围圈就
6. 如果你没有修复 **bug**，那么它就还活着。这句话听起来有点拗口，不过我想应该许多人都遇到过类似的情况：就是正在调试一个 **bug**，改着改着它就突然没了，以前能重现的办法也无法重现了，看起来好像是它受不了自杀了或者是被误杀了——然而，实际上它还有相当高的概率还活着，这句话说的就是这个意思，所以，除非你这个程序只用一次，否则，生要见虫，死要见尸，一定不能像武侠小说里一样看到别人掉下悬崖了就得得意洋洋扬长而去了。--bb





- 找出并修复 **bug** 之后为它写一个回归测试用例。这个也是在较大的项目中很必要的一个环节。简单的测试用例就是用原来可以重现 **bug** 的输入来运行程序，并要求程序得到正确的结果。回归测试的目的是为了保证不会在将来犯同样的错误——这在较大的项目中是比较常见的一件事。有了回归测试，在每次做较大更改之后运行一遍所有以前积累的回归测试，可以（在很大程度上）保证以前修复过的 **bug** 不会起死回生。

良好的编程习惯

编写 **warning free** 的代码

编译的时候打开编译器的 **warning** 选项(gcc 里是 **-Wall**)，并尽量保证代码编译不会产生 **warning**，因为很多 **warning** 往往预示可能会有 **bug**（例如示例代码 03）。

不过，有时候编译器可能过于严格，例如 **Visual Studio**，用一下 **strcpy** 也要报一个 **4996** 的警告，一堆 **4996** 的警告有可能会把真正有价值的信息给淹没掉。如果想要关掉这个 **warning**，可以更改编译选项，或者在代码里写上：

```
#pragma warning(disable:4996)
```

适当使用断言





前面已经说过断言可以用来调试，并且断言也能起到一定注释的作用。我没有说要“多用断言”，因为断言其实同注释是一样的，并不是越多越好，关键是要在合适的地方用合适的断言，嗯，我只能说这么多了。 \ (/ _ \) /

总之我是不想看到这样的情况出现的：

```
/// <summary>
/// Turns true into false and false into true
/// -- similar to the church of scientology.
/// <param name="_booInpt">True of false</param>
/// <returns>False or true</returns>
private bool trueandorfalse(bool _booInpt)
{
    // I'm quite sure though there is a very
    // clever C# standard command doing this,
    // I just can't find it right now ...
    if (_booInpt == true)
        return false;
    return true;
}
```

没错，这是真实的代码，来源请参考 The Daily WTF 里的这篇文章 (<http://thedailywtf.com/Articles/The-Clever-Coder.aspx>)，题外话：如果你觉得有趣，还有一些地方可以去休闲一下：

◇ <http://forums.sun.com/thread.jspa?threadID=5404590>





✧ <http://us.php.net/manual/en/function.abs.php#58508>

编写 debugger friendly 的代码

编写调试器友好的代码，最明显的一点就是对常量的定义，在 C 语言里通常有两种方式可以定义常量，一是使用 `#define` 来定义一个宏，这是很常见的用法，另一种是使用 `enum` 来定义枚举值。可能的时候尽量使用后者，一来可以把一组相关的常量放在一起，二来调试器能理解枚举值，却无法理解宏定义的常量值，即使编译时加上 `-g` 选项也不行，因为宏常量在编译器编译的时候已经被预编译器展开了。所以，如果使用 `enum` 的话，在调试时可以看到诸如 `OPT_OPEN`、`OPT_CLOSE` 这样的名字，而不是 `0`、`1` 这样的“无意义”的数字，还要再转回去查阅源代码对照是什么意思。

另外，可以专门针对调试写一些辅助函数，如果程序状态比较复杂，可以写一个诸如 `inspect_program_status` 的函数，将有价值的信息格式化输出出来，然后在调试器里可以用 `print` 或者 `call` 命令来调用这样的辅助函数来帮助查看程序状态。例如，你的程序是构建一个树，如果你有一个辅助函数能以树型格式将树的内容打印出来，应该会比在调试器里一个一个地 `follow` 指针慢慢展开要轻松许多。

