《C 程序设计》实验手册

（2021-2022 学年第 1 学期）

供 2021 级黄炜老师班级使用

2021 年 6 月 29 日

目 录

1 实验简介

本文档为 C 程序设计课程的实验手册。

1.1 教学目的

通过实验课教学，学生应了解计算机硬件与操作系统的基本概念，熟悉程序集成编译环境，熟练掌握程序的编译和运行、程序的调试、静态检查与测试，并在教师指导下完成一项课程项目，从而加深对理论课程知识的理解。

1.2 授课方式

课堂采用师生互动的方式完成。教师先做简要讲解，学生在教师指导下实际操作，由先完成的学生帮助尚未完成的学生，最后学生提问的方式进行。

1.3 特别说明

本文档是在总结过去几个学年 C 程序设计实验课教学经验的基础上编制的。目前，文档已设定整体框架，但内容是对以往零星发给学生的资料和网上的教程的剪贴。文字和详略可能未必完全准确，请以课堂讲授为准。

欢迎随时将宝贵建议和意见发给我。

2 集成开发环境

2.1 DevC++

DevC++是一种绿色版的 C / C ++集成开发环境（IDE）。本课程强烈推荐DevC++作为学生用 IDE。因为 DevC++有以下优点：

第一，绿色版。DevC++可以做成绿色版的压缩包。学生可以在平时将其保存在邮箱或者云盘里，到机房下载使用。不同于安装动辄数十分钟尺寸动辄数百兆的 IDE，DevC++支持初学者随时随地可以编程，十分方便。

第二，标准语法。DevC++集成 GCC 作为编译环境，可以使学生熟悉主流 C 编译器和主流 C 语法。学生做题的 OJ 平台一般搭建在 Linux 操作系统，使用 GCC 编译器。而微软 C/C++编译器的语法引入了部分微软自定义的成分，在该环境下通过的程序难以直接在 OJ 上使用。

但是，DevC++也有其缺点。由于 DevC++功能简单，因而在构建较为复杂的软件时，很少有程序员会使用它。另外，DevC++的调试功能相较于商业软件并不够友好。

2.2 Visual Studio

Microsoft Visual Studio（简称 VS）是美国微软公司的开发工具包系列产品。VS 是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具。其中，包含对 C 语言文件编译和运行的功能。

VS 作为商业软件（VS 还提供了免费版）有功能齐全、用户界面友好的优点。在编程方面，VS 能协助检查明显的数组溢出和初始化等错误。尤其在调试方面，VS 能直观地在断点间切换，并可以方便地查看指定内存区域的内容。在较复杂软件的开发和协作中使用 VS，可以节省开发管理上的精力。

然而，VS 并不适合初学者 OJ 做题。VS 不支持对单个 C 文件的编译，必须新建工程再新建 C 文件。部分初学者为了方便而使用默认的 C++文件格式，这可能使用一些 C 语言不支持的语法而没有发现。此外，VS 里引入了一些微软对

C 的附加定义，如强制使用 scanf_s 等。这在 VS 编译通过的程序复制粘贴到 OJ 系统中时，往往会出现错误。

2.3 其它开发环境

其它 IDR 还有：Eclipse、Qt 等。学有余力的同学应了解不同 IDE 的使用方法和自动生成的目录和文件，以便于将来在工作时找到参考代码后，快速找到编译方式。

3 编译器

3.1 GCC

GCC（GNU Compiler Collection，GNU 编译器套件）是由 GNU 开发的编程语言译器。GNU 编译器套件包括 C、C++、 Objective-C、 Fortran、Java、Ada 和 Go 语言前端，也包括了这些语言的库（如 libstdc++，libgcj 等）。

在使用 GCC 编译器的时候，我们必须给出一系列必要的调用参数和文件名称。GCC 编译器的调用参数大约有 100 多个，这里只介绍其中最基本、最常用的参数。具体可参考 GCC Manual。

-v gcc 执行时执行的详细过程，gcc 及其相关程序的版本号

3.2 Microsoft C Compiler

Microsoft C Compiler 是 Visual Studio 编译 C/C++文件时调用的编译器可执行程序。该程序允许用户使用批处理文件调用命令行编译 C 文件。

4 编译和运行

4.1 简介

计算机不能直接识别和执行用高级语言写的指令，必须用编译程序把 C 源程序翻译成二进制形式的目标程序，然后再将该目标程序与系统的函数库以及其他目标程序连接起来，形成可执行的目标程序。

在编好一个 C 源程序后，怎样上机进行编译和运行呢？一般要经过以下几个步骤：

（1）上机输入和编辑源程序。通过键盘向计算机输入程序，如发现有错误，要及时改正。最后将此源程序以文件形式存放在自己指定的文件夹内，文件用.c 作为后缀，生成源程序文件，如 f.c。

（2）对源程序进行编译，先用 C 编译系统提供的“预处理器”对程序中的预处理指令进行编译预处理。例如，对于#include<stdio.h>指令来说，就是将 stdio.h 头文件的内容读进来，取代#include<stdio.h>行。由预处理得到的信息和程序其他部分一起，组成一个完整的，可以用来进行正式编译的源程序，然后由编译系统对该源程序进行编译。

编译的作用首先是对源程序进行检查，判断它有无语法方面的错误，如有，则发出“出错信息”，告诉编程人员认真检查改正。修正程序后重新进行编译，如有错，再发出“出错信息”。如此反复进行，直到没有语法错误为止。这时，编译程序自动把源程序转换为二进制形式的目标程序，如果不特别指定，此目标程序一般也存放在用户当前目录下，此时源文件没有消失。

在用编译系统对源程序进行编译时，自动包括了预编译和正式编译两个阶段，一气呵成。用户不必分别发出二次指令。

（3）进行连接处理。经过编译所得到的二进制目标文件（后缀为.obj）还不能供计算机直接执行。前面也说明：一个程序可能包含若干个源程序文件，

而编译是以源程序文件为对象的，一次编译只能得到与一个源程序文件相对于的目标文件，它只是整个程序的一部分，必须把所有的编译后得到的目标模块连接装配起来，再与函数库相连接成一个整体，生成一个可供计算机执行的目标程序，称为可执行程序，在 Visual C++中其后缀为.exe，如 f.exe。

即使一个程序只包含一个源程序文件，编译后得到的目标程序也不能直接运行，也要经过连接阶段，因为要与函数库进行连接，才能生成可执行程序。

4.2 在 IDE 编译和运行

在 IDE 中，一般通过菜单项自动调用编译和运行命令行。一般地，在 VS 下运行在“生成”菜单下的“生成解决方案”项。在 DevC++下运行在“运行”菜单下的“编译”或“编译运行”项。

使用 IDE 编译和运行的好处在于软件制造商将大多数程序员经常使用的功能集成到少数菜单项中，支持一键运行，节省时间。

4.3 命令行编译

虽然 IDE 编译和运行省去程序员手工输入命令的烦恼，但是程序员在编译时很可能需要定制个性内容。使用命令行编译的好处有两点：第一，便于快速设置各种自定义配置。用鼠标寻找配置界面比用键盘直接输入命令行参数要慢得多。第二，支持程序员将不同的命令行合并于一个批处理文件中，使用者可以根据需要一键运行。

4.4 GCC 命令行编译

4.4.1 GCC 的基本用法

GCC 最基本的用法是：gcc [options] [filenames]

其中 options 就是编译器所需要的参数，filenames 给出相关的文件名称。

-c，只编译，不链接成为可执行文件，编译器只是由输入的.c 等源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output_filename，确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

-g，产生符号调试工具（GNU 的 gdb）所必要的符号资讯，要想对源代码进行调试，我们就必须加入这个选项。

-O，对程序进行优化编译、链接，采用这个选项，整个源代码会在编译、链接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、链接的速度就相应地要慢一些。

-O2，比-O 更好的优化编译、链接，当然整个编译、链接过程会更慢。

-Idirname，将 dirname 所指出的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。C 程序中的头文件包含两种情况：

A）#include <myinc.h>

B）#include “myinc.h”

其中，A 类使用尖括号（< >），B 类使用双引号（“ ”）。对于 A 类，预处理程序 cpp 在系统预设包含文件目录（如/usr/include）中搜寻相应的文件，而 B 类，预处理程序在目标文件的文件夹内搜索相应文件。

4.4.2 GCC 的简单编译

这里举一个示例程序如下：

```
//test.c
#include <stdio.h>
int main(void)
{
       printf("Hello World!\n");
       return 0;
}
```

这个程序，一步到位的编译指令是 `gcc test.c -o test`。实质上，上述编译过程是分为四个阶段进行的，即预处理（也称预编译，Preprocessing）、编译（Compilation）、汇编（Assembly）和连接（Linking）。

a. 预处理

gcc -E test.c -o test.i 或 gcc -E test.c 可以输出 test.i 文件中存放着 test.c 经预处理之后的代码。打开 test.i 文件，看一看，就明白了。后面那条指令，是直接在命令行窗口中输出预处理后的代码。

GCC 的 -E 选项，可以让编译器在预处理后停止，并输出预处理结果。在本例中，预处理结果就是将 stdio.h 文件中的内容插入到 test.c 中了。

b. 编译

预处理之后，可直接对生成的 test.i 文件编译，生成汇编代码：gcc -S test.i -o test.s

GCC 的 -S 选项，表示在程序编译期间，在生成汇编代码后，停止，-o 输出汇编代码文件。

c. 汇编

对于上一小节中生成的汇编代码文件 test.s，GAS 汇编器负责将其编译为目标文件，如下：gcc -c test.s -o test.o

d. 连接

GCC 连接器是 GAS 提供的，负责将程序的目标文件与所需的所有附加的目标文件连接起来，最终生成可执行文件。附加的目标文件包括静态连接库和动态连接库。对于上一小节中生成的 test.o，将其与 C 标准输入输出库进行连接，最终生成程序 test。gcc test.o -o test 在命令行窗口中，执行 ./test，让它说 HelloWorld 吧！

4.4.3 GCC 多个程序文件的编译

通常整个程序是由多个源文件组成的，相应地也就形成了多个编译单元，使用 GCC 能够很好地管理这些编译单元。假设有一个由 test1.c 和 test2.c 两个源文件组成的程序，为了对它们进行编译，并最终生成可执行程序 test，可以使用下面这条命令：

```
gcc test1.c test2.c -o test
```

如果同时处理的文件不止一个，GCC 仍然会按照预处理、编译和链接的过程依次进行。如果深究起来，上面这条命令大致相当于依次执行如下三条命令：

```
gcc -c test1.c -o test1.o
gcc -c test2.c -o test2.o
gcc test1.o test2.o -o test
```

（3）检错

```
gcc -pedantic illcode.c -o illcode
```

-pedantic 编译选项并不能保证被编译程序与 ANSI/ISO C 标准的完全兼容，它仅仅只能用来帮助 Linux 程序员离这个目标越来越近。或者换句话说，-pedantic 选项能够帮助程序员发现一些不符合 ANSI/ISO C 标准的代码，但不是全部，事实上只有 ANSI/ISO C 语言标准中要求进行编译器诊断的那些情况，才有可能被 GCC 发现并提出警告。

除了 -pedantic 之外，GCC 还有一些其它编译选项也能够产生有用的警告信息。这些选项大多以-W 开头，其中最有价值的当数 -Wall 了，使用它能够使 GCC 产生尽可能多的警告信息。

```
gcc -Wall illcode.c -o illcode
```

GCC 给出的警告信息虽然从严格意义上说不能算作错误，但却很可能成为错误的栖身之所。 一个优秀的 Linux 程序员应该尽量避免产生警告信息，使自己的代码始终保持标准、健壮的特性。所以将警告信息当成编码错误来对待，是一种值得赞扬的行为！ 所以，在编译程序时带上 -Werror 选项，那么 GCC 会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改，如下：

```
gcc -Werror test.c -o test
```

4.4.4 GCC 库文件连接

a. 使用第三方函数库

开发软件时，完全不使用第三方函数库的情况是比较少见的，通常来讲都需要借助许多函数库的支持才能够完成相应的功能。从程序员的角度看，函数库实际上就是一些头文件（ .h ）和库文件（Linux 下 .a 、 .so ，Windows 下 lib 、 dll ）的集合。虽然 Linux 下的大多数函数都默认将头文件放到 /usr/include/ 目录下，而库文件则放到 /usr/lib/ 目录下；Windows 所使用的库文件主要放在 Visual Studo 的目录下的 include 和 lib ，以及系统文件夹下。但也有的时候，我们要用的库不再这些目录下，所以 GCC 在编译时必须用自己的办法来查找所需要的头文件和库文件。

例如：我们的程序 test.c 是在 Linux 上使用 C 连接 MySQL，这个时候我们需要去 MySQL 官网 下载 MySQL Connectors 的 C 库，下载下来解压之后，有一个 include 文件夹，里面包含 MySQL Connectors 的头文件，还有一个 lib 文件夹，里面包含二进制 .so 文件 libmysqlclient.so 。其中 includle 文件夹的路径是 /usr/dev/mysql/include ， lib 文件夹是 /usr/dev/mysql/lib 。

b. 生成可执行文件

首先我们要进行编译 test.c 为目标文件，这个时候需要执行 gcc -c -I /usr/dev/mysql/include test.c –o test.o

最后我们把所有目标文件链接成可执行文件：gcc –L /usr/dev/mysql/lib –lmysqlclient test.o –o test

Linux 下的库文件分为两大类分别是动态链接库（通常以 .so 结尾）和静态链接库（通常以 .a 结尾），二者的区别仅在于程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。

c. 强制链接时使用静态链接库

默认情况下， GCC 在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在编译时加上 -static 选项，强制使用静态链接库。

在 /usr/dev/mysql/lib 目录下有链接时所需要的库文件 libmysqlclient.so 和 libmysqlclient.a，为了让 GCC 在链接时只用到静态链接库，可以使用下面的命令: gcc –L /usr/dev/mysql/lib –static –lmysqlclient test.o –o test

静态库链接时搜索路径顺序:

1. ld 会去找 GCC 命令中的参数 -L；

2. 再找 gcc 的环境变量 LIBRARY_PATH；

3. 再找内定目录 /lib 、 /usr/lib 和 /usr/local/lib 。

动态链接时、执行时搜索路径顺序:

1. 编译目标代码时指定的动态库搜索路径;

2. 环境变量 LD_LIBRARY_PATH 指定的动态库搜索路径;

3. 配置文件 /etc/ld.so.conf 中指定的动态库搜索路径;

4. 默认的动态库搜索路径 /lib；

5. 默认的动态库搜索路径 /usr/lib 。

有关环境变量:

1. LIBRARY_PATH 环境变量：指定程序静态链接库文件搜索路径;

2. LD_LIBRARY_PATH 环境变量：指定程序动态链接库文件搜索路径。

5 测试

5.1 简介

软件测试是使用人工或自动的手段来运行或测定某个软件系统的过程。其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。从是否关心软件内部结构和具体实现的角度划分，测试方法主要有白盒测试和黑盒测试。作为程序设计语言的实践，对源代码的测试十分重要。

5.2 测试数据

测试数据分很多种。一般而言可以把它们分成小数据、大数据和极限数据三种。

（1）小数据

样例无疑是所有 OIer 的最爱。大家学编程的时候就知道写完程序第一件事就是过样例，样例就是一个典型的小数据。小数据有三大优点：

第一，易于调试。调试模式虽然有助于在短时间内找到程序的问题，但是使用大量的数据十分调试。因此，设计合适的小数据，是找到程序错误的关键。

第二，易于设计。由于数据量小，我们往往可以手工设计质量更高的数据，同时对于数据本身也有直观的了解。很多的题都会有所谓的“变态数据”，这和极限数据有着一些不同，它虽然数据量不大，但是剑走偏锋，比如某矩阵题给你一个全都是 1 的矩阵之类的。这种狡猾的数据在评测的时候往往并不罕见，由于这样那样的原因，我们就栽了跟头。为了使得自己的程序更加强壮，我们需要预先测试自己的程序是否能够通过这样的数据。这种变态数据只能够由我们手工设计，因此一般都是小数据。

第三，覆盖面广。对于很多题目而言，测试数据理论上存在无穷多组；但是如果有 n<5 并且所有数都小于 10 的限制，那么数据的个数就变得有限了，不妨设是 1000 组。我们可以通过写一个程序，直接把这 1000 组小数据全部都制

