Linux环境及开发工具应用实践

-Linux开发工具1

zhaofang@email.buptsse.cn

软件学院　　赵方

目录

常用程序编译工具

- 常用的程序编译工具有：
- gcc、g++、egcs、pgcc、calls、cproto、indent、gprof、f2c、p2c

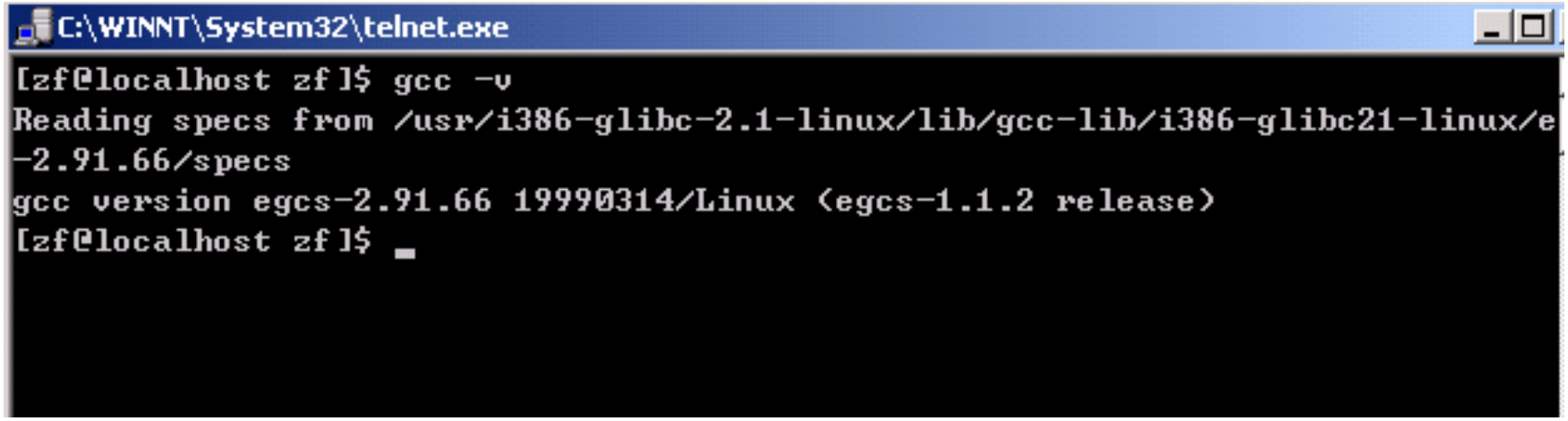
gcc

- gcc（GNU C编译器）
- 是一个在UNIX或linux系统上运行的、功能确定的编译器，主要用于对C/C++/Object C语言的编译。

- Gcc 是Linux开发的基础，名称是：GNU C Compiler 或者GNU Compiler Collection

- 键入：gcc -v可以显示目前使用的gcc的版本

gcc


C:\WINNT\System32\telnet.exe
[zf@localhost zf]$ gcc -v
Reading specs from /usr/i386-glibc-2.1-linux/lib/gcc-lib/i386-glibc21-linux/e
-2.91.66/specs
gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
[zf@localhost zf]$

gcc

- 说明：
  - i386表明现在使用的gcc是为386的微处理器而设计的。
  - glibc21指定发行的系统版本。
  - Linux指所支持的可执行文件格式。
- gcc支持多个平台，可以在安装时指定gcc编译后程序运行的平台，gcc将针对指定平台生成特定的程序。
- Make命令后面再加上configue—target= i386-glibc-2.1-linux—host=XXX on platform XXX，就可以安装面向不同目标平台的gcc。

gcc

- gcc的调用格式：
- $gcc [options] [filenames]
- 其中，filenames是所要编译的程序源文件。
- 当调用gcc时，gcc会完成预处理、编译、汇编和链接。
- 前三步分别生成目标文件，链接时，把生成的目标文件链接成可执行文件。