另外，如果可能的话，在为调试进行编译的时候(`-g`)，尽量不要打开优化选项(`-O`)，否则会对调试带来不少麻烦(最典型的一个情况就是如果函数被内联了，就没法在调试器中调用了)。





Program Proof

人们从来就没有放弃过尝试从数学的角度来证明程序的正确性。我在《C 专家编程》（好书，推荐阅读）上看到一封 1991 年的邮件列表里的邮件，整个帖子原来是讨论在 C 语言里如何不使用临时变量来交换两个数的值的，有人给出了那个经典的答案：

```
*a ^= *b;      /* Do 3 successive XORs */
*b ^= *a;
*a ^= *b;
```

于是有了这么一个回帖：

```
From: A proponent of program proofs
Date: Fri May 15 1991, 12:43:52 PDT
Subject: Re: Swapping 2 values without a temporary.
Someone asks if the following program fragment (to swap 2
values) works:
```

```
*a ^= *b;      /* Do 3 successive XORs */
*b ^= *a;
*a ^= *b;
```

Here's the answer.

Make the Standard Assumptions that (1) this sequence executes atomically, and (2) it executes without hardware failure, memory limitations or math failure. Then after the sequence





```
*a ^= *b; *b ^= *a; *a ^= *b;
```

*a, and *b will have the values f3(a), and f3(b) where:

```
f3 = lambda x.(x == a? f2(a) ^ f2(b): f2(x))
```

```
f2 = lambda x.(x == b? f1(b) ^ f1(a): f1(x))
```

```
f1 = lambda x.(x == a? *a ^ *b: *x)
```

or in more readable terms:

```
f3(a) = f2(a) ^ f2(b), f3(x) = f2(x) else
```

```
f2(b) = f1(b) ^ f1(a), f2(x) = f1(x) else
```

```
f1(a) = *a ^ *b, f1(x) = *x else (provided that *a and *b  
are defined, i.e. a != NULL, b != NULL). This leads to only  
two solutions (derived by beta reduction), namely:
```

```
if a and b are the same: f3(a) = f3(b) = 0
```

```
if a and b are different: f3(a) = b, f3(b) = a.
```

And about reliable verification and debugging:

mathematical verification and proof is the only reliable technique. Everything else is engineering hacks. And contrary to the commonly received myth, all of C is easily tractable in this way by mathematical analysis.

作者试图通过 lambda 演算来从数学上证明这段代码的正确性，并在末尾声称：数学上的证明是验证和调试程序的唯一可靠手段，其他所有的都只是“engineering hacks”。





他还说，虽然大家都认为不可能，但是其实 C 语言的所有东西都是可以通过这样的方式来从数学上进行分析的。当然，对此他并没有给出数学上的证明，且不论他的理论正确与否，仅 3 行的代码就需要这么长的东西来证明（而且证明的语言比 C 代码还要更难懂），感觉实在是意义不大，如果是一个完整的程序，那程序的正确性证明是不是可以出一本书了？这里有必要引用一下一个不知道出处的 quote:

- ✧ *The problem with engineers is that they cheat in order to get results.*
- ✧ *The problem with mathematicians is that they work on toy problems in order to get results.*
- ✧ *The problem with program verifiers is that they cheat on toy problems in order to get results.*

说得很传神。更何况，冗长的数学证明本身是不是有错还不一定呢，这里是之前的那个作者的后续跟帖，他指出了自己之前的证明有一点点问题:

```
From: A proponent of program proofs
Date: Fri May 15 1991, 13:07:34 PDT
Subject: Re: Swapping 2 values without a temporary.
where I previously wrote:
This leads to only two solutions (derived by beta reduction),
namely:
```





```
if a and b are the same: f3(a) = f3(b) = 0
if a and b are different: f3(a) = b, f3(b) = a.
I actually meant to write
```

```
f3(a) = *b, and f3(b) = *a...
```