作出来，然后逐个测试。虽然这些数据的数据量小，但是由于它们把小数据的所有可能的情况都包括在其中了，因此你的程序的大部分问题都能够在这 1000 组数据中有所体现。同时，因为是小数据，程序可以在很短的时间内运行出解，例如是0.05秒，这样，1000组数据，也不过只要50秒，完全可以接受。但是要注意，生成所有数据的同时，我们还要写一个效率差，确保正确的程序来验证结果的正确性。因此这种想法至少需要 2 个程序。（具体操作流程参考“大数据”部分）

（2）大数据

大数据是属于那种数据量比小数据大，同时可以使用较弱的替代算法得到结果的数据。一般的操作流程是这样的：先写一个随机化的制作大数据的程序；然后写一个针对题目的效率较差但是正确性能够保证的使用替代算法的程序；最后使用一个批处理文件，进行多次对比测试，即生成一个数据，然后再比较两个程序的结果。一定要注意这三个程序的文件输入输出和批处理的实现，这些地方很容易出错。

大数据的使用方法和前面讲过的小数据的穷举方法差不多，但是相比之下有些许不同：

第一，数据量不同。数据量变大之后，对程序是一个新的挑战，一些更加难以发现的问题可能会显露出来。

第二，可以随机化。与小数据不同，由于大数据的数据个数过多，不能够穷举完成，因此推荐使用随机化。而随机化显然比穷举要容易编写得多，因此大数据的实现更加方便。而随机化的缺点是，变态数据未必能够随机到。

而与极限数据相比，大数据的优点是可以使用替代算法。极限数据往往不能使用替代算法，因为替代算法往往不能在几秒钟，几分钟甚至几个小时内得出解。

因此大数据是最值得提倡的，我认为如果条件允许，每一题都应该用大数据来测试一下，确保正确性。

（3）极限数据

在很多人的观念里，极限数据是非常重要的一种数据。我发现很多同学在通过样例之后第一个测试的数据就是极限数据。我认为这是一种非常不好的习惯，因为极限数据并没有我们想象的那么重要。我们能够通过极限数据了解到什么呢？无非是我们的程序是否会超时，外加我们的程序是否会越界。事实上一些简单得随机化得到的极限数据或者手工出的带规律的极限数据甚至未必真的能够给我们翔实的信息，因为就算对于这些数据我们的程序能够快速得到结果，我们仍然不知道我们的程序是不是真的不会在处理别的数据时超时或者越界。所谓的极限数据，其实不过是用来测试你的数组有没有越界而已，这才是它的最大的用处。

此外，极限数据容易给你一些虚假的信息。例如你测试一个 1000*1000 的全是 1 的矩阵，结果程序在 0.01 秒内出解了，结果无疑是正确的。于是你得出你的程序不越界、不超时、不错误的结论。一些没有经验的同学甚至可能就不再检查这个程序而努力去完成下一道题。然而这么一个特殊的数据，其实什么问题都说明不了。

因此你又不得不使用特殊数据，因为如果你使用了一个随机的数据，你又不能够确定自己的程序结果是不是正确。

当然，我并没有宣扬以后大家不要再设计极限数据。极限数据仍然有它的重要性。但是它的设计应该是最后一步，在至少设计并测试大数据，确保了程序的正确性之后，再检查程序的极限情况下是不是会超时或越界。一般的测评数据中，极限数据的个数并不会很多，我们不应该因小失大。

对于每一道 OJ 题目，其测试数据的可以有以下几个分类。我们以四个 0~10 之间的整数比大小选取第二大的数为例。所有测试数据一共为 11^4=14641 个。

这些数据可以分为：样例和非样例，其中样例一般是 1 个（或 2 个），非样例有 14640 个。由此可见，只测试样例是远远不够的。

这些数据还可以分为：测试数据和非测试数据。出题人将这 14641 个数据一起作为后台测试一般是不可能的，不仅生成数据费时费力，判题耗时也太长。因此，测试数据一般不多，例如有 10 个，而非测试数据有 14631 个。

这些数据也可以分为：坑和非坑。考虑到编程人员经验所限，有可能在一些想不到的情况处理不好，使得程序在这些数据下出错。坑人的数据也不多，例如有 20 个，而非测试数据有 14621 个。

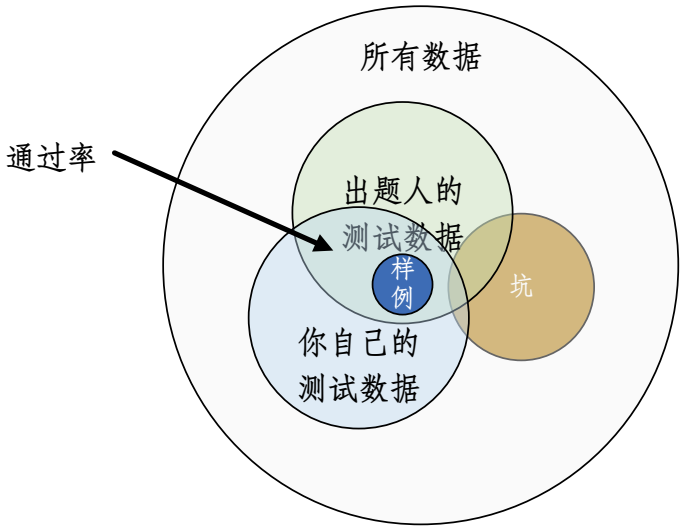
上面几种划分之间没有排除关系。有可能出题人出的 10 个测试数据中，有 2 个是坑人的，1 个样例，还有 7 个普通数据。其中非测试数据中，有 18 个是坑人的，另有 14613 个普通数据。做题人只需要通过 10 个测试数据。

假设你的程序在其中 14600 个数据运行结果正确，在另外 41 个数据运行结果错误。不管这些数据在不在坑中，关键看这几个数据在不在测试数据里。测试数据是怎么样的，除非与出题人窜通，其情况是不可预知的。如果你错误的数据，不慎落在测试数据里，设有 2 个，即无法获得通过，并且通过率显示为 80%。

因此，我建议大家重视测试，对 14641 个数据都进行测试。随着取值范围的不断增加，假设有题目要求的数据范围是 0~1000，则测试数据为 1004006004001 个。每个都测试显然都不现实。但这些数据出错的概率不是均匀分布的。例如你的程序如果在 0 1 2 3 的输入中正确，很难在 0 1 2 4 中出现错误。因此可以将上述测试用例进一步划分测试类，例如考虑到“大于”、“小于”符号的使用，该题测试数据只有次序的区别，分为 4 个在 0~3 的整数组合，共 256 个测试数据。将这 256 个测试数据测试一遍，即可以很大概率获得通过。当然了，有经验的同学可以根据他对自己代码的了解，进一步缩小范围，可以获得更少的测试数据。

但是应该注意，测试类划分错误，或者贪图方便只测试一部分的数据，将增加测试正确、但在 OJ 上答案错误的概率。

同样地，出题人对测试类的划分错误，或测试数据过少，也会以一定概率放过一些有瑕疵的程序。出题人一般会保留 2～3 组比较坑的数据。



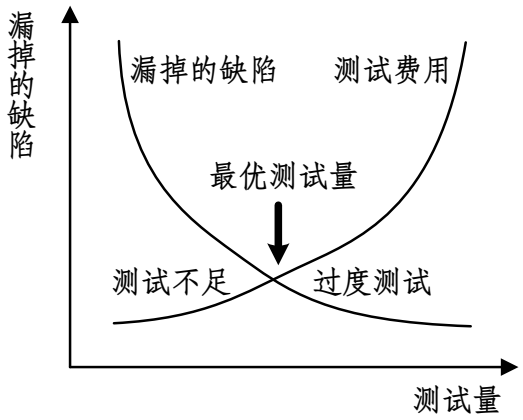
请看上图，通过率是蓝绿重叠部分和绿色部分的面积之比。无法保证上述圆圈面积和位置关系。只要蓝色覆盖了绿色，就能通过。但至少可以知道，一个正常的题目，深蓝色部分一定真包含于绿色部分，绿色部分一定包含于灰色中，且与黄色部分有一些交集。

OJ 做题是唯结果论的，主要分两步：

（1）猜测后台数据是什么；

（2）如何快速找到速度和时间在规定范围内的解题方法。

测试是在测试量和漏掉的缺陷之间做出权衡，从而找到最优的测试量。如果测试不足，测试量过少而漏掉的缺陷过多，提交的程序是错误的，则答题的时间就浪费了。所谓过犹不及，如果测试过度，测试量过多，虽然漏掉的缺陷过少，提交的程序及时正确，但测试浪费掉了大多的时间，影响了规定时间内能做出来的题量。



5.3 测试方法

虽然对测试数据的种类有了一定的了解，但是很多同学也许对如何测试仍有疑惑。测试程序的原始方法是全部手工的，大致可以概括为：手工运行测试数据生成程序、手工运行源程序、手工运行替代算法的程序，最后目测两个程序的结果是否一样。这个方法虽然直观，但是有很多不可忽视的缺点。比如效率太低，操作时容易有较大误差等。

freopen 是将原本需要从键盘（stdin）导入或导出到屏幕（stdout）的文本，重定向到文件中，便于编辑和阅读。因此可以利用 freopen 生成测试输入文件，利用 freopen 将输入文件导入到程序中，利用 freopen 将输出导出到文件中，便于阅读比较。

我们以“寻找第二大的数”题为例，题意大致为寻找给定四个数中第二大的数（即小于最大值的数中最大的数），如果四个数相同则输出该数。

初步测试为样例输入的测试。由于样例输入字符较多，测试又需要反复进行，人工输入耗时费力。可以将测试文件下载下来（“样例输入”标题后有下载图标），在程序中使用“`freopen("in.txt","r",stdin);`”将输入导向文本文件。以后每次运行时，由文本文件向程序供应输入。

类似地，当程序输出较多，而输出到屏幕输出难以阅读比较的时候，可以使用“`freopen("out.txt","w",stdout);`”将输出导向文本文件。便于反复打开使用。

以该题为例，我们应当测试 0123、1023、1032、……等几种情况，但精确地划分测试类需要花费较多的思考时间，因此建议测试 0000~3333 共 4*4*4*4=256 种情形。做题人必须自制输入文件，同样利用 freopen，书写“freopen("in.txt","w",stdout);”并运行程序。即：

```
#include <stdio.h>
int main()
{
	freopen("in.txt","w",stdout);
	for (int i=0; i<4; i++)
		for (int i2=0; i2<4; i2++)
			for (int i3=0; i3<4; i3++)
				for (int i4=0; i4<4; i4++)

	printf("%d %d %d %d\n",i,i2,i3,i4);
}
```

则生成了输入文件，下次运行时，可以作为输入文件使用。

```
#include <stdio.h>
int main()
{
	freopen("in.txt","r",stdin);
	freopen("out.txt","w",stdout);
	（程序主体）
}
```

运行后，逐行检视，最终确定有问题的输入。再将 in.txt 清空，将有问题的输入写入。用断点调试（含：单步调试）调试程序，确定程序在哪里出问题。

我在这里详细地介绍一下使用批处理的方法。假设我们需要测试一个程序 sample.exe，它的输入输出文件分别是 sample.in、sample.out，那么首先我们制作一个数据生成程序 samplemaker.exe，然后写一个用替代算法的程序 sample2.exe，注意这个程序的输入文件是 sample.in，但是输出文件是 sample2.out。最后写一个 txt 文件：

Samplemaker.exe Sample.exe Sample2.exe

Fc sample.out sample2.out Pause

然后把这个 txt 文件的后缀改成 bat，就变成了一个批处理文件了。双击运行这个批处理文件，就可以自动测试，得出两个程序的结果是否相同了。Fc 命令是用来比较两个文件是否相同，pause 表示暂停，以便让我们能够看清楚结果。

但是这个批处理只能够测试一次，我们需要运行它 n 次来进行 n 次的测试。这个效率仍然偏低。提高效率的方法是，在批处理文件中加入循环语句，如下：

```
:loop
Samplemaker.exe Sample.exe Sample2.exe
Fc sample.out sample2.out Pause Goto loop
```

这样的话，这个批处理就能够运行无数次，每次你只需要敲一下键盘就可以了。这样的效率已经很高了，你需要做的不过是盯着 fc 的结果，找到一个不一样的就关掉批处理，找到 sample.in，然后调试。

不过，对于有些题目来说，这样的效率仍然不够，1 秒可能只能测试几个数据。更好的方法应该是，把 pause 删除，然后把 fc 的结果输入进文件，然后写一个程序判断 fc 的结果，如果发现结果不一样，就把 sample.in 记录下来。如下：

```
:loop
Samplemaker.exe Sample.exe Sample2.exe
Fc sample.out sample2.out >result.txt Judge.exe
Goto loop
```

其中，judge.exe 需要我们编写。它应该从 result.txt 中读入数据，判断结果，如果不一样，就进入死循环（例如使用 while true do;语句）。这样你就能够一眼看出有问题出现了，关掉批处理，然后检查 sample.in。

5.4 设计数据的实例

下面我就 NOIP2007 的 4 道题，给大家讲一讲我校部分学生设计测试数据的体会。

（1）count 统计数字

输入一个数 n（n≤200000）和 n 个自然数（每个数都不超过 1.5*109），请统计出这些自然数各自出现的次数，按顺序从小到大输出。输入数据保证不相同的数不超过 10000 个。

本题的算法是很简单的，排序+统计。这里推荐使用快速排序，因为快速排序是众多排序中运行效率最高的一个。对于本题，可以设计三种数据：

1.小数据：小数据应该设计几个变态数据。例如 10 个 0，或者 9，8，7，6，5，4，3，2，1 之类。主要是测试一些不容易随机得到的情况。

2.大数据：本题的代替算法有多个。比较简单的算法应该是冒泡排序或者使用数组下标排序之类。这里推荐使用数组下标排序，因为对于本题的要求（统计次数），这个排序有巨大的优势（编程复杂度很低）。

3.极限数据：由于本题的数据量很大，有必要测试一下程序是否会越界。

（2）expand 字符串的展开

我们可以用减号对连续字母或数字进行缩写，于是字符串 a-dha3-68 就可以展开为 abcdha34568。

输入三个参数 p1，p2，p3，再输入一个仅由数字、小写字母和减号组成的字符串（长度不超过 100），请按参数展开此字符串。

各个参数的意义如下：

参数 p1=1→所有填充的字母都写成小写； 参数 p1=2→所有填充的字母都写成大写；

参数 p1=3→所有填充的字母和数字都用星号代替； 参数 p2=k→同一个填充字符连续写 k 遍； 参数 p3=1→顺序填充； 参数 p3=2→逆序填充。

另外，如果减号两边的字符一个是数字一个是字母，或者减号右边的 ASCII 码没左边的大，则该处不变。

本题是比较简单的模拟，关键是要看清楚题意。本题不适合出大数据或者极限数据，推荐使用静态查错的方法。当然，也可以设计一些有针对性的小数据。

（3）game 矩阵取数游戏

一个 n 行 m 列的矩阵，每次你需要按要求取出 n 个数，m 次正好将所有数取完。每取出一个数你都会有一个得分，请求出最终的得分最大是多少。

每一次取数的要求：每一行中恰好取一个数，且只能取剩下的数中最左边或最右边位置上的数。

每取一个数的得分：所取数的数值乘以 2i，i 表示这是第 i 轮取数。 矩阵中的数为不超过 100 的自然数，1≤n，m≤80。 本题是比较简单的动态规划。因为是动态规划，一般很少有较好的替代算法。由于行和列之间没有必然的联系，因此我们完全可以考虑 n=1 的情况。这样的话，可能使用 2m 的穷举算法，这样 m 可以到 20 左右。因此对于本题，可以设计：

1.小数据：针对一些特殊情况，例如所有的数据都是 0，或者所有的数都是 1 的情况；

2.大数据：使用穷举算法加随机化生成数据，可以测试程序的正确性； 3. 极限数据：测试程序是否会超时。

（4）core 数网的核

树上的任两点间都有唯一路径。定义某一点到树上某一路径的距离为该点到路径上所有点的路径长度中的最小值。定义树中某条路径的“偏心距”为所有其他点到此路径的距离的最大值。定义树的直径为树的最大路径（可能不唯一）。给出一个有 n 个节点的无根树，请找出某个直径上的一段长度不超过 s 的路径（可能退化为一个点），使它的偏心距最小。请输出这个最小偏心距的值。

题目已经告诉你如下定理：树的所有直径的中点必然重合（这个中点可能在某条边上）。其实这个结论很明显嘛，因为如果中点不重合的话必然可以找到一条更长的路。

5≤n≤300，0≤s≤1000，边权是不超过 1000 的正整数。

相对于前面的三题而言，本题有一定难度。本题告诉了我们很多条件，因此正确的算法有很多，例如 dfs 或者 bfs 加穷举等，这里不作讨论。由于本题与树有关，因此并不容易手工出数据。对于本题，可以设计：