gcc

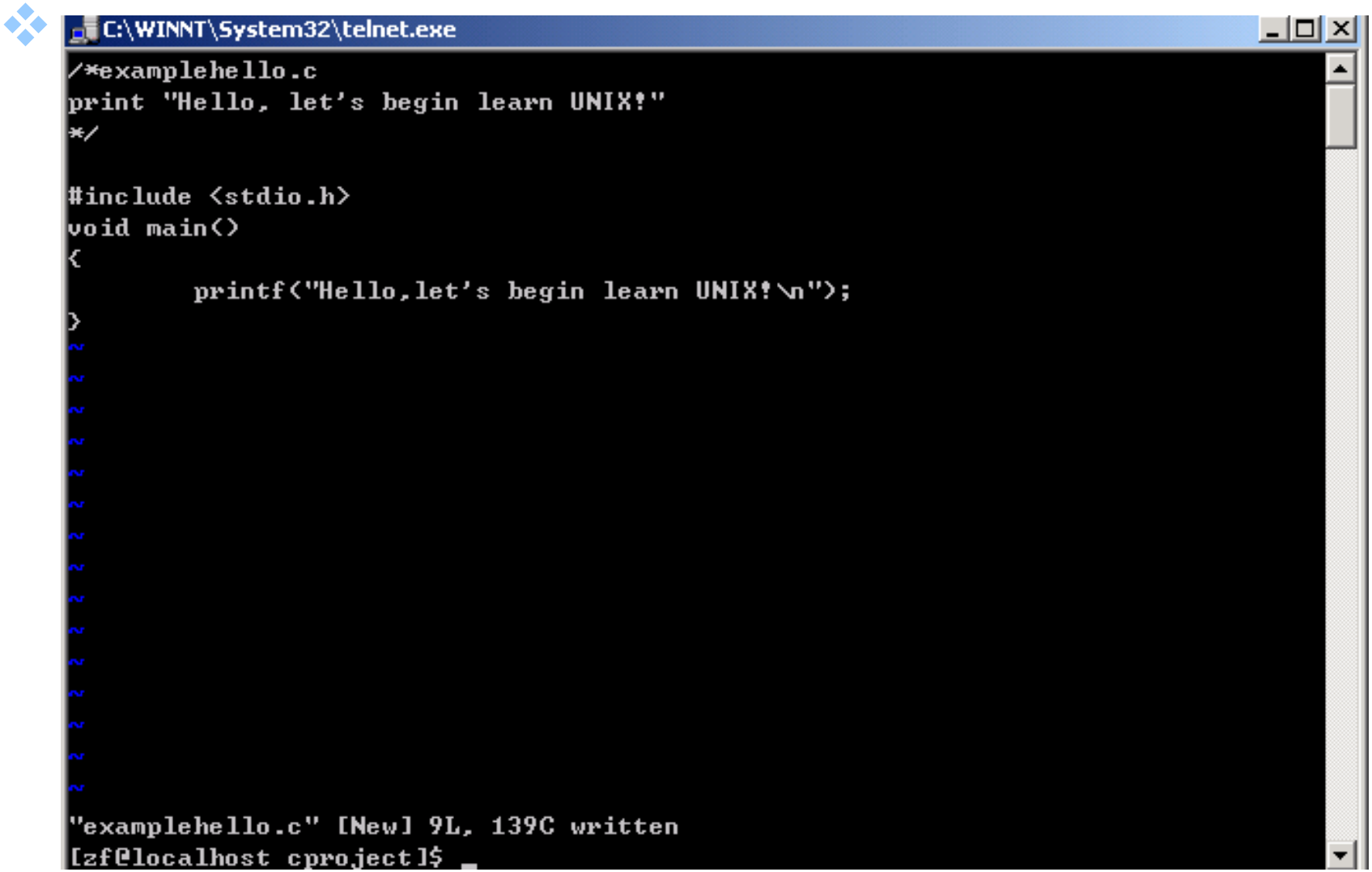
- gcc支持不同的源程序文件进行不同处理。

文件格式	文件含义	文件格式	文件含义
file.c	需预处理的C语言程序	file.cc	必须被预处理的C++源程序
file.i	不需预处理的C语言程序	file.cxx	必须被预处理的C++源程序
file.ii	不需预处理的C++语言程序	file.cpp	必须被预处理的C++源程序
file.h	C语言头文件，需编译和链接	file.s(S)	汇编代码

gcc

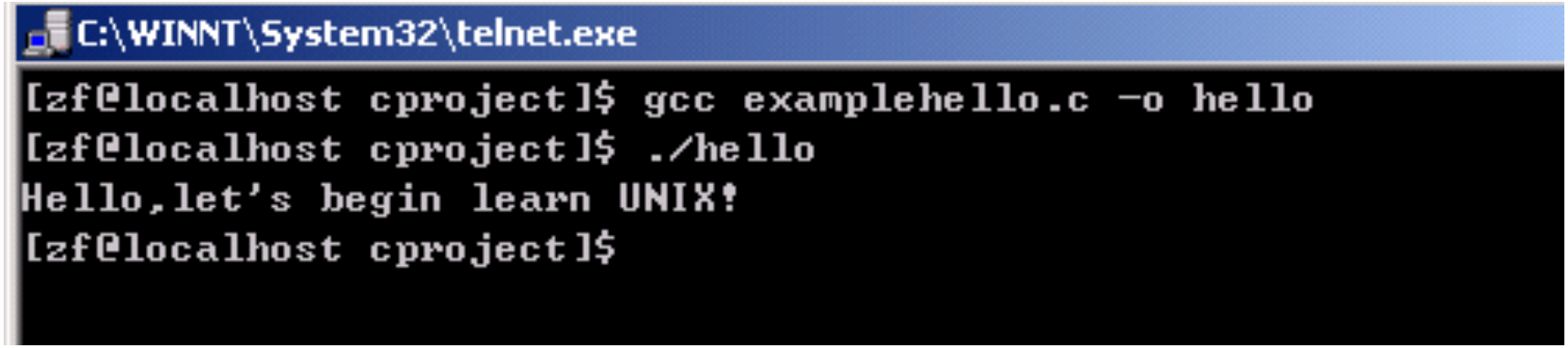
- 当不用任何选项编译一个程序时，gcc将会生成一个名为a.out的可执行文件。
- 可以使用-o编译选项为产生的可执行文件指定一个文件名代替a.out。
- 注意：当使用-o选项时，-o后面必须跟上一个文件名。
- 例如：文件examplehello.c
-