不仅他的证明存在错误，而且这段被从数学上证明了的程序也是有问题的：虽然有点非法输入的嫌疑，但是如果我传递的两个指针 **a**、**b** 是指向同一个整数的话，这段代码会把它变为 **0**——这当然不是期待的结果，自己和自己交换怎么就变成 **0** 了呢？

总的来说，“**engineering hacks**”虽然不能证明程序绝对是对的，却能找到程序出错的地方。而数学的方法，往往都需要建立在一个严格的数学模型上，这样的模型通常是从现实生活抽象出来或者甚至是凭空捏造的，但是不论何种情况，总是和真实的世界还是有一定出入的，所以，大概还是不能 **100%** 适用了。反正（纯粹个人观点）我觉得 **program proof** 就好比试图从数学上来证明一幅画美不美一样的奇怪。 :-/

BUG 分类

低级错误造成的 BUG

这是人们最常犯的一类错误，通常即使是经验丰富的程序员也不能完全避免，造成的原因有很多，比如





- 不小心产生的打字错误，比如示例代码 `o3`，又比如把

```
char *msgs[] = {"foo", "bar", "baz"};
```

写成了

```
char *msgs[] = {"foo" "bar", "baz"};
```

仍然能编译通过，但是却完全变了一个意思。

- 也有可能是大脑一时混乱，或者不在状态，写出了诸如将函数的局部变量的地址返回，或者在对链表进行删除是发生的指针错误之类的。
- 还有可能就是对概念本来就理解不清楚，比如，这样的多维数组的使用：

```
#include <stdio.h>

int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};

void foo(int **b) {
    printf("%d\n", b[1][2]);
}

int main() {
    foo((int **)a);
}
```





可以通过养成良好的编程习惯，以及从编译器、语言等各方面的辅助来尽量避免低级错误的发生。

内存 BUG

在前面的内存调试一节已经详细讨论过了，这里就不再重复了。

逻辑错误

这样的错误代码本身并没有任何问题，但是程序的算法可能是错的，因此会得到错误的结果。这类问题通常都涉及到问题相关的特定领域的专业知识。

IT'S FEATURE, NOT BUG

C 语言有很多令人崩溃的 **feature**，容易让人混淆或者忽略，从而造成一些问题，比如 **switch** 的每一个 **case** 后面（通常）都要加上一个 **break**，再比如 C 里那难用的多维数组，但是，不管怎么说，这是 **feature**，不是 **bug**，至少不是 C 语言的 **bug**。ヽ (' _) ㄥ

未定义行为

未定义行为(**undefined behavior**)是指一些由标准明确规定的“未定义”的行为，听起来比较拗口，其实就是 C 语言标准规定了，这里可能是这样也可能是那样的，具体怎么样就要看具体的实现了，而如果程序员凭自己的直觉假定必然是某种实现方式而依赖与这个来进行编程的话，那他写出来的程序就属于行为未





定义的，也许在一个编译器上可以运行，在另一个编译器上就无法运行了。典型的例子是示例代码 07，这里涉及到一个 **Sequence Point** 的概念，**Sequence Point** 就是程序的一个点，在这个点之前的副作用(side effect)和之后的副作用的求值是有先后顺序的。C 语言里面有不少运算符都是一个 **Sequence Point**，包括：

- 逻辑运算符：&&, ||
- 问号运算符：Cond? Val1:Val2
- 逗号运算符：Val1, Val2
- 函数调用：在所有参数都求值之后才会进入函数体，然而，函数的参数列表并不是一个逗号运算符的串联，参数之间的求值先后顺序也是未定义的