（1）大数据：随机化可以出一些大数据，但是一个好的替代算法不容易想到。我觉得穷举还算不错的想法。可以使用 n2 的时间穷举长度不超过 s 的路径，然后再使用 n 的时间穷举别的点，再用 n 的时间算出距离。这样 n 可以大概到 100 左右，算是不错的算法了。

（2）极限数据：出不出都可以，因为只要算法恰当，本题应该不会超时或者越界。

5.5 注意事项

最后笔者还要强调一些问题。

第一，不能够做错测试数据。测试数据生成程序时非常容易写的，手工出测试数据也很轻松，但是不能够轻视出数据这一个过程。数据的格式、范围以及是否合法都非常重要。如果你不慎出错了数据，你很可能认为不是数据的问题，而是程序的问题，然后花费大量的时间检查自己的程序。这无疑是非常不值得的。

第二，测试数据虽然是非常优秀的检查程序的方法，但是不能够过于依赖，静态查错仍然非常重要。对于小数据，也许还可以轻松跟踪调试，但是大数据或者极限数据就需要花费大量时间了。因此，笔者着重推荐静态查错的方法。

第三，打字速度和编程熟练程度就变得非常重要。认真看完本文就会发现，对于某一个程序，写三四个小程序来测试它其实一点都不过分或者夸张。一个

一小时能够写 10 个程序的选手和一个一小时只能写 3 个程序的选手对于算法的理解可能是旗鼓相当的，但是程序的正确率绝对会有天壤之别。冰冻三尺，非一日之寒，勤加练习才是确保在比赛中取得好成绩的根本。

6 静态检查

6.1 简介

代码检查可以有效的提高代码质量，更进一步的说代码检查不仅仅是为了提高代码质量，已深入到代码程序的逻辑检查、内存使用情况的检查甚至更高层面的检查，很大程度上影响了程序的功能和性能。

代码检查分为：

（1）动态检查：程序运行时检查，侧重于内存和资源使用情况检查；

（2）静态检查：指不运行被测程序本身，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性。

6.2 静态检查的优点

代码静态检查带来的好处：

第一，帮助程序开发人员自动执行静态代码分析，快速定位代码隐藏错误和缺陷；

第二，帮助代码设计人员更专注于分析和解决代码设计缺陷；

第三，显著减少在代码逐行检查上花费的时间，提高软件可靠性并节省软件开发和测试成本。

6.3 可检查的问题

静态检查可以检测的问题有：变量未初始化；空指针引用（野指针）；数据类型不匹配；返回局部变量；数组越界；内存泄漏等。

如下实例，通过静态检查工具可以检查的错误信息（这些问题在代码编译的时候可能不会出现）。

```
#include <stdio.h>

int main (int argc, char **argv)
```

```
{
 char  cBuf[10] = { 0 };
 char *pTemp;
 int   i;
       for (i = 0; i <= 10; i++) {
     cBuf[i] = 0;
    }
       printf("output %s\n", pTemp);
       return  (0);
}
```

编译后没有出现报错提示，然后运行静态检查，会出现如下图所示报错现象。

6.4 常用工具

静态检测工具种类很多，下面介绍几种常见的检测工具。

1、cppcheck

Cppcheck 是一种 C/C++代码缺陷静态检查工具，不同于 C/C++编译器及其它分析工具，Cppcheck 只检查编译器检查不出来的 bug，不检查语法错误，作为编译器的一种补充检查，cppcheck 对产品的源代码执行严格的逻辑检查。 执行的检查包括：

1.自动变量检查

2.数组的边界检查

3.class 类检查

4.过期的函数，废弃函数调用检查

5.异常内存使用，释放检查

6.内存泄漏检查，主要是通过内存引用指针

7.操作系统资源释放检查，中断，文件描述符等

8.异常 STL 函数使用检查

9.代码格式错误，以及性能因素检查

2、pc-lint

PC-Lint 是 GIMPEL SOFTWARE 公司开发的 C/C++软件代码静态分析工具，它的全称是 PC-Lint/FlexeLint for C/C++，PC-Lint 能够在 Windows、MS-DOS 和 OS/2 平台上使用，以二进制可执行文件的形式发布，而 FlexeLint 运行于其它平台，以源代码的形式发布。PC-lint 在全球拥有广泛的客户群，许多大型的软件开发组织都把 PC-Lint 检查作为代码走查的第一道工序。

PC-Lint 不仅能够对程序进行全局分析，识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针以及冗余的代码，还能够有效地帮你提出许多程序在空间利用、运行效率上的改进点。

pc-lint 提供的检测类型：

1.强类型检查

2.变量值跟踪

3.赋值顺序检查

4.弱定义检查

5.格式检查

6.缩进检查

7.const 变量检查

8.volatile 变量检查

3、splint

针对 C 语言的开源程序静态分析工具 splint(原来的 LCLint)， 是一个 GNU 免费授权的 Lint 程序，是一个动态检查 C 语言程序安全弱点和编写错误的程序。Splint 会进行多种常规检查,包括未使用的变量,类型不一致,使用未定义变量,无法执行的代码,忽略返回值,执行路径未返回,无限循环等错误。

splint 提供的检测类型有：

1.解引用空指针(Null Dereferences)

解引用空指针将导致我们在程序运行时产生段错误(Segmentation fault)。

2.类型(Types)

我们在编程中经常用到强制类型转换，将有符号值转换为无符号值、大范围类型值赋值给小范围类型，程序运行的结果会出无我们的预料

3.内存管理(Memory Management)

C 语言程序中，将近半数的 bug 归功于内存管理问题，关乎内存的 bug 难以发现并且会给程序带来致命的破坏。

4.缓存边界(Buffer Sizes)

splint 会对数组边界、字符串边界作检测，使用时需要加上+bounds 的标志。

7 调试

7.1 简介

程序调试是将编制的程序投入实际运行前，用手工或编译程序等方法进行测试，修正语法错误和逻辑错误的过程。这是保证计算机信息系统正确性的必不可少的步骤。编完计算机程序，必须送入计算机中测试。根据测试时所发现的错误，进一步诊断，找出原因和具体的位置进行修正。

7.2 步骤

第一步，用编辑程序把编制的源程序按照一定的书写格式送到计算机中，编辑程序会根据使用人员的意图对源程序进行增、删或修改。

第二步，把送入的源程序翻译成机器语言，即用编译程序对源程序进行语法检查并将符合语法规则的源程序语句翻译成计算机能识别的“语言”。如果经编译程序检查，发现有语法错误，那就必须用编辑程序来修改源程序中的语法错误，然后再编译，直至没有语法错误为止。

第三步，使用计算机中的连接程序，把翻译好的计算机语言程序连接起来，并扶植成一个计算机能真正运行的程序。在连接过程中，一般不会出现连接错误，如果出现了连接错误，说明源程序中存在子程序的调用混乱或参数传递错误等问题。这时又要用编辑程序对源程序进行修改，再进行编译和连接，如此反复进行，直至没有连接错误为止。

第四步，将修改后的程序进行试算，这时可以假设几个模拟数据去试运行，并把输出结果与手工处理的正确结果相比较。如有差异，就表明计算机的程序存在有逻辑错误。如果程序不大，可以用人工方法去模拟计算机对源程序的这几个数据进行修改处理；如果程序比较大，人工模拟显然行不通，这时只能将计算机设置成单步执行的方式，一步步跟踪程序的运行。一旦找到问题所在，仍然要用编辑程序来修改源程序，接着仍要编译、连接和执行，直至无逻辑错误为止。也可以在完成后再进行编译。

7.3 原则

一，用头脑去分析思考与错误征兆有关的信息。

二，避开死胡同。

三，只把调试工具当做手段。利用调试工具，可以帮助思考，但不能代替思考，因为调试工具给的是一种无规律的调试方法。

四，避免用试探法，最多只能把它当做最后手段。

五，在出现错误的地方，可能还有别的错误。

六，修改错误的一个常见失误是只修改了这个错误的征兆或这个错误的表现，而没有修改错误本身。如果提出的修改不能解释与这个错误有关的全部线索，那就表明只修改了错误的一部分。

七，注意修正一个错误的同时可能会引入新的错误。

八，修改错误的过程将迫使人们暂时回到程序设计阶段。修改错误也是程序设计的一种形式。

九，修改源代码程序，不要改变目标代码。

7.4 方法

一，简单调试方法：步骤

1，在程序中插入打印语句、优点是能够显示程序的动态过程，比较容易检查源程序的有关信息。缺点是效率低，可能输入大量无关的数据，发现错误带有偶然性。

2，运行部分程序。有时为了测试某些被怀疑有错的程序段，却将整个程序反复执行许多次，在这种情况下，应设法使被测程序只执行需要检查的程序段，以提高效率。

3，借助调试工具。大多数程序设计语言都有专门的调试工具，可以用这些工具来分析程序的动态行为。

二，回溯法排错。确定最先发现错误症状的地方，人工沿程序的控制流往回追踪源程序代码，直到找到错误或范围。

三，归纳法排错。是一种系统化的思考方法，是从个别推断全体的方法，这种方法从线索（错误征兆出发），通过分析这些线索之间的关系找出故障。主要有 4 步：

（1）收集有关数据。收集测试用例，弄清测试用例观察到哪些错误征兆，以及在什么情况下出现错误等信息。

（2）组织数据。整理分析数据，以便发现规律，即什么条件下出现错误，什么条件下不出现错误。

（3)导出假设。分析研究线索之间的关系，力求找出它们的规律，从而提出关于错误的一个或多个假设，如果无法做出假设，则应设计并执行更多的测试用例，以便获得更多的数据。

（4）证明假设。假设不等于事实，证明假设的合理性是极其重要的，不经证明就根据假设排除错误，往往只能消除错误的征兆或只能改正部分错误。证明假设的方法是用它解释所有原始的测试结果，如果能圆满地解释一切现象，则假设得到证明，否则要么是假设不成立或不完备，要么是有多个错误同时存在。

四，演绎法排错。设想可能的原因，用已有的数据排除不正确的假设，精化并证明余下的假设。

五、对分查找法。如果知道每个变量在程序内若干个关键点上的正确值，则可用赋值语句或输入语句在程序中的关键点附近“注入”这些变量的正确值，然后检查程序的输出。如果输出结果是正确的，则表示错误发生在前半部分，否则，不妨认为错误在后半部分。这样反复进行多次，逐渐逼近错误位置。

7.5 分类

调试主要有两种：静态调试和动态调试。

静态调试可以采用如下两种方法：

（1）输出寄存器的内容。在测试中出现问题，设法保留现场信息。把所有寄存器和主存中有关部分的内容打印出来（通常以八进制或十六进制的形式打印），进行分析研究。用这种方法调试，输出的是程序的静止状态（程序在某一时刻的状态），效率非常低，不得已时才采用。

（2）为取得关键变量的动态值，在程序中插入打印语句。这是取得动态信息的简单方法，并可检验在某时间后某个变量是否按预期要求发生了变化。此方法的缺点是可能输出大量需要分析的信息，必须修改源程序才能插入打印语句，这可能改变关键的时序关系，引入新的错误。

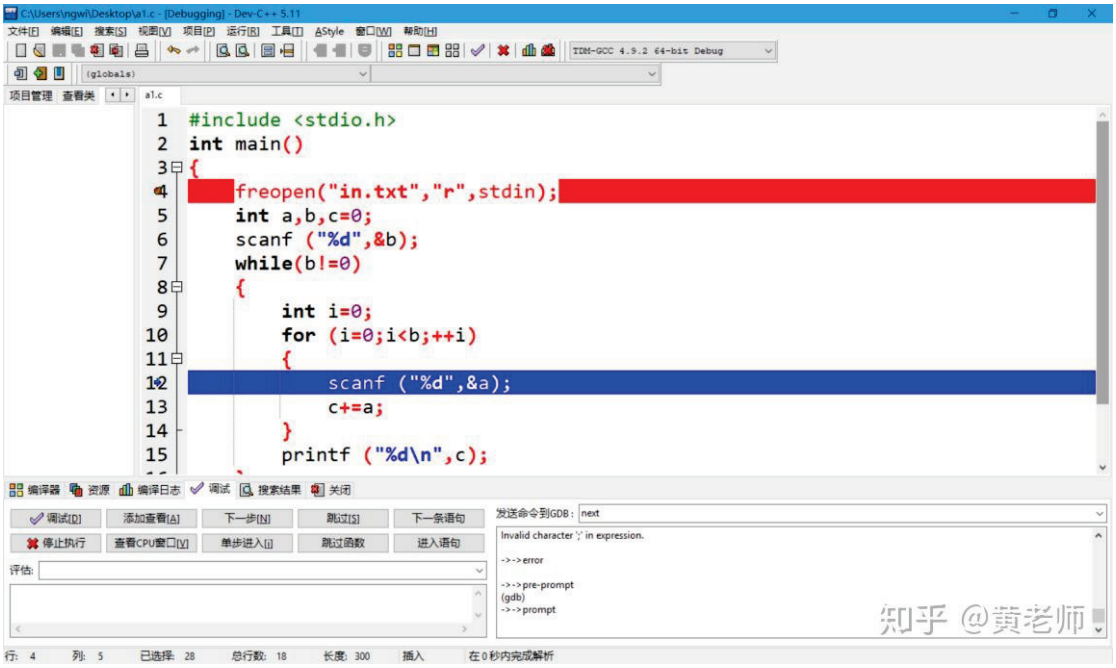
动态调试通常是利用程序语言提供的调试功能或专门的调试工具来分析程序的动态行为。一般程序语言和工具提供的调试功能有检查主存和寄存器；设置断点，即当执行到特定语句或改变特定变量的值时，程序停止执行，以便分析程序此时的状态。

调试主要有两种：**断点调试**和**日志调试**。在较小的代码里，或者在代码的某个细节，断点调试较为有效；对存在多次迭代循环的算法题和工程调试，日志调试较为有效。

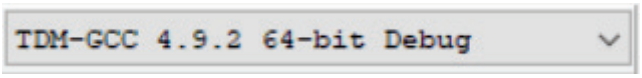
对 OJ 程序走向有疑问的同学可以使用单步调试和断点调试功能，了解程序的走向。

方法如下：

第一步，输入你的程序，在你程序的声明语句之后（如果 C99 允许在 main 函数的第一行），加上“freopen("in.txt","r",stdin);”，并在对应 in.txt 中输入你的测试输入，这样你就不必和输入搅在一起，更专心地调试程序了。



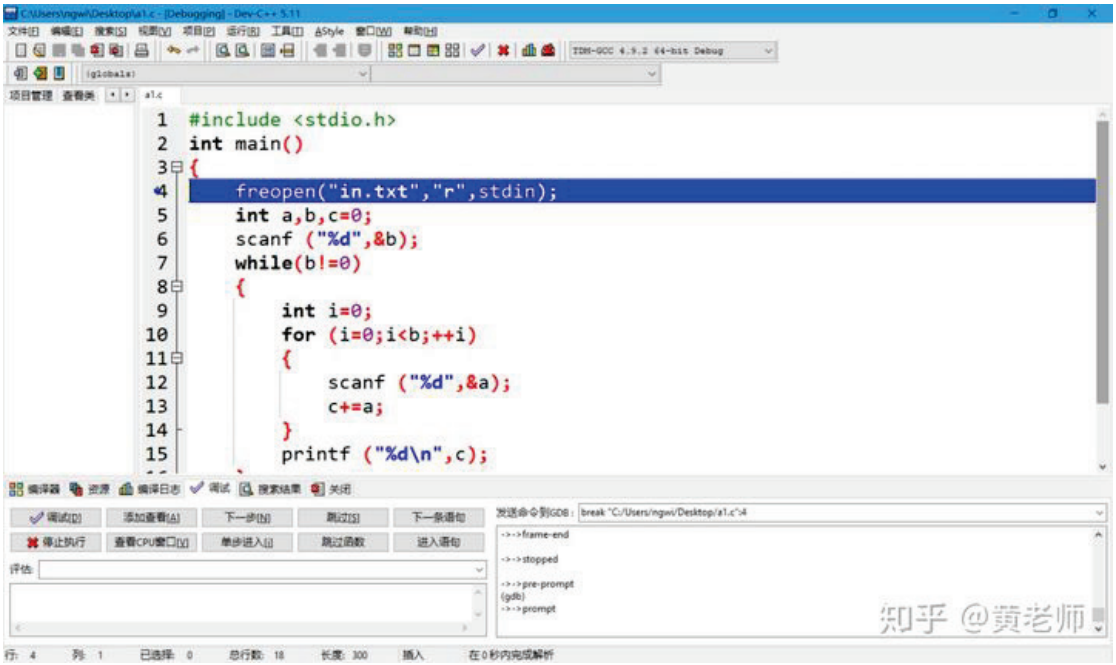
第二步，设置断点，在 freopen 这一行按下鼠标左键或者按下 F4 设置断点。同时，右上角的编译选项应改为 64bit。