gcc


C:\WINNT\System32\telnet.exe
/*examplehello.c
print "Hello, let's begin learn UNIX!"
*/
#include <stdio.h>
void main()
{
printf("Hello,let's begin learn UNIX!\n");
}
"examplehello.c" [New] 9L, 139C written
[zf@localhost cproject]$

gcc

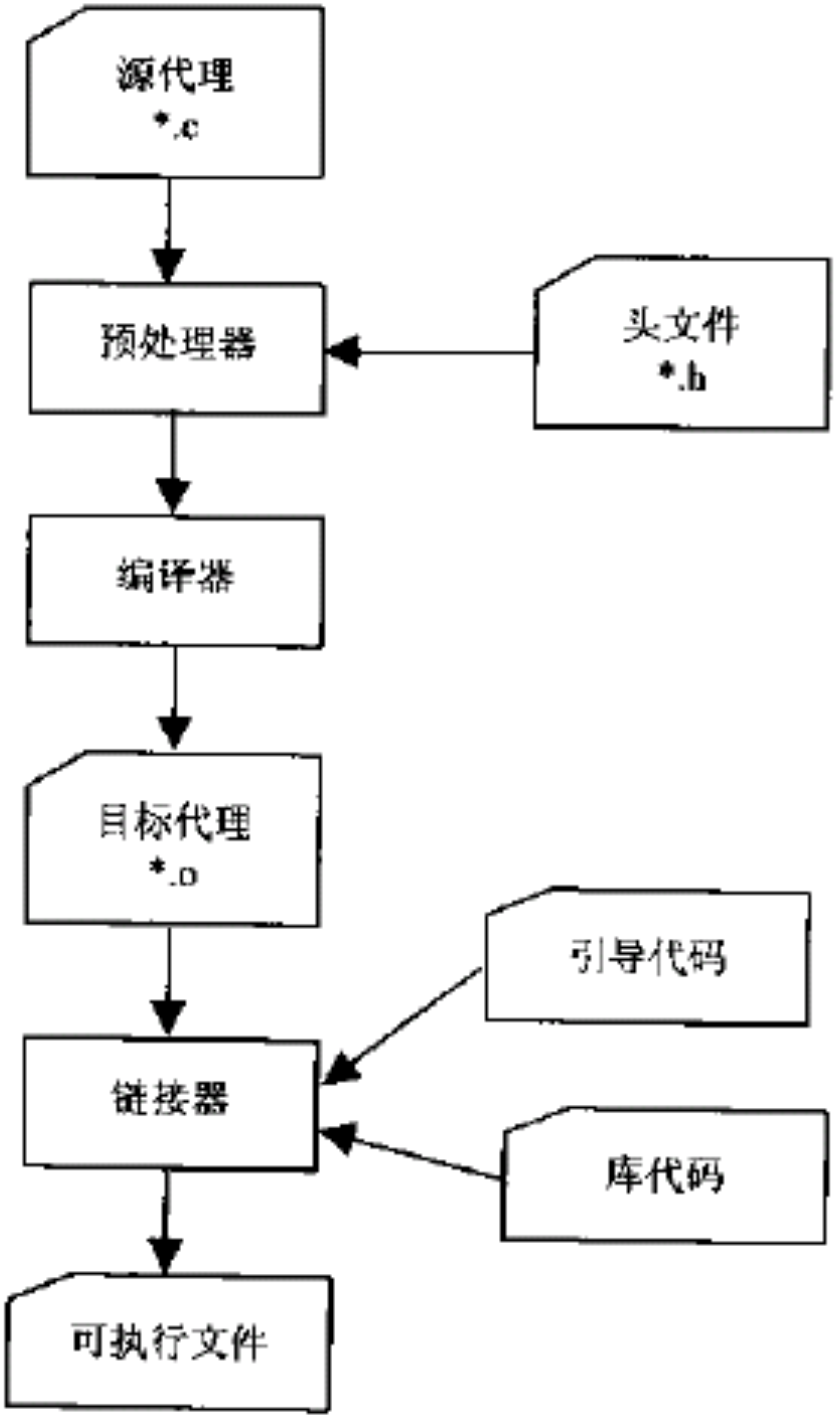
- 在命令行键入以下命令编译运行这段程序：
- $gcc examplehello.c –o hello
- $./hello