然而也有不少著名的不是 **Sequence Point** 的运算符，比如算术运算符，赋值运算符，还有索引运算符 **A[B]** 中 **A** 和 **B** 的求值顺序也是未定义的。因此，虽然有必要明白 **i++** 和 **++i** 的区别，但是却不要纠结于 **i++ + ++i** 这样的东西，因为求值顺序的不同会导致不同的结果。

示例代码

这里列出一部分演示时用到的代码，除了个别非常简单的例子之外，其他的程序均是来自我自己、俱乐部的朋友们或者是在学校的论坛上贴出来过的真实代码（或者抽取出来的版本）。





我尽量保证每段代码都是一个完整的可以编译运行的版本，这样方便大家自己去尝试。另外，每段代码会有三种相关信息：

1. 代码的目的：虽然对于某一些比较典型的 **bug**，经验丰富的程序员有时候能凭“直觉”直接看出可能会有问题的部分，甚至不需要理解完整的代码。但是，没有理解程序就开始调试并不是一个好习惯，能解决的问题也是相当有限的。更多的时候，我们需要先理解代码“试图做什么”，才能对比代码“实际做了什么”从而找出问题所在。因此，在每段代码之前我会包括一段简短的说明，描述代码的目的。
2. 程序的 **bug**：换句话说，是代码 **bug** 的外在表现，即程序的实际行为和期待的行为不同的地方。对于一些棘手的 **bug** 来说，要重现问题也是很困难的一件事，需要特定的运行环境、特地的输入数据或者甚至是完全随机性地出现和消失的。这部分信息将会被放在小册子末尾，可以通过代码编号查到。
3. 代码的 **bug**：即造成问题的真正原因，找到它通常是整个 **debug** 过程中最困难的一步。这个信息我被放在小册子末尾，可以通过代码编号查阅到。

如果你能独立找出这里列出的所有 **bug**，那么恭喜你，你的 **debug** 能力已经相当强了。当然，也不要因此而太过得意，这里的代码虽然都来自实际的程序，但是也都是为了方便讲解而做了筛选和抽取，大致是在一段相对较短的程序中的比较单一的 **bug**。然而在实际的系统中，情况远比这里复杂，代码数量可能会非常大，甚至可能会有许多 **bug** 互相勾结；另一方面，**debug** 本身就不是一





个纯技术活（技术是必要不充分条件），同经验、心情、灵感、运气、宇宙射线、伦敦的蝴蝶等各种因素都有可能相关。

01. BUGGY 的 COPY_STRING

这段程序是我搜集的几个很典型的 bug 拼凑在一起生造出来的一段代码，功能就是将 main 函数的 argv 第一个元素（即可执行文件自己的路径）复制一下然后打印出来。

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

char *copy_string(char *str) {
    char *mem = (char *)malloc(strlen(str));
    if (mem=NULL) {
        printf("error: failed to allocate memory\n");
        return NULL;
    }
    strcpy(mem, str);
    return mem;
}

int main(int argc, char **argv) {
    char *copied = copy_string(argv[0]);
    if (copied != NULL) {
```





```
    printf("Copied string: %s\n", copied);  
}  
free(copied);  
return 0;  
}
```

02. CRASH CALL STACK

这段代码没有什么实际意义，它试图在运行之后得到自己所在的目录(`argv[0]`总是可执行文件本身的路径)，并打印出来，完全是我为了演示而写的，不过其中的 `bug` 片段却是实际存在的一个 `bug` 物种，并且在过去我也曾亲自遇到过。

```
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
#define FILESEP '/'  
#define BUFLen 1024  
  
void get_directory(char *filename, char *buf) {  
    size_t idx;  
    for (idx = strlen(filename)-1; idx >= 0; --idx) {  
        if (filename[idx] == FILESEP)  
            break;  
    }  
    memcpy(buf, filename, idx);  
}
```