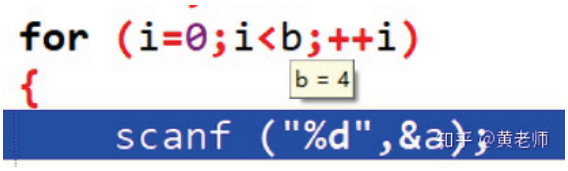
注意：32bit 模式也可以，但一定要是 Debug，不是 Release 或者其它。

第三步，编译并以调试模式运行程序。先按下 F9 编译程序，再按 F5 调试程序。

第四步，实施调试。蓝色语句是断点所在位置，也就是程序运行到这一行之上一行。单击左下角工具箱中的按钮，可以分别实施操作。例如单击“下一步”按钮，可以看下一步程序执行的是哪一条语句。



如果你要查看变量的值，可以将鼠标停靠在其上方查看。



或者发送命令到 GDB。例如输入：p i 并回车（意思为 print i 的值）右下角窗口第一行即出现 0（意思是 i 为 0）。

7.6 单步调试

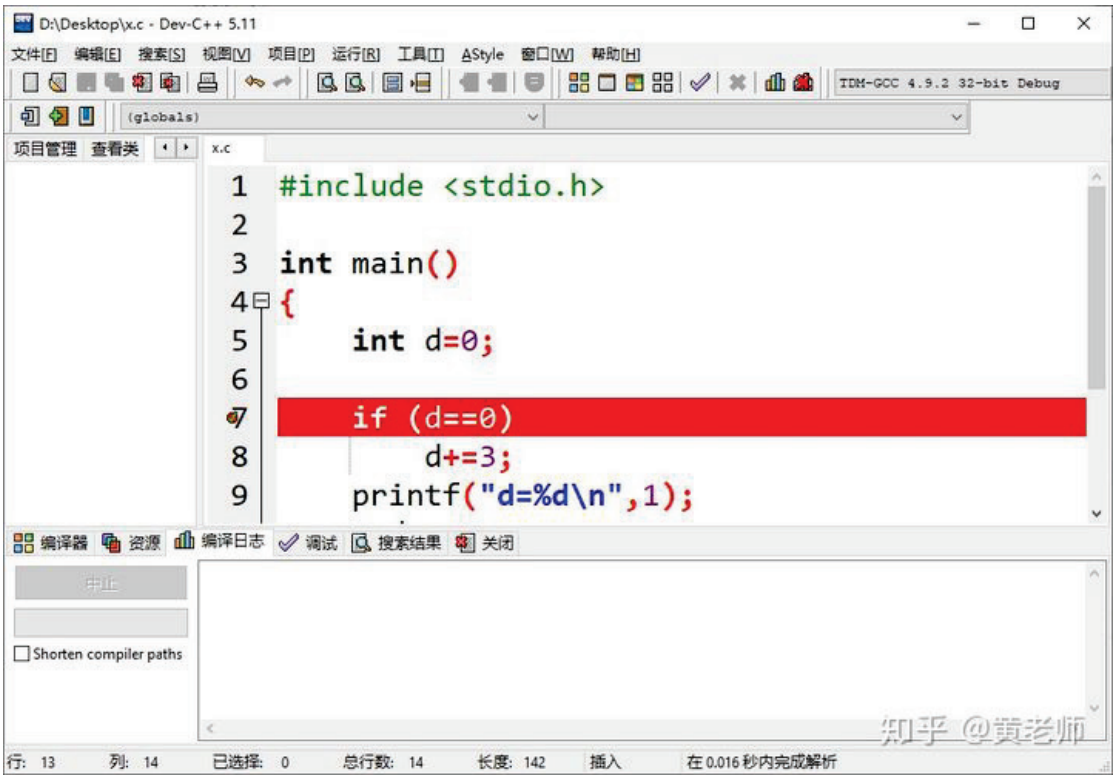
单步调试是**断点调试**的一种。断点调试要求程序按运行路径暂停在用户预先设定的行。单步调试在调试时，程序暂停在主函数的第一行（除了声明语句外），然后，每次运行只执行一行，程序暂停在新的一行。

第一步，建立一个 C 源代码文件。

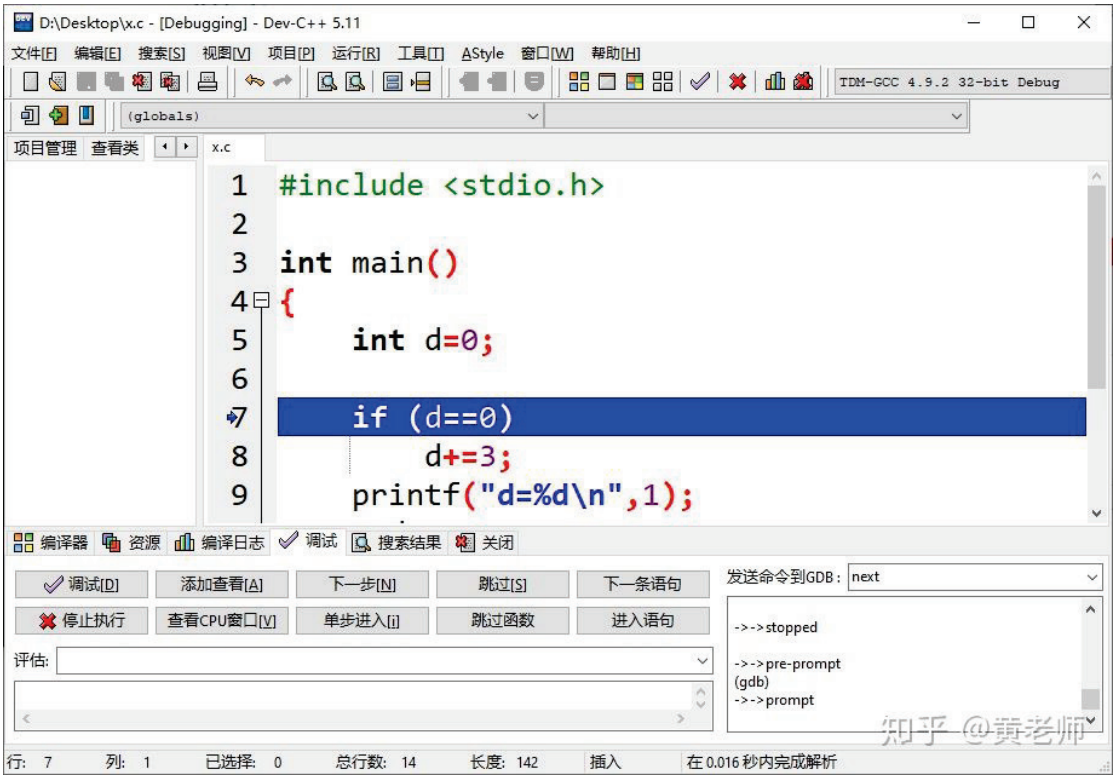
```
#include <stdio.h>

int main()
{
	int d=0;

	if (d==0)
		d+=3;
```

```
        printf("d=%d\n",1);
        --d;
        printf("d=%d\n",2);

        return 0;
}
```

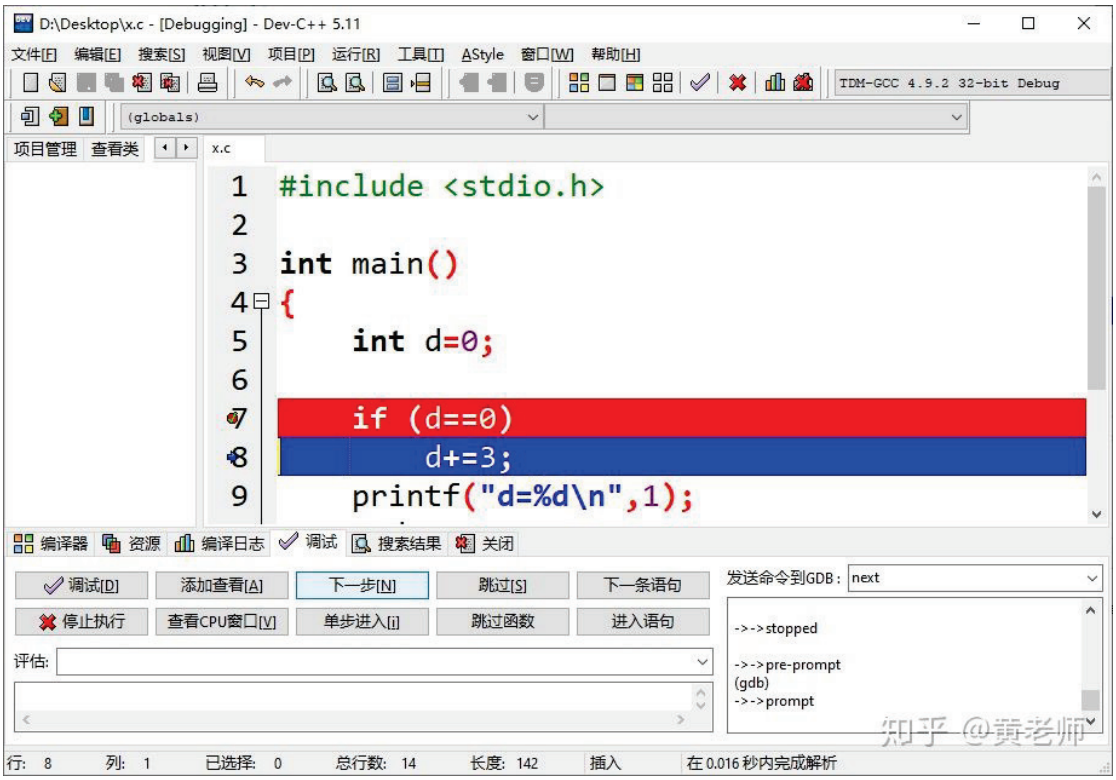
第二步，点击某一行的行号，或将光标移动到某一行，单击 F4 切换断点状态。红色底色行是断点行。



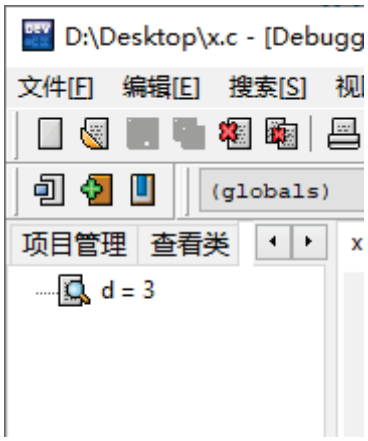
第三步，按下 F5 进行调试。程序会暂停在第一个断点上。该暂停位置为蓝色底色。注意**程序运行到该行之前一句**，而**当前行是未被执行**的。



第四步，单击窗口下方的“下一步[N]”按钮，程序运行到下一句。如示例，d==0 成立，程序应该是断在第 8 行 d+=3;一句。



将鼠标放在变量上，则可以看到变量的值。选定一个变量，右键菜单或下方按钮中，单击“添加查看”，也可以看到变量的值。再次强调，是该行语句运行前的值，即使有的程序员会在一行写入多条语句（**这是个坏习惯**）。



在该行调用了其它已有源代码的函数（printf 函数在没有获得源代码的情况下是无法进入的）时，也可以单击“单步进入”，进行被调用函数的第一句。

7.7 日志式调试

题目：输入a,b,c,d，$0 \le a,b,c,d \le 1000$。输出其中的第二大数，如果四个数相等，则输出其中一个数。

某同学提交的程序是：

```
#include <stdio.h>
int main(void)
{
    int a, b, c, d, x, y, z, p;
    while (scanf("%d %d %d %d", &a, &b, &c, &d) != EOF)
    {
        if (a <= b)
        {
            x = a;
            a = b;
            b = x;
        }
        if (c <= d)
        {
            y = c;
            c = d;
            d = y;
```

```
        }
        if (a >= c)
        {
            z = a;
            a = c;
            c = z;
        }
        if (b <= d)
        {
            p = b;
            b = d;
            d = p;
        }
        if (a <= b)
            a = b;
        printf("%d\n", a);
    }
    return 0;
}
```

样例输入为：

```
1 2 3 4
2 3 4 5
7 3 2 1
1 1 1 1
5 5 5 3
```

样例输出为：

```
3
4
3
1
3
```

但该同学的输出为：

```
3
4
1
3
5
```

此时，先确定程序的几个重要位置，在重要位置先加上打印__LINE__和状态（a, b, c, d, x, y, z, p 的值）。

程序为：

```
#include <stdio.h>
int main(void)
{
    int a, b, c, d, x, y, z, p;
    while (scanf("%d %d %d %d", &a, &b, &c, &d) != EOF)
    {
        printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__, a,
b, c, d, x, y, z, p);
        if (a <= b)
        {
            x = a;
            a = b;
            b = x;
            printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__,
a, b, c, d, x, y, z, p);
        }
        if (c <= d)
        {
            y = c;
            c = d;
            d = y;
            printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__,
a, b, c, d, x, y, z, p);
        }
        if (a >= c)
        {
            z = a;
            a = c;
            c = z;
            printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__,
a, b, c, d, x, y, z, p);
        }
        if (b <= d)
        {
            p = b;
            b = d;
            d = p;
            printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__,
a, b, c, d, x, y, z, p);
        }
        if (a <= b)
        {
```

```
            printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__,
a, b, c, d, x, y, z, p);
            a = b;
        }
        printf("%d\n", a);
        printf("%d: %d %d %d %d %d %d %d %d\n", __LINE__, a,
b, c, d, x, y, z, p);
    }
    return 0;
}
```

输入：

```
5 5 5 3
```

输出：

```
8: 5 5 5 3 1 3 0 1
14: 5 5 5 3 5 3 0 1
28: 5 5 5 3 5 3 5 1
39: 5 5 5 3 5 3 5 1
5
```

即先运行第 8 行，然后第 14、28、39 行。

说明：`(a <= b)` 、`(a >= c)`、`(a <= b)`，但这样的第 2 个数（b）不是第二大的数。这是整个程序的问题，应再补强。

7.8 GDB 调试器

7.8.1 概述

GDB 是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX 下的软件全是命令行的，这给程序开发提代供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。

于是 UNIX 下的软件比 Windows 下的软件更能有机地结合，各自发挥各自的长处，组合成更为强劲的功能。而 Windows 下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。在这里并不是要和 Windows 做

个什么比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具。或许，各位比较喜欢那种图形界面方式的，像 VC、BCB 等 IDE 的调试，但如果你是在 UNIX 平台下做软件，你会发现 GDB 这个调试工具有比 VC、BCB 的图形化调试器更强大的功能。所谓“寸有所长，尺有所短”就是这个道理。

一般来说，GDB 主要帮忙你完成下面四个方面的功能：

1、启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。

2、可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）

3、当程序被停住时，可以检查此时你的程序中所发生的事。

4、动态的改变你程序的执行环境。

从上面看来，GDB 和一般的调试工具没有什么两样，基本上也是完成这些功能，不过在细节上，你会发现 GDB 这个调试工具的强大，大家可能比较习惯了图形化的调试工具，但有时候，命令行的调试工具却有着图形化工具所不能完成的功能。让我们一一看来。

7.8.2 GDB 调试示例

源程序：tst.c

```
#include <stdio.h>

int func(int n)
{
    int sum=0,i;
    for(i=0; i%lt;7; i++)
    {
        sum+=i;
    }
    return sum;
}
```

```
main()
{
    int i;
    long result = 0;
    for(i=1; i<=100; i++)
    {
        result += i;
    }

    printf("result[1-100] = %d \n", result );
    printf("result[1-250] = %d \n", func(250) );
}
```

编译生成执行文件：（Linux 下）gcc -g tst.c -o tst，使用 GDB 调试：gdb tst

```
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-suse-linux"...
(gdb)
```

输入 l 。l 命令相当于 list，从第一行开始列出源代码。

```
1 #include <stdio.h>
2
3 int func(int n)
4 {
5     int sum=0,i;
6     for (i=0; i < 7; i++) {
8         sum+=i;
9 }
10 return sum;
(gdb)
```