```
    buf[idx] = '\\0';
}

void call_level_n(char *filename, int n) {
    if (n == 0) {
        char directory[BUFLen];
        get_directory(filename, directory);
        printf("The directory is %s\\n", directory);
    } else {
        call_level_n(filename, n-1);
    }
}

void greet() {
    printf("Hello! I'm a program that print the directory of
myself.\\n");
}

int main(int argc, char **argv) {
    greet();
    call_level_n(argv[0], 5);
    return 0;
}
```

03. 诡异的 SWITCH 语句





这段代码的原型是 **moonykily** 同学之前曾经遇到过的一个 **bug**，不过原来的代码已经找不到了，所以我捏造了这么一个简短的程序。程序的目的是构造一个函数，根据参数的不同得到不同的欢迎词。

```
#include <stdio.h>
#include <string.h>

#define HELLO_WORLD 0
#define HELLO_KID 1
#define HELLO_MSTC 2

void get_greeting(char *buf, int what) {
    switch(what) {
        case HELLO_WORLD:
            strcpy(buf, "Hello world!");
            break;
        case HELLO_KID:
            strcpy(buf, "Hello kid!");
            break;
        default: /* all others considered MSTC */
            strcpy(buf, "Hello MSTC!");
            break;
    }
}

int main() {
```





```
char buf[128];
for (int i = 0; i < 4; ++i) {
    get_greeting(buf, i);
    printf("[%d] => %s\n", i, buf);
}
return 0;
}
```

04. 按字母序列出文件

这是来自一位同学的作业代码，没有做任何修改，代码的目的很明了，排序后列出在/etc 目录下的所有文件：

```
#include<stdio.h>
#include<sys/types.h>
#include<dirent.h>
#include<unistd.h>
#include<string.h>
int main()
{
    DIR *dir;
    int n = 0,i,j;
    char *q;
    struct dirent *ptr;
    char *p[100];
    char s[100][256];
```





```
dir = opendir("/etc");
while((ptr = readdir(dir)) != NULL)
{
    strcpy(s[n], ptr->d_name);
    n++;
}
for(i = 0; i < n; i++)
    p[i] = s[i];
for(i = 0; i < n-1; i++)
{
    for(j = 0; j < n-1-i; j++)
    {
        if(strcmp(p[j], p[j+1]) > 0)
        {
            q = p[j];
            p[j] = p[j+1];
            p[j+1] = q;
        }
    }
}
for(i = 0; i < n; i++)
    printf("%s\n", p[i]);
closedir(dir);
}
```

05. 计算反应时间





这段代码来自一位同学的作业，目的是通过显示一个字符然后让人输入看到的字符来计算人的反应时间：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<unistd.h>
int main()
{
    int i , j;
    int n,num;
    float rate;
    struct timeval tv1,tv2;
    struct timezone tz;
    long rt,tt,at;
    srand((int)time(0));
    tt = 0;
    n = 0;
    for(i = 0; i < 10 ;i ++)
    {
        printf("请输入您所看到的数字\n");
        j = 1 + (int)(10.0*rand()/(RAND_MAX + 1.0));
        printf("%d\n",j);
        gettimeofday(&tv1, &tz);
        scanf("%d",&num);
        if(num == j)
```





```
{
    gettimeofday(&tv2 ,&tz);
    rt = tv2.tv_usec-tv1.tv_usec;
    printf("您第%d 次的反应时间是%ld 毫秒\n",i+1,rt);
    n ++;
}
else
    printf("您输入的是错误的数字\n");
    tt += rt;
}
at = tt / 10;
rate = n / 10.0 * 100;
printf("您这十次反应正确率是%f %，平均反应时间是%ld 毫秒\n",rate, at);
return 0;
}
```

o6. R,N,M,T,I,J

单字母的变量名，这是一段典型的 ACM 代码，题目大致是输入一个 test case 数，每个 test case 的输入是两个数 n 和 m ，输出 $(0!+1!+...+n!)^m$ 的结果。原来的代码是一段 java 代码，用了 `BigInteger` 来处理输入数据过大的情况，现在我暂且忽略过大的输入，转化为一个简易的 C 语言版本。