输入回车键 Enter，重复上一次命令。

```
11 }

12
13
14 main()
15 {
16     int i;
17     long result = 0;
18     for(i=1; i<=100; i++)
19     {
20         result += i;
(gdb)
```

输入 break 16，设置断点，在源程序第 16 行处。

```
Breakpoint 1 at 0x8048496: file tst.c, line 16.
(gdb)
```

也可以输入 break func，设置断点，在函数 func() 入口处。

```
Breakpoint 2 at 0x8048456: file tst.c, line 5.
(gdb)
```

随时可以输入 info break，查看断点信息。

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x08048496 in main at tst.c:16
2 breakpoint keep y 0x08048456 in func at tst.c:5
(gdb)
```

输入 r 运行程序，r 是 run 命令简写 。

```
Starting program: /home/hchen/test/tst
Breakpoint 1, main () at tst.c:17
17 long result = 0;
(gdb)
```

这句话表明，在断点处停住。此时，可以输入 n 单条语句执行，n 是 next 命令简写。

```
18 for(i=1; i<=100; i++)
```

```
(gdb)
```

可以继续输入 n，查看程序的运行轨迹是不是自己所期望的。

```
20 result += i;
(gdb)
```

可以多次输入 n。

```
18 for(i=1; i<=100; i++)
(gdb) n
20 result += i;
(gdb)
```

可以输入 c，继续运行程序，c 是 continue 命令简写。

```
Continuing.
result[1-100] = 5050
```

这是程序的输出。

```
Breakpoint 2, func (n=250) at tst.c:5
5 int sum=0,i;
(gdb)
```

可以多次输入 n。

```
6 for(i=1; i<=n; i++)
(gdb)
```

可以输入 p i 打印变量 i 的值，p 是 print 命令简写。

```
$1 = 134513808
(gdb) n
8 sum+=i;
(gdb) n
6 for(i=1; i<=n; i++)

(gdb) p sum
$2 = 1
(gdb) n
8 sum+=i;
(gdb) p i
$3 = 2
```

```
(gdb) n
6 for(i=1; i<=n; i++)
(gdb) p sum
$4 = 3
(gdb)
```

可以输入 bt 查看函数堆栈。

```
#0 func (n=250) at tst.c:5
#1 0x080484e4 in main () at tst.c:24
#2 0x400409ed in __libc_start_main () from /lib/libc.so.6
(gdb)
```

可以输入 finish 退出函数。

```
Run till exit from #0 func (n=250) at tst.c:5
0x080484e4 in main () at tst.c:24
24 printf("result[1-250] = %d /n", func(250) );
Value returned is $6 = 31375
(gdb)
```

可以输入 c 继续运行。

```
Continuing.
result[1-250] = 31375
```

这是程序输出。

```
Program exited with code 027.
```

程序退出，调试结束。此时也可以输入 q 退出 GDB。

好了，有了以上的感性认识，还是让我们来系统地认识一下 gdb 吧。

7.8.3 GDB 的命令

(a) 使用 GDB

一般来说 GDB 主要调试的是 C/C++的程序。要调试 C/C++的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（cc/gcc/g++）的 -g 参数可以做到这一点。

如：`cc -g hello.c -o hello` 或 `g++ -g hello.cpp -o hello`

如果没有 -g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当你用-g 把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用 GDB 来调试他。

启动 GDB 的方法有以下几种：

1、gdb program：其中 program 是你的执行文件，一般在当然目录下。

2、gdb core：用 GDB 同时调试一个运行程序和内核文件，即 core 是程序非法执行后 core dump 产生的文件。

3、gdb：如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程 ID。GDB 会自动附加（attach）上去，并调试它。其中 program 应该在 PATH 环境变量中搜索得到。

GDB 启动时，可以加上一些 GDB 的启动开关，详细的开关可以用 gdb -help 查看。我在下面只例举一些比较常用的参数：

-symbols 或 -s，从指定文件中读取符号表。

-se file，从指定文件中读取符号表信息，并把他用在可执行文件中。

-core 或 -c，调试 core dump 的 core 文件。

-directory 或 -d，加入一个源文件的搜索路径。默认搜索路径是环境变量中 PATH 所定义的路径。

(b) GDB 的命令概貌

启动 GDB 后，就你被带入 GDB 的调试环境中，就可以使用 gdb 的命令开始调试程序了，GDB 的命令可以使用 help 命令来查看，如下所示：

```
$ gdb
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
```

```
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-suse-linux".
(gdb) help

List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping
the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in
that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

GDB 的命令很多，GDB 把之分成许多个种类。help 命令只是列出 gdb 的命令种类，如果要看种类中的命令，可以使用 help 命令，如： help breakpoints，查看设置断点的所有命令。也可以直接 help 来查看命令的帮助。

GDB 中，输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该要标志着一个唯一的命令，在 Linux 下，你可以敲击两次 TAB 键来补齐命令的全称，如果有重复的，那么 gdb 会把其列出来。

示例一：在进入函数 func 时，设置一个断点。可以敲入 break func，或是直接就是 b func

```
(gdb) b func
```

```
Breakpoint 1 at 0x8048458: file hello.c, line 10.
```

示例二：敲入 b 按两次 TAB 键，你会看到所有 b 打头的命令：

```
(gdb) b
backtrace break bt
(gdb)
```

示例三：只记得函数的前缀，可以这样：

```
(gdb) b make_ TAB

```

（再按下一次 TAB 键，你会看到:）

```
make_a_section_from_file make_environ
make_abs_section make_function_type
make_blockvector make_pointer_type
make_cleanup make_reference_type
make_command make_symbol_completion_list
(gdb) b make_
```

GDB 把所有 make 开头的函数全部列出来给你查看。

示例四：调试 C++ 的程序时，函数名可以重载。如：

```
(gdb) b 'bubble( M-?
bubble(double,double) bubble(int,int)
(gdb) b 'bubble(
```

你可以查看到 C++ 中的所有的重载函数及参数。（注： M-? 和“按两次 TAB 键”是一个意思）

要退出 GDB 时，只用发 quit 或命令简称 q 就行了。

(c) GDB 中运行 shell 程序

在 GDB 环境中，你可以执行 UNIX 的 shell 的命令，使用 GDB 的 shell 命令来完成，它调用 UNIX 的 shell 来执行，环境变量 SHELL 中定义的 UNIX 的 shell 将会被用来执行，如果 SHELL 没有定义，那就使用 UNIX 的标准 shell ： /bin/sh 。（在 Windows 中使用 Command.com 或 cmd.exe 。）

还有一个 GDB 命令是 make ：可以在 GDB 中执行 make 命令来重新编译自己的程序。这个命令等价于 shell make 。

(d) 在 GDB 中运行程序

当以 gdb 方式启动 gdb 后，gdb 会在 PATH 路径和当前目录中搜索的源文件。如要确认 gdb 是否读到源文件，可使用 l 或 list 命令，看看 gdb 是否能列出源代码。

在 gdb 中，运行程序使用 r 或是 run 命令。程序的运行，你有可能需要设置下面四方面的事。

1、程序运行参数。

set args 可指定运行时参数。（如：set args 10 20 30 40 50 ）

show args 命令可以查看设置好的运行参数。

2、运行环境。

path 可设定程序的运行路径。

show paths 查看程序的运行路径。

set environment varname [=value] 设置环境变量。如：set env USER=hchen

show environment [varname] 查看环境变量。

3、工作目录。

cd 相当于 shell 的 cd 命令。

pwd 显示当前的所在目录。

4、程序的输入输出。

info terminal 显示你程序用到的终端的模式。

使用重定向控制程序输出。如：run > outfile

tty 命令可以指写输入输出的终端设备。如：tty /dev/ttyb

(e) 调试已运行的程序

两种方法：

1、在 UNIX 下用 ps 查看正在运行的程序的 PID（进程 ID），然后用 gdb PID 格式挂接正在运行的程序。

2、先用 gdb 关联上源代码，并进行 gdb，在 gdb 中用 attach 命令来挂接进程的 PID。并用 detach 来取消挂接的进程。

(f) 暂停 / 恢复程序运行

调试程序中，暂停程序运行是必须的，GDB 可以方便地暂停程序的运行。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停往等等。以便于你查看运行时的变量，以及运行时的流程。

当进程被 gdb 停住时，你可以使用 info program 来查看程序的是否在运行，进程号，被暂停的原因。

在 gdb 中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（WatchPoint）、捕捉点（CatchPoint）、信号（Signals）、线程停止（Thread Stops）。如果要恢复程序运行，可以使用 c 或是 continue 命令。

一、设置断点（BreakPoint）

我们用 break 命令来设置断点。正面有几点设置断点的方法：

break 在进入指定函数时停住。C++ 中可以使用 class::function 或 function(type,type)格式来指定函数名。

break 在指定行号停住。

break +offset

break -offset

在当前行号的前面或后面的 offset 行停住。offiset 为自然数。

break filename:linenum

在源文件 filename 的 linenum 行处停住。

break filename:function

在源文件 filename 的 function 函数的入口处停住。

break *address

在程序运行的内存地址处停住。

break

break 命令没有参数时，表示在下一条指令处停住。

break ... if

...可以是上述的参数，condition 表示条件，在条件成立时停住。比如在循环境体中，可以设置 break if i=100，表示当 i 为 100 时停住程序。

查看断点时，可使用 info 命令，如下所示：（注：n 表示断点号）

info breakpoints [n]

info break [n]

二、设置观察点（WatchPoint）

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

watch 为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停住程序。

rwatch 当表达式（变量）expr 被读时，停住程序。

awatch 当表达式（变量）的值被读或被写时，停住程序。

info watchpoints 列出当前所设置了的所有观察点。

三、设置捕捉点（CatchPoint）

你可设置捕捉点来补捉程序运行时的一些事件。如：载入共享库（动态链接库）或是 C++的异常。设置捕捉点的格式为：

catch 当 event 发生时，停住程序。event 可以是下面的内容：

1、throw 一个 C++抛出的异常。（throw 为关键字）

2、catch 一个 C++捕捉到的异常。（catch 为关键字）

3、exec 调用系统调用 exec 时。（exec 为关键字，目前此功能只在 HP-UX 下有用）

4、fork 调用系统调用 fork 时。（fork 为关键字，目前此功能只在 HP-UX 下有用）

5、vfork 调用系统调用 vfork 时。（vfork 为关键字，目前此功能只在 HP-UX 下有用）

6、load 或 load 载入共享库（动态链接库）时。（load 为关键字，目前此功能只在 HP-UX 下有用）

7、unload 或 unload 卸载共享库（动态链接库）时。（unload 为关键字，目前此功能只在 HP-UX 下有用）

tcatch 只设置一次捕捉点，当程序停住以后，应点被自动删除。

四、维护停止点

上面说了如何设置程序的停止点，GDB 中的停止点也就是上述的三类。在 GDB 中，如果你觉得已定义好的停止点没有用了，你可以使用 delete、clear、disable、enable 这几个命令来进行维护。

clear 清除所有的已定义的停止点。

clear 清除所有设置在函数上的停止点。

clear 清除所有设置在指定行上的停止点。

delete [breakpoints] [range...]

删除指定的断点，breakpoints 为断点号。如果不指定断点号，则表示删除所有的断点。range 表示断点号的范围（如：3-7）。其简写命令为 d。

比删除更好的一种方法是 disable 停止点，disable 了的停止点，GDB 不会删除，当你还需要时，enable 即可，就好像回收站一样。

disable [breakpoints] [range...]

disable 所指定的停止点，breakpoints 为停止点号。如果什么都不指定，表示 disable 所有的停止点。简写命令是 dis.

enable [breakpoints] [range...]

enable 所指定的停止点，breakpoints 为停止点号。

enable [breakpoints] once range...

enable 所指定的停止点一次，当程序停止后，该停止点马上被 GDB 自动 disable。

enable [breakpoints] delete range...

enable 所指定的停止点一次，当程序停止后，该停止点马上被 GDB 自动删除。

五、停止条件维护

前面在说到设置断点时，我们提到过可以设置一个条件，当条件成立时，程序自动停止，这是一个非常强大的功能，这里，我想专门说说这个条件的相关维护命令。一般来说，为断点设置一个条件，我们使用 if 关键词，后面跟其断点条件。并且，条件设置好后，我们可以用 condition 命令来修改断点的条件。（只有 break 和 watch 命令支持 if，catch 目前暂不支持 if）

condition 修改断点号为 bnum 的停止条件为 expression。

condition 清除断点号为 bnum 的停止条件。

还有一个比较特殊的维护命令 ignore，你可以指定程序运行时，忽略停止条件几次。

ignore 表示忽略断点号为 bnum 的停止条件 count 次。

六、为停止点设定运行命令

我们可以使用 GDB 提供的 command 命令来设置停止点的运行命令。也就是说，当运行的程序在被停止住时，我们可以让其自动运行一些别的命令，这很有利行自动化调试。对基于 GDB 的自动化调试是一个强大的支持。

```
commands [bnum]
... command-list ...
end
```

为断点号 bnum 指写一个命令列表。当程序被该断点停住时，gdb 会依次运行命令列表中的命令。

例如：

```
break foo if x>0
commands
printf "x is %d/n",x
continue
end
```

断点设置在函数 foo 中，断点条件是 x>0，如果程序被断住后，也就是，一旦 x 的值在 foo 函数中大于 0，GDB 会自动打印出 x 的值，并继续运行程序。

如果你要清除断点上的命令序列，那么只要简单的执行一下 commands 命令，并直接在打个 end 就行了。

七、断点菜单

在 C++中，可能会重复出现同一个名字的函数若干次（函数重载），在这种情况下，break 不能告诉 GDB 要停在哪个函数的入口。当然，你可以使用 break 也就是把函数的参数类型告诉 GDB，以指定一个函数。否则的话，GDB 会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了。如：

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
```

```
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

可见，GDB 列出了所有 after 的重载函数，你可以选一下列表编号就行了。0 表示放弃设置断点，1 表示所有函数都设置断点。

八、恢复程序运行和单步调试

当程序被停住了，你可以用 continue 命令恢复程序的运行直到程序结束，或下一个断点到来。

也可以使用 step 或 next 命令单步跟踪程序。

continue [ignore-count]

c [ignore-count]

fg [ignore-count]

恢复程序运行，直到程序结束，或是下一个断点到来。ignore-count 表示忽略其后的断点次数。continue，c，fg 三个命令都是一样的意思。

step 单步跟踪，如果有函数调用，他会进入该函数。进入函数的前提是，此函数被编译有 debug 信息。很像 VC 等工具中的 step in。后面可以加 count 也可以不加，不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

next 同样单步跟踪，如果有函数调用，他不会进入该函数。很像 VC 等工具中的 step over。后面可以加 count 也可以不加，不加表示一条条地执行，加表示执行后面的 count 条指令，然后再停住。

set step-mode

set step-mode on

打开 step-mode 模式，于是，在进行单步跟踪时，程序不会因为没有 debug 信息而不停住。

这个参数有很利于查看机器码。

set step-mod off

关闭 step-mode 模式。

finish 运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。

until 或 u 当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

stepi 或 si

nexti 或 ni

单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令。与之一样有相同功能的命令是“display/i $pc”，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）

九、信号（Signals）

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是 UNIX，比较重要应用程序一般都会处理信号。UNIX 定义了许多信号，比如 SIGINT 表示中断字符信号，也就是 Ctrl+C 的信号，SIGBUS 表示硬件故障的信号；SIGCHLD 表示子进程状态改变信号；SIGKILL 表示终止程序运行的信号，等等。信号量编程是 UNIX 下非常重要的一种技术。

GDB 有能力在你调试程序的时候处理任何一种信号，你可以告诉 GDB 需要处理哪一种信号。你可以要求 GDB 收到你所指定的信号时，马上停住正在运行的程序，以供你进行调试。

你可以用 GDB 的 handle 命令来完成这一功能。

handle 在 GDB 中定义一个信号处理。信号可以以 SIG 开头或不以 SIG 开头，可以用定义一个要处理信号的范围（如：SIGIO-SIGKILL，表示处理从 SIGIO 信号到 SIGKILL 的信号，其中包括 SIGIO，SIGIOT，SIGKILL 三个信号），也可以使用关键字 all 来标明要处理所有的信号。

一旦被调试的程序接收到信号，运行程序马上会被 GDB 停住，以供调试。其可以是以下几种关键字的一个或多个。

nostop 当被调试的程序收到信号时，GDB 不会停住程序的运行，但会打出消息告诉你收到这种信号。

stop 当被调试的程序收到信号时，GDB 会停住你的程序。

print 当被调试的程序收到信号时，GDB 会显示出一条信息。

noprint 当被调试的程序收到信号时，GDB 不会告诉你收到信号的信息。

pass noignore 当被调试的程序收到信号时，GDB 不处理信号。这表示，GDB 会把这个信号交给被调试程序会处理。

nopass ignore 当被调试的程序收到信号时，GDB 不会让被调试程序来处理这个信号。

info signals、info handle 查看有哪些信号在被 GDB 检测中。

十、线程（Thread Stops）

如果你程序是多线程的话，你可以定义你的断点是否在所有的线程上，或是在某个特定的线程。GDB 很容易帮你完成这一工作。

break thread

break thread if ...

linespec 指定了断点设置在的源程序的行号。threadno 指定了线程的 ID，注意，这个 ID 是 GDB 分配的，你可以通过“info threads”命令来查看正在运行程序中的线程信息。如果你不指定 thread 则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件。如：

(gdb) break frik.c:13 thread 28 if bartab > lim

当你的程序被 GDB 停住时，所有的运行线程都会被停住。这方便你你查看运行程序的总体情况。而在你恢复程序运行时，所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

(g) 查看栈信息

当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。当你的程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。你可以用 GDB 命令来查看当前的栈中的信息。

下面是一些查看函数调用栈信息的 GDB 命令：

backtrace

bt

打印当前的函数调用栈的所有信息。如：

```
(gdb) bt
#0 func (n=250) at tst.c:6
#1 0x08048524 in main (argc=1, argv=0xbffff674) at tst.c:30
#2 0x400409ed in __libc_start_main () from /lib/libc.so.6
```

从上可以看出函数的调用栈信息：__libc_start_main --> main() --> func()

backtrace

bt

n 是一个正整数，表示只打印栈顶上 n 层的栈信息。

backtrace <-n>

bt <-n>

-n 表一个负整数，表示只打印栈底下 n 层的栈信息。

如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈。

frame

f

n 是一个从 0 开始的整数，是栈中的层编号。比如：frame 0，表示栈顶，frame 1，表示栈的第二层。

up

表示向栈的上面移动 n 层，可以不打 n，表示向上移动一层。

down

表示向栈的下面移动 n 层，可以不打 n，表示向下移动一层。

上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：

select-frame 对应于 frame 命令。

up-silently 对应于 up 命令。

down-silently 对应于 down 命令。

查看当前栈层的信息，你可以用以下 GDB 命令：

frame 或 f

会打印出这些信息：栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。

info frame

info f

这个命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内内地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等。如：

```
(gdb) info f
Stack level 0, frame at 0xbffff5d4:
eip = 0x804845d in func (tst.c:6); saved eip 0x8048524
called by frame at 0xbffff60c
source language c.
Arglist at 0xbffff5d4, args: n=250
Locals at 0xbffff5d4, Previous frame's sp is 0x0
Saved registers:
ebp at 0xbffff5d4, eip at 0xbffff5d8
```

info args

打印出当前函数的参数名及其值。

info locals

打印出当前函数中所有局部变量及其值。

info catch

打印出当前的函数中的异常处理信息。

7.8.4 查看源程序

一、显示源代码

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g 的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB 会报告程序停在了那个文件的第几行上。你可以用 list 命令来打印程序的源代码。还是来看一看查看源代码的 GDB 命令吧。

list

显示程序第 linenum 行的周围的源程序。

list

显示函数名为 function 的函数的源程序。

list

显示当前行后面的源程序。

list -

显示当前行前面的源程序。

一般是打印当前行的上 5 行和下 5 行，如果显示函数是是上 2 行下 8 行，默认是 10 行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

set listsize

设置一次显示源代码的行数。

show listsize

查看当前 listsize 的设置。

list 命令还有下面的用法：

list ,

显示从 first 行到 last 行之间的源代码。

list ,

显示从当前行到 last 行之间的源代码。

list +

往后显示源代码。

一般来说在 list 后面可以跟以下这们的参数：

行号。

<+offset> 当前行号的正偏移量。

<-offset> 当前行号的负偏移量。

哪个文件的哪一行。

函数名。

哪个文件中的哪个函数。

<*address> 程序运行时的语句在内存中的地址。

二、搜索源代码

不仅如此，GDB 还提供了源代码搜索的命令：

forward-search

search

向前面搜索。

reverse-search

全部搜索。

其中，就是正则表达式，也主一个字符串的匹配模式，关于正则表达式，我就不在这里讲了，还请各位查看相关资料。

三、指定源文件的路径

某些时候，用-g 编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB 提供了可以让你指定源文件的路径的命令，以便 GDB 进行搜索。

directory

dir

加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX 下你可以使用“:”，Windows 下你可以使用“;”。

directory

清除所有的自定义的源文件搜索路径信息。

show directories

显示定义了的源文件搜索路径。

四、源代码的内存

你可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟“行号”，“函数名”，“文件名:行号”，“文件名:函数名”，这个命令会打印出所指定的源代码在运行时的内存地址，如：

(gdb) info line tst.c:func

Line 5 of "tst.c" starts at address 0x8048456 and ends at 0x804845d .

还有一个命令（disassemble）你可以查看源程序的当前执行时的机器码，这个命令会把目前内存中的指令 dump 出来。如下面的示例表示查看函数 func 的汇编代码。

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 : push %ebp
0x8048451 : mov %esp,%ebp
0x8048453 : sub $0x18,%esp
0x8048456 : movl $0x0,0xfffffffc(%ebp)
0x804845d : movl $0x1,0xfffffff8(%ebp)
0x8048464 : mov 0xfffffff8(%ebp),%eax
0x8048467 : cmp 0x8(%ebp),%eax
0x804846a : jle 0x8048470
0x804846c : jmp 0x8048480
0x804846e : mov %esi,%esi
0x8048470 : mov 0xfffffff8(%ebp),%eax
0x8048473 : add %eax,0xfffffffc(%ebp)
0x8048476 : incl 0xfffffff8(%ebp)
0x8048479 : jmp 0x8048464
0x804847b : nop
0x804847c : lea 0x0(%esi,1),%esi
0x8048480 : mov 0xfffffffc(%ebp),%edx
0x8048483 : mov %edx,%eax
0x8048485 : jmp 0x8048487
0x8048487 : mov %ebp,%esp
0x8048489 : pop %ebp
0x804848a : ret
End of assembler dump.
```

7.8.5 查看运行时数据

在你调试程序时，当程序被停住时，你可以使用 print 命令（简写命令为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式是：

print

print /

是表达式，是你所调试的程序的语言的表达式（GDB 可以调试多种编程语言），是输出的格式，比如，如果要把表达式按 16 进制的格式输出，那么就是/x。

一、表达式

print 和许多 GDB 的命令一样，可以接受一个表达式，GDB 会根据当前的程序运行的数据来计算这个表达式，既然是表达式，那么就可以是当前程序运行中的 const 常量、变量、函数等内容。可惜的是 GDB 不能使用你在程序中所定义的宏。

表达式的语法应该是当前所调试的语言的语法，由于 C/C++是一种大众型的语言，所以，本文中的例子都是关于 C/C++的。（而关于用 GDB 调试其它语言的章节，我将在后面介绍）在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中。

@ 是一个和数组有关的操作符，在后面会有更详细的说明。

:: 指定一个在文件或是一个函数中的变量。

{} 表示一个指向内存地址的类型为 type 的一个对象。

二、程序变量

在 GDB 中，你可以随时查看以下三种变量的值：

1、全局变量（所有文件可见的）

2、静态全局变量（当前文件可见的）

3、局部变量（当前 Scope 可见的）

如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量，也就是说，如果一个全局变量和一个函数中的局部变

量同名时，如果当前停止点在函数中，用 print 显示出的变量的值会是函数中的局部变量的值。如果此时你想查看全局变量

的值时，你可以使用“::”操作符：

file::variable

function::variable

可以通过这种形式指定你所想查看的变量，是哪个文件中的或是哪个函数中的。例如，查看文件 f2.c 中的全局变量 x 的值：

gdb) p 'f2.c'::x

当然，“::”操作符会和 C++中的发生冲突，GDB 能自动识别“::” 是否 C++的操作符，所以你不必担心在调试 C++程序时会出现异常。

另外，需要注意的是，如果你的程序编译时开启了优化选项，那么在用 GDB 调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。这个是很正常的，因为优化程序会删改你的程序，整理你程序的语句顺序，剔除一些无意义的变量等，所以在 GDB 调试这种程序时，运行时的指令和你所编写指令就有不一样，也就会出现你所想象不到的结果。对付这种情况时，需要在编译程序时关闭编译优化。一般来说，几乎所有的编译器都支持编译优化的开关，例如，GNU 的 C/C++编译器 GCC，你可以使用“-gstabs”选项来解决这个问题。关于编译器的参数，还请查看编译器的使用说明文档。

三、数组

有时候，你需要查看一段连续的内存空间的值。比如数组的一段，或是动态分配的数据的大小。你可以使用 GDB 的“@”操作符，“@”的左边是第一个内存的地址的值，“@”的右边则你你想查看内存的长度。例如，你的程序中有这样的语句：

int *array = (int *) malloc (len * sizeof (int));

于是，在 GDB 调试过程中，你可以以如下命令显示出这个动态数组的取值：

p *array@len

@的左边是数组的首地址的值，也就是变量 array 所指向的内容，右边则是数据的长度，其保存在变量 len 中，其输出结果，大约是下面这个样子的：

(gdb) p *array@len

$1 = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}

如果是静态数组的话，可以直接用 print 数组名，就可以显示数组中所有数据的内容了。

四、输出格式

一般来说，GDB 会根据变量的类型输出变量的值。但你也可以自定义 GDB 的输出的格式。

例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用 GDB 的数据显示格式：

x 按十六进制格式显示变量。

d 按十进制格式显示变量。

u 按十六进制格式显示无符号整型。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

a 按十六进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

(gdb) p i

$21 = 101

(gdb) p/a i

$22 = 0x65

(gdb) p/c i

$23 = 101 'e'

(gdb) p/f i

$24 = 1.41531145e-43

(gdb) p/x i

$25 = 0x65

(gdb) p/t i

$26 = 1100101

五、查看内存

你可以使用 examine 命令（简写是 x）来查看内存地址中的值。x 命令的语法如下所示：

x/

n、f、u 是可选的参数。

n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。

f 表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是 s，如果地十是指令地址，那么格式可以是 i。

u 表示从当前地址往后请求的字节数，如果不指定的话，GDB 默认是 4 个 bytes。u 参数可以用下面的字符来代替，b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当我们指定了字节长度后，GDB 会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。

表示一个内存地址。

n/f/u 三个参数可以一起使用。例如：

命令：x/3uh 0x54320 表示，从内存地址 0x54320 读取内容，h 表示以双字节为一个单位，3 表示三个单位，u 表示按十六进制显示。

六、自动显示

你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的 GDB 命令是 display。

display

display/

display/

expr 是一个表达式，fmt 表示显示的格式，addr 表示内存地址，当你用 display 设定好了一个或多个表达式后，只要你的程序被停下来，GDB 会自动显示你所设置的这些表达式的值。

格式 i 和 s 同样被 display 支持，一个非常有用的命令是：

display/i $pc

$pc 是 GDB 的环境变量，表示着指令的地址，/i 则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

下面是一些和 display 相关的 GDB 命令：

undisplay

delete display

删除自动显示，dnums 意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

disable display

enable display

disable 和 enalbe 不删除自动显示的设置，而只是让其失效和恢复。

info display

查看 display 设置的自动显示的信息。GDB 会打出一张表格，向你报告当然调试中设置了多少个自动显示设置，其中包括，设置的编号，表达式，是否 enable。

七、设置显示选项

GDB 中关于显示的选项比较多，这里我只例举大多数常用的选项。

set print address

set print address on

打开地址输出，当程序显示函数信息时，GDB 会显出函数的参数地址。系统默认为打开的，如：

(gdb) f

#0 set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")

at input.c:530

530 if (lquote != def_lquote)

set print address off

关闭函数的参数地址显示，如：

(gdb) set print addr off

(gdb) f

#0 set_quotes (lq="<<", rq=">>") at input.c:530

530 if (lquote != def_lquote)

show print address

查看当前地址显示选项是否打开。

set print array

set print array on

打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。这个选项默认是关闭的。与之相关的两个命令如下，我就不再多说了。

set print array off

show print array

set print elements

这个选项主要是设置数组的，如果你的数组太大了，那么就可以指定一个来指定数据显示的最大长度，当到达这个长度时，GDB 就不再往下显示了。如果设置为 0，则表示不限制。

show print elements

查看 print elements 的选项信息。

set print null-stop

如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为 off。

set print pretty on

如果打开 printf pretty 这个选项，那么当 GDB 显示结构体时会比较漂亮。如：

$1 = {

next = 0x0,

flags = {

sweet = 1,

sour = 1

},

meat = 0x54 "Pork"

}

set print pretty off

关闭 printf pretty 这个选项，GDB 显示结构体时会如下显示：

$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}

show print pretty

查看 GDB 是如何显示结构体的。

set print sevenbit-strings

设置字符显示，是否按“/nnn”的格式显示，如果打开，则字符串或字符数据按/nnn 显示，如“/065”。

show print sevenbit-strings

查看字符显示开关是否打开。

set print union

设置显示结构体时，是否显式其内的联合体数据。例如有以下数据结构：

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
Bug_forms;
struct thing {
  Species it;
  union {
Tree_forms tree;
Bug_forms bug;
  } form;
};
struct thing foo = {Tree, {Acorn}};
```

当打开这个开关时，执行 p foo 命令后，会如下显示：

$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}

当关闭这个开关时，执行 p foo 命令后，会如下显示：

$1 = {it = Tree, form = {...}}

show print union

查看联合体数据的显示方式

set print object

在 C++中，如果一个对象指针指向其派生类，如果打开这个选项，GDB 会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，GDB 就不管虚函数表了。这个选项默认是 off。

show print object

查看对象选项的设置。

set print static-members

这个选项表示，当显示一个 C++对象中的内容是，是否显示其中的静态数据成员。默认是 on。

show print static-members

查看静态数据成员选项设置。

set print vtbl

当此选项打开时，GDB 将用比较规整的格式来显示虚函数表时。其默认是关闭的。

show print vtbl

查看虚函数显示格式的选项。

八、历史记录

当你用 GDB 的 print 查看程序运行时的数据时，你每一个 print 都会被 GDB 记录下来。GDB 会以 1,2, 这样的方式为你每一个命令编上号。于是，你可以使用这个编号访问以前的表达式，如 3.....这样的方式为你每一个 p□r□i□n□t 命令编上号。于是，你可以使用这个编号访问以前的表达式，如 1。这个功能所带来的好处是，如果你先前输入了一个比较长的表达式，如果你还想查看这个表达式的值，你可以使用历史记录来访问，省去了重复输入。

九、GDB 环境变量

你可以在 GDB 的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个 GDB 的变量很简单只需。使用 GDB 的 set 命令。GDB 的环境变量和 UNIX 一样，也是以$起头。如：

set $foo = *object_ptr

使用环境变量时，GDB 会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组。

show convenience

该命令查看当前所设置的所有的环境变量。

这是一个比较强大的功能，环境变量和程序变量的交互使用，将使得程序调试更为灵活便捷。

例如：

set $i = 0

print bar[$i++]->contents

于是，当你就不必，print bar[0]->contents, print bar[1]->contents 地输入命令了。输入这样的命令后，只用敲回车，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

十、查看寄存器

要查看寄存器的值，很简单，可以使用如下命令：

info registers

查看寄存器的情况。（除了浮点寄存器）

info all-registers

查看所有寄存器的情况。（包括浮点寄存器）

info registers

查看所指定的寄存器的情况。

寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址（ip），程序的当前堆栈地址（sp）等等。你同样可以使用 print 命令来访问寄存器的情况，只需要在寄存器名字前加一个符号就可以了。如：符号就可以了。如：peip。

7.8.6 改变程序的执行

一旦使用 GDB 挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在 GDB 中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，比如，你可以在程序的一次运行中走遍程序的所有分支。

一、修改变量值

修改被调试程序运行时的变量值，在 GDB 中很容易实现，使用 GDB 的 print 命令即可完成。如：

(gdb) print x=4

x=4 这个表达式是 C/C++的语法，意为把变量 x 的值修改为 4，如果你当前调试的语言是 Pascal，那么你可以使用 Pascal 的语法：x:=4。

在某些时候，很有可能你的变量和 GDB 中的参数冲突，如：

(gdb) whatis width

type = double

(gdb) p width

$4 = 13

(gdb) set width=47

Invalid syntax in expression.