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





```
int r, n, m, t;

int main() {
    int t;
    scanf("%d", &t);
    for (int i = 0; i < t; ++i) {
        scanf("%d%d", &n, &m);
        r = 1; t = 1;
        if (n >= m) n = m;
        for (int j = 1; j <= n; ++j) {
            t *= j;
            if (t >= m) {
                t %= m;
                if (t == 0) break;
            }
            r += t;
            if (r >= m) r %= m;
        }
        if (r >= m) r %= m;
        printf("%d\n", r);
    }

    return 0;
}
```

07. SEGTREE





这是一个构建 segtree 的程序（C++）。

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdio>
#include <cstring>

#include <limits.h>

using namespace std;

struct segnode {
    segnode(int _a, int _b) : a(_a), b(_b)
    {
        value = INT_MAX;
        right = left = -1;
    }
    segnode() {}
    int a, b;
    int value;
    int left, right;
};

typedef vector<segnode> segtree;
```





```
static int __make_segtree(segtree& tree, int a, int b)
{
    segnode node(a, b);
    tree.push_back(node);
    size_t pos = tree.size() - 1;
    if (a != b) {
        tree[pos].left = __make_segtree(tree, a, (a+b) / 2);
        tree[pos].right = __make_segtree(tree, (a+b)/2+1,b);
    }
    return pos;
}

void
make_segtree(segtree& ret, int maxn)
{
    __make_segtree(ret, 0, maxn);
}

int main(int argc, char* argv[])
{
    segtree tree;
    make_segtree(tree, 50000);
    return 0;
}
```





示例代码问题的重现

01. BUGGY 的 COPY_STRING

这个程序问题很多，并且很好重现，直接编译运行就会出错。除了这个最直接的错误之外，还有几个在运行时潜伏期很长的 bug，都是和内存相关的。

02. CRASH CALL STACK

虽然真正的大 bug 不在这里，但是宏定义 `FILESEP` 是引发 bug 的直接开关，在 `Linux` 下的路径分隔符是 `'/'`，然而 `Windows` 下却是 `'\\'`，让这个使用了错误路径分隔符的程序在 `Windows` 下运行，就会暴露出真正的 bug 所在，效果是非法内存访问。如果想在 `Linux` 下也尝试一下这个 bug，不妨再把 `FILESEP` 改为 `'\\'`。

03. 诡异的 SWITCH 语句

根据代码，`0`、`1`、`2` 分别会返回不同的欢迎词，而由于使用了 `switch` 语句的默认选项，所以所有其他值会视为和 `2` 一样，都返回 `Hello MSTC`，然而实际编译运行却得到这样的结果：