因为，set width 是 GDB 的命令，所以，出现了“Invalid syntax in expression”的设置错误，此时，你可以使用 set var 命令来告诉 GDB，width 不是你 GDB 的参数，而是程序的变量名，如：

(gdb) set var width=47

另外，还可能有些情况，GDB 并不报告这种错误，所以保险起见，在你改变程序变量取值时，最好都使用 set var 格式的 GDB 命令。

二、跳转执行

一般来说，被调试程序会按照程序代码的运行顺序依次执行。GDB 提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由 GDB 的 jump 命令来完：

jump

指定下一条语句的运行点。可以是文件的行号，可以是 file:line 格式，可以是+num 这种偏移量格式。表式着下一条运行语句从哪里开始。

jump

这里的是代码行的内存地址。

注意，jump 命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序 Core Dump。所以最好是同一个函数中进行跳转。

熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump 命令也就是改变了这个寄存器中的值。于是，你可以使用“set $pc”来更改跳转执行的地址。如：

set $pc = 0x485

三、产生信号量

使用 singal 命令，可以产生一个信号量给被调试的程序。如：中断信号 Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号非常有利程序的调试。

语法是：signal ，UNIX 的系统信号量通常从 1 到 15。所以取值也在这个范围。 single 命令和 shell 的 kill 命令不同，系统的 kill 命令发信号给被调试程序时，是由 GDB 截获的，而 single 命令所发出一信号则是直接发给被调试程序的。

四、强制函数返回

如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用 return 命令强制函数忽略还没有执行的语句并返回。

return

return

使用 return 命令取消当前函数的执行，并立即返回，如果指定了，那么该表达式的值会被认作函数的返回值。

五、强制调用函数

call

表达式中可以一是函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值是 void，那么就不显示。 另一个相似的命令也可以完成这一功能——print，print 后面可以跟表达式，所以也可以用他来调用函数，print 和 call 的不同是，如果函数返回 void，call 则不显示，print 则显示函数返回值，并把该值存入历史数据中。

（7）在不同语言中使用 GDB

GDB 支持下列语言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般说来，GDB 会根据你所调试的程序来确定当然的调试语言，比如：发现文件名后缀为“.c”的，GDB 会认为是 C 程序。文件名后缀为“.C, .cc, .cp, .cpp, .cxx, .c++”的，GDB 会认为是 C++程序。而后缀是“.f, .F”的，GDB 会认为是 Fortran 程序，还有，后缀为如果是“.s, .S”的会认为是汇编语言。

也就是说，GDB 会根据你所调试的程序的语言，来设置自己的语言环境，并让 GDB 的命令跟着语言环境的改变而改变。比如一些 GDB 命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。例如 C/C++中对指针的语法是*p，而在 Modula-2 中则是 p^。并且，如果你当前的程序是由几种不同语言一同编译成的，那到在调试过程中，GDB 也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，真是体贴开发人员的一种设计。

下面是几个相关于 GDB 语言环境的命令：

show language

查看当前的语言环境。如果 GDB 不能识为你所调试的编程语言，那么，C 语言被认为是默认的环境。

info frame

查看当前函数的程序语言。

info source

查看当前文件的程序语言。

如果 GDB 没有检测出当前的程序语言，那么你也可以手动设置当前的程序语言。使用 set

language 命令即可做到。

当 set language 命令后什么也不跟的话，你可以查看 GDB 所支持的语言种类：

(gdb) set language

The currently understood settings are:

local or auto Automatic setting based on source file

c Use the C language

c++ Use the C++ language

asm Use the Asm language

chill Use the Chill language

fortran Use the Fortran language

java Use the Java language

modula-2 Use the Modula-2 language

pascal Use the Pascal language

scheme Use the Scheme language

于是你可以在 set language 后跟上被列出来的程序语言名，来设置当前的语言环境。

7.8.7 小结

GDB 是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX下的软件全是命令行的，这给程序开发提代供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。

UNIX 下的软件比 Windows 下的软件更能有机地结合，各自发挥各自的长处，组合成更为强劲的功能。而 Windows 下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。在这里并不是要和 Windows 做个什么比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。

我根据版本为5.1.1的GDB所写的这篇文章，所以可能有些功能已被修改，或是又有更为强劲的功能。而且，我写得非常仓促，写得比较简略，并且，其中我已经看到有许多错别字了（我用五笔，所以错字让你看不懂），所以，我在这里对我文中的差错表示万分的歉意。

文中所罗列的 GDB 的功能时，我只是罗列了一些带用的 GDB 的命令和使用方法，其实，我这里只讲述的功能大约只占GDB所有功能的60%吧，详细的

文档，还是请查看 GDB 的帮助和使用手册吧，或许，过段时间，如果我有空，我再写一篇 GDB 的高级使用。

我个人非常喜欢 GDB 的自动调试的功能，这个功能真的很强大，试想，我在 UNIX 下写个脚本，让脚本自动编译我的程序，被自动调试，并把结果报告出来，调试成功，自动 checkin 源代码库。一个命令，编译带着调试带着 checkin，多爽啊。只是 GDB 对自动化调试目前支持还不是很成熟，只能实现半自动化，真心期望着 GDB 的自动化调试功能的成熟。

8 在线判题系统

8.1 学习路线

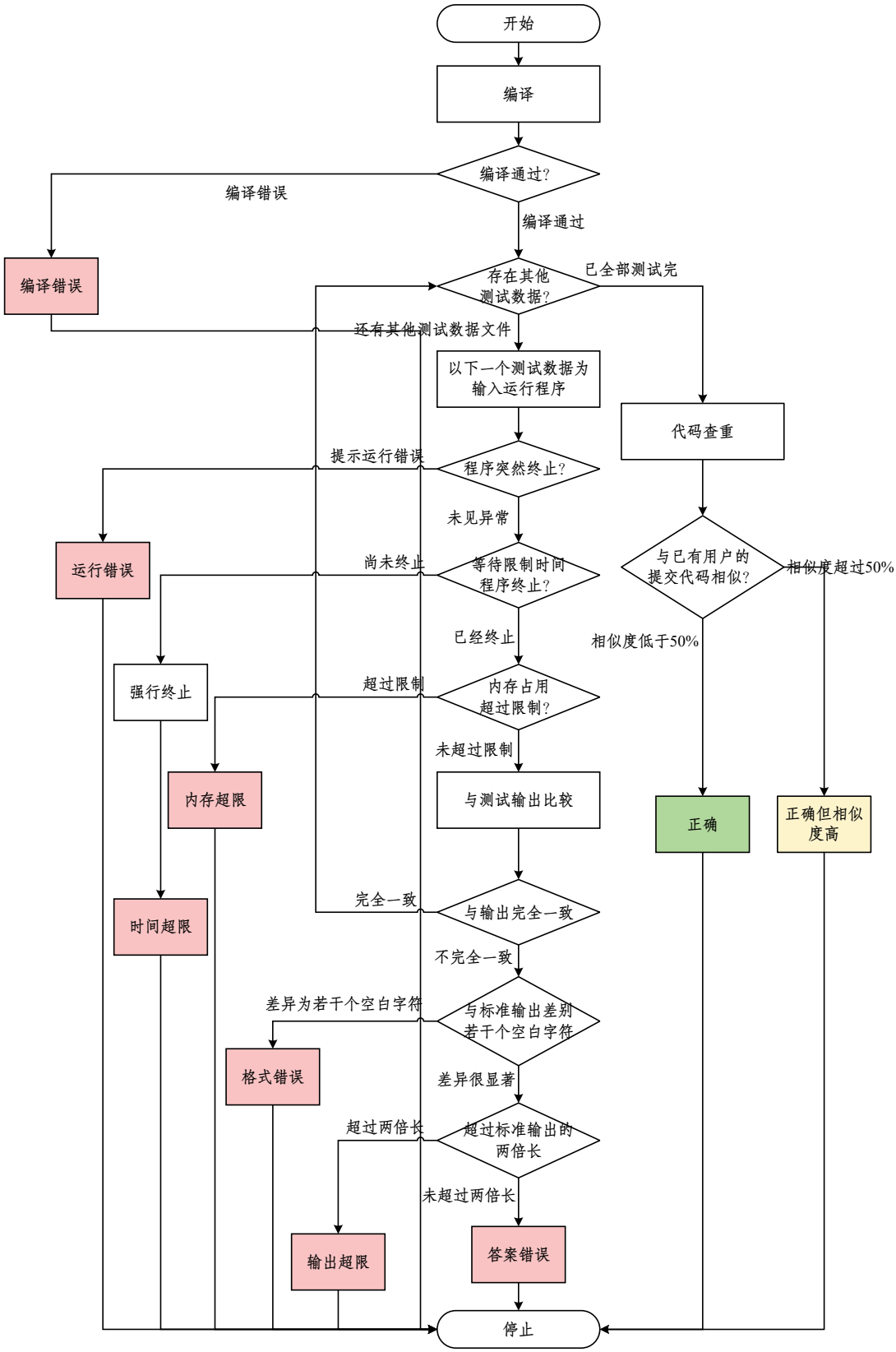
学生可以按先易后难的顺序完成洛谷 OJ 的题单，其地址如下：

https://www.luogu.com.cn/training/list

学有余力的学生可以继续阅读刘汝佳编著的《算法竞赛入门经典（第二版）》。

8.2 判题机工作原理

OJ 判题机的工作原理如下流程图所示。

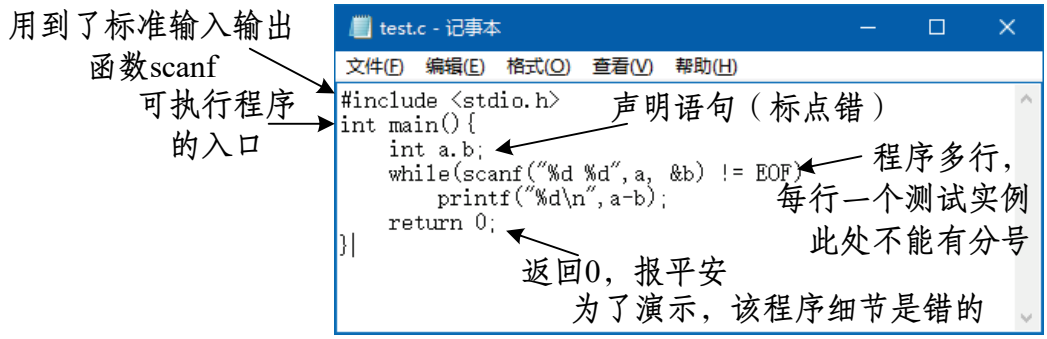


8.3 避免错误的方法

错误一般分为以下几种：

第一，编译错误。编译不通过，则提示编译错误。出现此类错误是很不应该的。为了避免此类错误，学生应先在自己的机器上输入程序。

注意：while(scanf("%d %d", &a, &b)!=EOF)指的是，当输入结束之前，重复运行主体，每行执行 1 次。这样能配合有多个输入、重复次数未知、直至 EOF 结束的类型。如果没有特殊说明，OJ 都是这个类型。



第二，运行错误。程序在运行时提前终止，则提示运行错误。运行错误一般是你没有很好地测试程序的一些边界情况所导致的。明显的边界情况有：例如程序要求输入是长度为 100 的字符串，或者是 1000 个数据，而你测试时却仅限于样例输入，只测试长度为 2 的字符串，这是远远不够的。而不明显的边界情况主要指的是软件测试在单元测试时的测试类划分。

也可能你调用了 gets() 、 puts() 、getch() 、 putchar() 、 system() 等函数，这些函数可能损坏本系统，OJ 君不允许你用。

OJ 君没有义务告诉你哪里可能发生一个运行错误哦，这是考点，考谁的测试更全面。

第三，内存超限。OJ 监测占用内存变化，如果超过“限制内存”，则提示内存超限。一般出现此类问题，是你在程序里开辟过大的内存空间所致。你也可能动态开辟空间却忘了释放它，经过几万次的循环以后内存就超限了。

排除上述情况后，很可能你对空间的把控不够好，导致在大数据的测试下程序明显超限了。例如：求巨型整数的乘法，你图方便开辟了一个 int a[1000][10000]也就是接近 4MB，而你只存储一位十进制数字，存储效率是 0.1（1 个 int 是 32 比特，而十进制位只要 3~4 个比特）。

这是考点，考谁的解题方法更好。有兴趣可以学习一些算法知识。

第四，时间超限。OJ 在等待规定的“限制时间”之后，程序仍未终止，则提示时间超限，并强行终止。

明显的情况是：程序在测试输入情况下死循环了。一个情况是你的程序会死循环，另一个情况是，你使用了错误的输入格式。例如程序要求输入是“3 5”，而你自行测试的时候输入“3,5”，这种情况下，在“3 5”反而会产生错误。例如程序的最后一行是以“EOF”结束的，而你自行测试的时候是以回车结束，并没有 Ctrl+Z 模拟“EOF”，也会产生错误。

排除上述情况后，很可能你的算法不够好，导致在大数据的测试下程序明显超时了。例如：求 n!的最后一位，显然在 5!及以后答案就是 0 了，可是你的程序一个个算，最后虽然也输出 0，但在测试数据 n=200000000 的时候，很可能时间超限了。

这是考点，考谁的解题方法更好。有兴趣可以学习一些算法知识。

如果你的算法不够好，你可以找一个正确的代码，通过测试大量数据（或者少量循环多次），直观感受耗时的差异。然后你可以在一些循环里加上计数器 count++，然后打印一下 count，两个程序都打印，做出对比。只有这样，你才能才能服膺，才能静下心来看别人的代码和自己的代码有什么区别。

第五，格式错误。你要注意看输出到底有没有空格或者回车（可以选中答案），然后将你的输出重定向到文件里（选中你的输出）看看有没有空格或回车，来进行比较。

尽量不要再使用键盘输入的方式测试数据，因为你有可能按了回车键而没注意到这个回车是需要用 printf 输出的，不是手动添加的，因此，重定向更接近实际。问题来了，如何重定向呢？你需要把你的测试数据存到一个文件里。然后通过这个方法来重定向输入和输出。

现在你可以打开 test.txt，选中它，看看内容是不是只有 3、回车，如果不是再说。注意，一个字符也不能少。

这样你的输入文件可以很长一串，如果因此程序的输出有好多好多，超过了肉眼可以看的范围怎么办？那就用 diff 命令。你把标准答案存在一个文件里，比较。

第六，答案错误。终究解决避免“答案错误”的问题需要用到“对拍法”自己百度吧。我简要地说下，就是你编写一个代码简单但可能无法计算大数据或者耗时特别长的（但一定是答案正确）的程序，然后用随机数生成题目规定的输入，用这个一定正确的程序输出得到参考答案。然后你正式做题的时候，讲你的程序与这个输入输出做比较，直到找到你程序的大部分缺陷。

值得指出的是 OJ 君提供了答案错误查询功能，大家不要依赖于答案，不要觉得答案错了我就改吧，改到对了就对了。不要觉得 AC 是第一位的，独立完成独立思考才是第一位的。答案错误开放查询是让你发现你在思考问题的时候会在哪些地方思虑不周，以期改善，不是让你以答案为终点。只会写一道题是没有用的。

8.4 构建一致的开发环境

不同的 OJ 系统开发环境不同，这里以某 OJ 为例。

第一步，搭建环境。以下以最轻量级的 Ubuntu Server 系统为例，搭建编译环境需要如下几个过程：

搜索 Fedora、FreeBSD、RedHat 等系统的安装盘 ISO 文件；

当出现以下界面，即为安装成功，此时你可以输入用户名，回车后输入密码（密码不显示）；

```
Ubuntu 16.04.1 LTS ubuntu tty1
ubuntu login:
```

第二步，安装 GCC 编译器和调试器：

安装前选择中国区的源服务器以加速下载：备份 /etc/apt/sources.list ，用下述内容替换该文件（其中单词 xenial 应当换成原有文件相应的单词），然后应用更新 sudo apt-get update ；

```
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial main
restricted
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial main
restricted
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-updates
main restricted
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
updates main restricted
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial universe
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial
universe
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-updates
universe
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
updates universe
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial
multiverse
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial
multiverse
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-updates
multiverse
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
updates multiverse
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-backports
main restricted universe multiverse
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
backports main restricted universe multiverse
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-security
main restricted
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
security main restricted
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-security
universe
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
security universe
deb http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-security
multiverse
deb-src http://mirrors.xmu.edu.cn/ubuntu/archive/ xenial-
security multiverse
```