```
[0] => Hello world!  
[1] => Hello kid!  
[2] => Hello kid!  
[3] => Hello kid!
```





MSTC 去哪里了？

04. 按字母序列出文件

编译，运行，段错误。

05. 计算反应时间

编译，运行，发现计算出来的反应时间有时候是负数。

06. R,N,M,T,I,J

程序好像很不正常，例如，输入 `1 2 3`，输出 `1` 之后，程序本该退出，但是却并没有退出。

07. SEGTREE

在 Windows 下运行是正常的，在 Linux 下运行会挂掉。

示例代码 BUG 解析

01. BUGGY 的 COPY_STRING





这段代码集中了几个很容易犯的低级错误，仔细阅读代码就可以发现，然而这类错误在实际编程中其实也是出现频率相当高的，而且阅读自己写的代码时较难检查到，平时编程是需要相当小心。

- ◇ 程序运行即非法内存访问而退出，可以看到指针 `mem` 为 `NULL`，理论上前面的 `if` 语句已经排除了 `NULL` 的情况，然而问题正是出在这里：
`mem==NULL` 被写成了 `mem=NULL`，结果 `mem` 被赋值为 `NULL` 了，原来的内存丢失（泄漏）了，这个表达式的结果也为 `NULL`，导致 `if` 内的语句未能执行。最后 `strcpy` 往 `0` 地址内存写数据，导致非法内存访问而退出。
- ◇ 另外，另一个错误存在于 `malloc(strlen(str))`，应该写为 `malloc(strlen(str)+1)`，保证末尾的 `\0` 有存储空间，否则会产生 `1` 个字节的内存覆盖，程序有可能会非法退出，也有可能潜伏而造成其他问题，如果运气好的话，甚至还有可能正常地运行，这样的 `bug` 比较难查，可以借助内存调试器的帮助进行排查。
- ◇ 还有一个错误存在于 `main` 函数中，就是最后一条 `free` 语句，如果内存申请失败的话，`copied` 指针将会是 `NULL`，此时对它进行 `free` 会造成非法内存访问。然而这样的 `bug` 潜伏期极长，因为正常情况下都不会碰到内存分配失败的情况，当真正出现内存不足的时候，它就会冒出来火上浇油，让程序 `crash` 掉。

02. CRASH CALL STACK





错误的路径分隔符 **FILESEP** 定义固然是引发 **bug** 的直接原因,然而真正的 **bug** 其实是由一个低级错误所引发的内存错误,运气好的话(好吧,其实应该是“不好”的话),还可以体验到全栈崩溃的壮观景象。

这个程序是我为了演示内存错误的杀伤力而特地构造的,错误源于 **FILESEP**, **Linux** 下和 **Windows** 下采用不同的路径分隔符,这让写一个可移植性的程序变得很麻烦,对于这个程序来说,这将导致 `filename[idx] == FILESEP` 无法满足,理想情况下,我们最多得到一个错误的输出就完了,然而这里另一个 **bug** 横插一脚,让局势一发不可收拾:注意看 **for** 循环的条件 `idx >= 0`,问题就在这里, `idx` 是 `site_t` 类型的,这是一个无符号整数类型,在等于零时再递减就会轮回到最大值,永远都不会小于零!所以这个 **for** 循环永远都不会退出,除非在茫茫内存中恰巧碰到一个等于 **FILESEP** 的字符。

在 **Visual Studio** 里运行,我遇到了两种结局:

1. 在 **for** 循环结束之前指针野到操作系统无法忍受了,系统以非法内存访问的罪名将程序强行终止,此时堆栈尚且正常。
2. 碰巧找到了这么一个字符, **for** 循环终止了,此时 `idx` 是一个非常大的值, `memcpy` 紧接着来了一个危险的大片内存拷贝,之前的函数调用栈完全被无意义的值覆盖(注意栈是由高地址向低地址增长的),程序被强行终止,然而此时现场已没有全尸了, **Visual Studio** 的 **Call Stack** 窗口只能无奈地给出一些无意义的值:





Call Stack	
	Name
→	feeeffee()
	ntdll.dll!779e65f9()
	[Frames below may be incorrect and/or missing, no symbols loaded for ntdll.dll]
	ntdll.dll!779e65cb()
	ntdll.dll!779c8d3d()
	ntdll.dll!779e6457()
	ntdll.dll!779e65f9()
	ntdll.dll!779e65cb()

代码中的 `call_level_n` 是为了制造多层调用堆栈用的，正常情况下应该能看到清晰的调用堆栈。而 `greet` 函数则纯粹是为了搅乱局势用的。

03. 诡异的 SWITCH 语句

好吧，如果你不管三七二十一，直接就开始调程序，估计会崩溃掉，为什么 `switch` 语句明明写在那里了，却没有按照预期的执行呢？难道是编译器的 `bug`？也许你会想到看看生成出来的汇编代码对不对，那样应该能找到问题所在了，不过，其实找到错误通常更容易：在 **Visual Studio** 里打开这段代码，可以看到 `default` 关键字并没有被高亮——发现了么？它被写成了 `default!`！那为什么编译器没有报错呢？因为这是完全符合语法的！

按照 C 语言的语法，虽然我们试图写 `switch` 的默认选项，然而实际上却是在这里我们定义了一个标号为 `default` 的标签，标签是什么？`goto` 语句总是知道的





喽，没错，就是那个被很多人（比如著名的 Edsger Dijkstra）骂的东西，Perldoc 上是这么描述它的：

goto - create spaghetti code

当然，`goto` 并不是全无用处，但是，不到万不得已请不要使用它，是的，这是讲调试的，不过其实跑题并不严重，因为如果滥用 `goto` 的话，调试的时候将是一场噩梦。

总的来说，这是一个相当低级的错误，打字速度太快了（其实我估计 moonykily 犯这个错误的主要原因是他的键盘太破了，我用过他的键盘，难按程度是无法想像的），但是，这么低级的错误实在是有点难以想象，如果打开编译器的警告选项的话会很快发现，而且随便找一个带语法高亮的编辑器……好了，另一个悲剧出在这里，

```
1 switch (what) {
2     default:
3         break;
4     default:
5         break;
6 }
```

moonykily 同学是 VIM 粉丝，VIM 很聪明，它认出了 `default` 是一个标签，于是也给高亮了一下，可以要命的是，它用了和正常的 `default` 一样的颜色来高亮……

04. 按字母序列出文件

很低级的错误，硬编码了缓冲区的大小，结果通常 Linux 系统里/etc 下的文件都超过 100 个，所以内存错误了。





05. 计算反应时间

代码中用两个结构的毫秒数之差来计算反应时间，然而没有考虑进位问题，如果反应时间大于 1 秒的话，就需要再结合秒之差来计算了，因此出现了负值。

06. R,N,M,T,I,J

bug 处在变量 `t` 上，全局变量 `t` 本来是做临时变量使用，但是在内部又定义了一个变量 `t` 用来表示 test case，于是全局的 `t` 被里面的 `t` 隐藏掉了，结果同一个 `t` 被当作两个变量来用，就导致程序混乱了。

07. SETGREE

用 gdb 跟踪程序很诡异，在

```
tree[pos].left = __make_segtree(tree, a, (a + b) / 2);
```

这样的一个语句挂掉了。用 step 跟踪进函数之后，发现先从 vector 中取出了元素，然后 `__make_segtree` 被调用，都结束之后在赋值的时候才挂掉了，相当诡异。而且，只在某些特殊值的时候才挂掉，例如，这里在 `pos==4095` 的时候挂掉了。一切都是好好的，跟踪，观察变量都是很正常的，怎么就在最后一个等号运算的时候就挂掉了？

跟踪到这个程度也几乎是没有什么办法了，不过 4095 这个数字比较诡异，了解 STL 的 vector 的实现的应该知道它里面内存重新分配是按照 2 的指数来做





的, pos 等于 4095 时再运行 `__make_segtree`, 刚好会再插入一个元素到 `vector` 中, 此时 `vector` 元素个数为 4096, 应该是碰巧需要重新分配内存了, 于是原来的内存就被 `free` 掉了, 数据移动到了新的大块内存处, 可是之前取出来的那个引用还指向原来的内存地址, 于是赋值时一写那个已经 `free` 掉的地址, 就出现内存错误而退出了。

这个错误一方面可以看作是内存错误, 另一方面也可以当作是由于未定义的行为引发的错误。如果复制运算符 (也就是等号) 右边的表达式总是比左边的表达式先求值的话, 就不会出现这样的问题, 然而 C 语言里并没有做这样的规定, 一个赋值运算, 左边和右边的表达式的求值顺序是没有做任何规定的, 编译器可以采用任意的顺序 (由于 Visual C++ 编译器恰好采用了相反的顺序所以运行的时候没有出问题)。