通过在控制台（Terminal）执行 sudo apt-get install build-essential 以安装 GCC 等编辑器；

可使用文本编辑器为 vi 。

用户也可以自行选择安装 Telnet 或 OpenSSH 远程桌面或者图形用户界面。

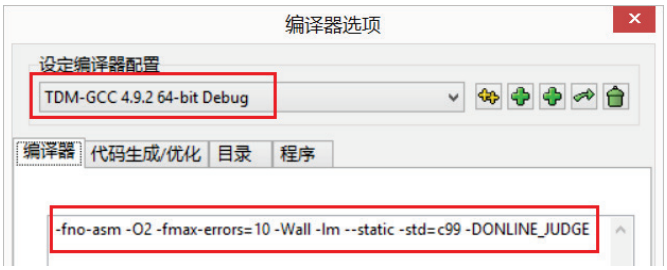
8.5 语言和编译选项

为了防止编译错误，请大家不要在输入框直接编辑程序。在使用 DevC++时，注意以下情况，与比赛系统保持一致性。

以下以 C 语言为例（MyOJ2 只支持 C 语言），如果你常用的语言是 C++语言（积分赛 OJ 应该会支持的），那么也应该查找对应的语言设置。

保存文件时，应选择文件类型为 C 语言（*.c）而不是 C++语言（*.cpp）。

比赛前应打开比赛系统的 FAQ 页面（菜单中的【常见问答】），找到 C 语言的编译命令行。随后，在 DevC++的菜单【工具\编译器选项】里设置。



设置后，应在编译时选用对应的语言和对应的编译选项，以免徒劳。

9 课程项目：2048游戏

9.1 项目简介

本课程项目要求学生在教师指导下，设计并实现一个2048游戏。

9.2 项目需求

项目拟实现：

1、有一个4*4的棋盘。

2、开始时棋盘内随机出现两个数字，出现的数字仅可能为2或4。

3、玩家可以选择上下左右四个方向，棋盘内的棋子按该方向位移。

a) 若有棋子在移动方向的一侧（如：右侧）为空，则将其移动到该位。

b) 若有棋子在移动方向的一侧（如：右侧）为同样的数字，则将其与该侧数字相加。不可将两个以上数字相加。

c) 若该方向无棋子满足a)或b)条件，则不移动棋子。

4、每有效移动一步，棋盘的空位（无数字处）随机出现一个数字（要求同第2套）。

5、若四个方向都满足第3条c)的规则，即无法移动棋子，则判负。

6、若有任何的棋子为2048，则判胜。

9.3 关键技术

1、随机数

产生随机数有多种不同的方法。这些方法被称为随机数发生器。随机数最重要的特性是：它所产生的后面的那个数与前面的那个数毫无关系。在实际应用中往往使用伪随机数就足够了。这些数列是“似乎”随机的数，实际上它们是通过一个固定的、可以重复的计算方法产生的。计算机或计算器产生的随机

数有很长的周期性。它们不真正地随机，因为它们实际上是可以计算出来的，但是它们具有类似于随机数的统计特征。这样的发生器叫做伪随机数发生器。rand()函数即是伪随机数发生器。

随机数主要用于：选择新增数字 在棋盘的位置，其中 r_1, r_2 ， $r \in \{2,4\}.$ ，$r_1, r_2 \in \{0,1,2,3\}.$ 。

使用方法：

（1）先用当前时间设置随机数的种子。例如： srand((int)time(NULL)); 。因为程序运行的时间是随机的，用它来生成下一个数字也是不可预测的。

（2）每次使用时，用 rand() 生成一个大整数，然后再模 ，获得 之间的整数。

2、获取键盘输入

可以获取键盘输入，但不显示在界面上。

相关函数：getch()

需要包含的函数头：（自行百度解决）

如果不会用的话，可以用 getchar()或 scanf()，反正无非做出来难看点，难不难看和能不能用是两个问题。

3、清空屏幕

可以清空屏幕，使得棋盘固定在左上角。

相关函数：clrscr()

需要包含的函数头：（自行上网搜索解决）

如果不会用的话，可以不清空，反正无非做出来难看点，难不难看和能不能用是两个问题。

（1）基本知识

需要开发这样一个游戏，需要以下几部分的知识，如有用到建议自行预习，或避开相关技术。我们还有半个学期时间可以改进它。

1、数组：用于表示棋盘；

2、循环语句：用于将作用相近的代码进行简化；

3、选择语句：用于判断不同的情况种类。

4、数据类型和格式化输入输出。

没有了，真的没用到其它技术了。

（2）高级专题

对于入学前已有编程经验的同学，可以考虑：

1、加入计分功能（先观察别人的 2048 是如何计分的）；

2、优化和封装相关代码，使之更加的“高内聚、低耦合”，提供给同学使用；

3、为这个游戏做测试程序；

4、调试这个程序，减少用户误操作；

5、做个图形界面（使用 Qt）；

6、允许用户设置最高值非 2048 而是其它数值；

7、调研现有 2048 程序的功能，加入你想得到的其它功能。

8、指导其他同学开发，解答他们遇到的困难。

至此，你们已经编写了第一个有人用的小软件。别忘了打上 Copyright @ 20xx, XXX all rights reserved.哦。

9.4 示例程序

本项目的参考代码源自于：https://github.com/mevdschee/2048.c

9.5 评分标准

项目	说明	不合格	合格	优秀
1. 出勤	缺勤应有正规请假手续，另排时间	无故缺勤		
2. 语言	学生打开源代码等所有文件	核心部分不用 C 语言实现		
3. 编译	学生编译程序	无法编译		
4. 运行	将 2048 改成较小的值，如 16，运行程序	无法运行或闪退		
5. 功能点	51. 有 4×4 棋盘	有任何 1 项未正确完成	全部完成	
	52. 初始化棋盘，随机下 2 个棋子			
	53. 上下左右移动棋子，合并相同棋子			
	54. 移动棋子后出现新棋子			
	55. 赢输判定			
	56. 界面可读性好（棋盘对齐）			
	57. 用户主动退出程序（非 Ctrl+C）			
	58. 积分统计			
6. 创新性	61. 游戏中存盘和下次游戏继续	全部未完成	完成 1 项	完成 2 项以上
	62. 可悔棋 5 次以上			
	63. 下棋提示			
	64. 多线程			
	65. 网络连接双机对战			
	66. 图形用户界面			
7. 稳定性	崩溃次数：演示时程序失去和用户之间的交互，被退出或输出不止	≥2	≤1	
	崩溃次数：教师试玩时在不按说明输入（如随机按键盘）	≥3	≤2	
8. 注释	注释次数	≤3	≥4	
	开头注释包含姓名、邮件地址、日期	否	是	
9. 编程风格	变量名和函数名有意义，不用拼音、不滥用缩写	否	是	
	代码缩进是否正确	混乱		
	正确包含必须的头文件	否	是	
	全程滥用单字母的（除循环变量外）	滥用		
	滥用全局变量	滥用		

项目	说明	不合格	合格	优秀
	滥用 **goto** 语句（在退出多层循环或统一错误处理之外的情况使用 **goto**）	滥用		
	使用常量不给予物理意义常量名的	滥用		
10. 高内聚低耦合	存在封装的可能而不封装的：存在多处大段复制代码或只有个别修改的	≥**2** 处	**1** 处	无

10 代码阅读

10.1 简介

阅读是一种运用语言文字来获取信息、认识世界、发展思维，并获得审美体验与知识的活动。作为程序设计语言的学习，对源代码的阅读十分重要。

第一，有用的编程经验往往体现于编程人员的代码中。这些经验无法通过有经验的编程人员完整地总结归纳并写入课本。即便写入课本，那对于初学者而言，也将是枯燥而空洞的。

第二，在实际工作中编程人员需要阅读其他人的代码。个人的能力是有限的，无法在所有场合下抛开前人工作基础而重新开始工作。当你从前辈手中移交新的项目，或者你来到新的项目，或者你从开源代码库中找到值得借鉴的源代码，都要阅读大量由他人编写的代码。

因此，本课程建议学生养成阅读代码的习惯。即：建立正确的代码阅读观念与态度，学习代码阅读的技巧，识别自己在代码阅读上的问题，学会代码阅读，提高工作效率，提升软件调试的效率。

10.2 阅读材料

课程提供最值得学习阅读的 10 个 C 语言开源项目代码，供学生学习。包括：网站压力测试工具 Webbench，轻型 HTTP 服务器 Tinyhttpd，轻量级 JSON 编解码器 cJSON，单元测试轻量级框架 CMockery，开源的事件驱动库 Libev，高性能分布式内存对象缓存系统 Memcached，编程语言 Lua 的编译器，嵌入式关系数据库 SQLite，操作系统 UNIX v6，操作系统 NETBSD。

10.3 阅读示例

一.http 请求

http 请求由三部分组成，分别是：起始行、消息报头、请求正文

```
Request Line<CRLF>
Header-Name: header-value<CRLF>
Header-Name: header-value<CRLF>
//一个或多个，均以<CRLF>结尾
<CRLF>
body//请求正文
```

1、起始行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本，格式如下：

```
Method Request-URI HTTP-Version CRLF
```

其中 Method 表示请求方法；Request-URI 是一个统一资源标识符；HTTP-Version 表示请求的 HTTP 协议版本；CRLF 表示回车和换行（除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符）。

2、请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET 请求获取 Request-URI 所标识的资源

POST 在 Request-URI 所标识的资源后附加新的数据

HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头

PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识

DELETE 请求服务器删除 Request-URI 所标识的资源

TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT 保留将来使用

OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

应用举例：

GET 方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用 GET 方法向服务器获取资源，eg:

```
GET /form.html HTTP/1.1 (CRLF)
```

POST 方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。

eg:

```
POST /reg.jsp HTTP/ (CRLF)
Accept:image/gif,image/x-xbit,... (CRLF)
...
HOST:www.guet.edu.cn (CRLF)
Content-Length:22 (CRLF)
Connection:Keep-Alive (CRLF)
Cache-Control:no-cache (CRLF)
(CRLF)          //该 CRLF 表示消息报头已经结束，在此之前为消息报头
user=jeffrey&pwd=1234  //此行以下为提交的数据
```

二.tinyhttpd 源代码分析

tinyhttpd 总共包含以下函数：

void accept_request(int);//处理从套接字上监听到的一个 HTTP 请求

void bad_request(int);//返回给客户端这是个错误请求，400 响应码

void cat(int, FILE *);//读取服务器上某个文件写到 socket 套接字

void cannot_execute(int);//处理发生在执行 cgi 程序时出现的错误

void error_die(const char *);//把错误信息写到 perror

void execute_cgi(int, const char *, const char *, const char *);//运行 cgi 脚本，这个非常重要，涉及动态解析

int get_line(int, char *, int);//读取一行 HTTP 报文

void headers(int, const char *);//返回 HTTP 响应头

void not_found(int);//返回找不到请求文件

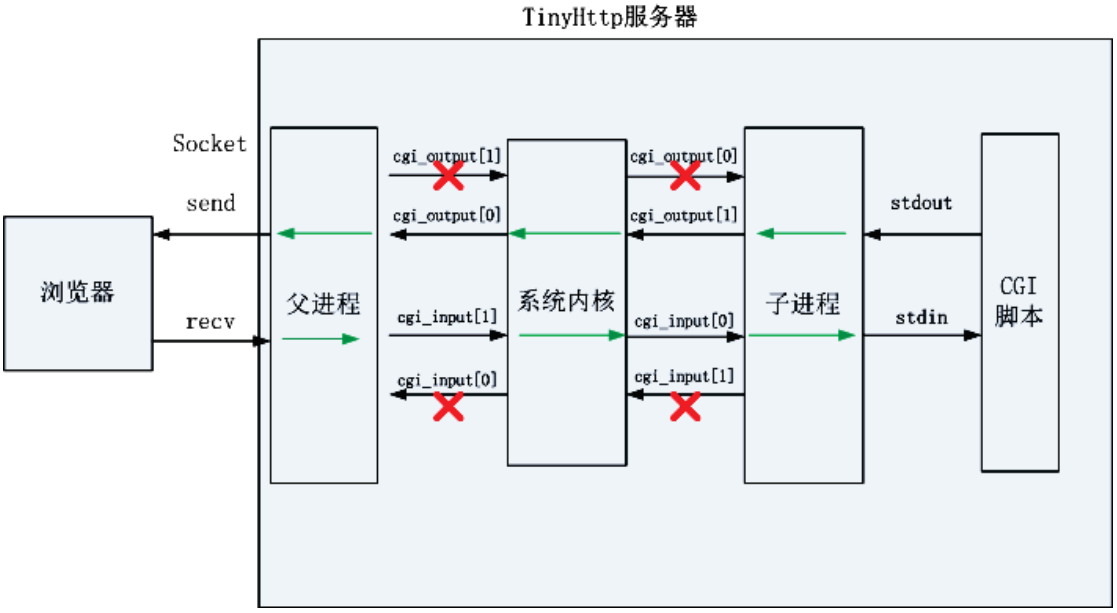
void serve_file(int, const char *);//调用 cat 把服务器文件内容返回给浏览器。

int startup(u_short *);//开启 http 服务，包括绑定端口，监听，开启线程处理链接

void unimplemented(int);//返回给浏览器表明收到的 HTTP 请求所用的 method 不被支持。

建议源代码阅读顺序： main -> startup -> accept_request -> execute_cgi

按照以上顺序，看一下浏览器和 tinyhttpd 交互的整个流程：

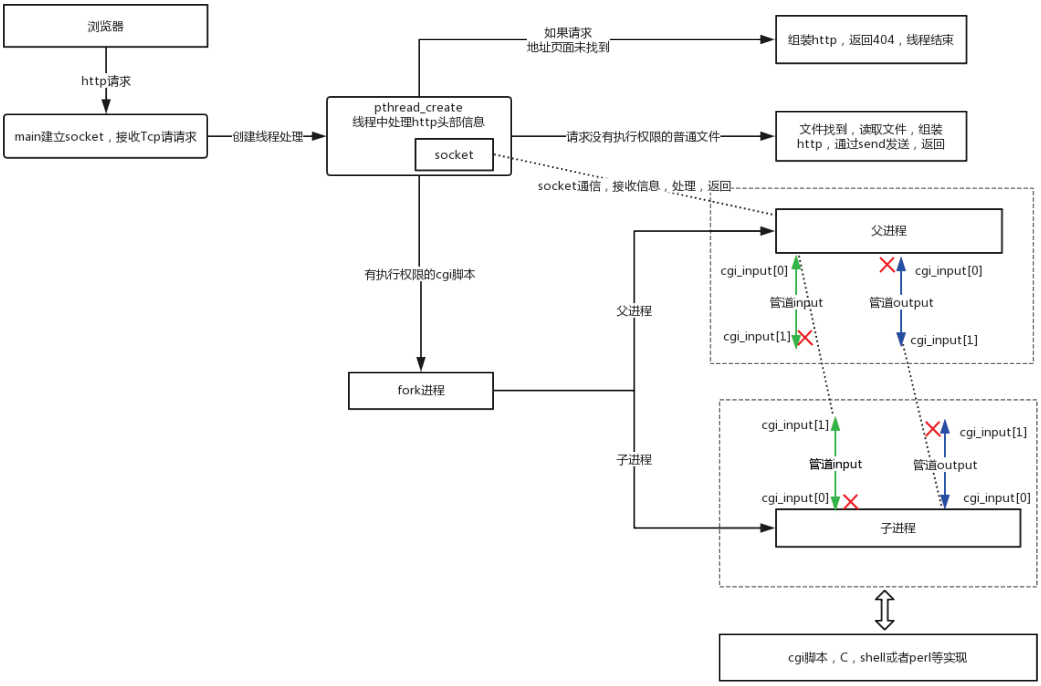


三、注释版源代码

注释版源代码已经放到 github 上了。地址为：

https://github.com/qiyeboy/SourceAnalysis

总体系统架构图：


浏览器
http请求
main建立socket，接收Tcp请请求
创建线程处理
pthread_create
线程中处理http头部信息
socket
如果请求
地址页面未找到
组装http，返回404，线程结束
请求没有执行权限的普通文件
文件找到，读取文件，组装
http，通过send发送，返回
socket通信，接收信息，处理，返回
有执行权限的cgi脚本
fork进程
父进程
子进程
父进程
cgi_input[0]
cgi_input[0]
管道input
管道output
cgi_input[1]
cgi_input[1]
cgi_input[1]
cgi_input[1]
管道input
管道output
cgi_input[0]
cgi_input[0]
子进程
cgi脚本，C，shell或者perl等实现