gcc

- gcc在上述过程中：
  - 运行预处理程序cpp在examplehello.c文件中插入包含文件stdio.h
  - 把经过预处理后的源代码编译成目标文件
  - 链接程序ld把目标文件于库文件链接并生成名为hello的二进制可执行文件。

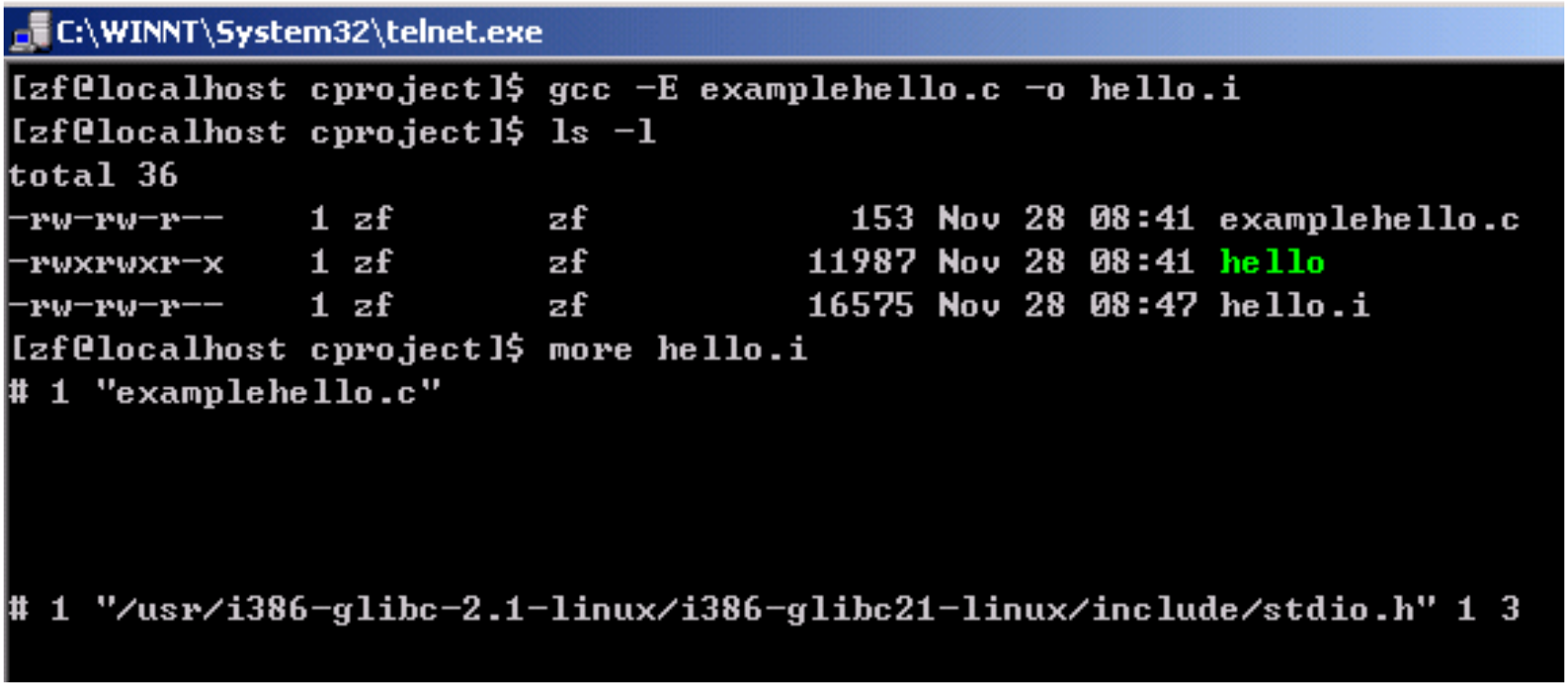
gcc



gcc

- 我们把上述过程分成几个步骤进行：
- $gcc –E examplehello.c –o hello.i
- $ls –l
- $more hello.i
- 第一行的-E可以使gcc在预处理后停止编译，并生成hello.i文件，将stdio.h的内容和其它应当被预处理的文件包含进来。

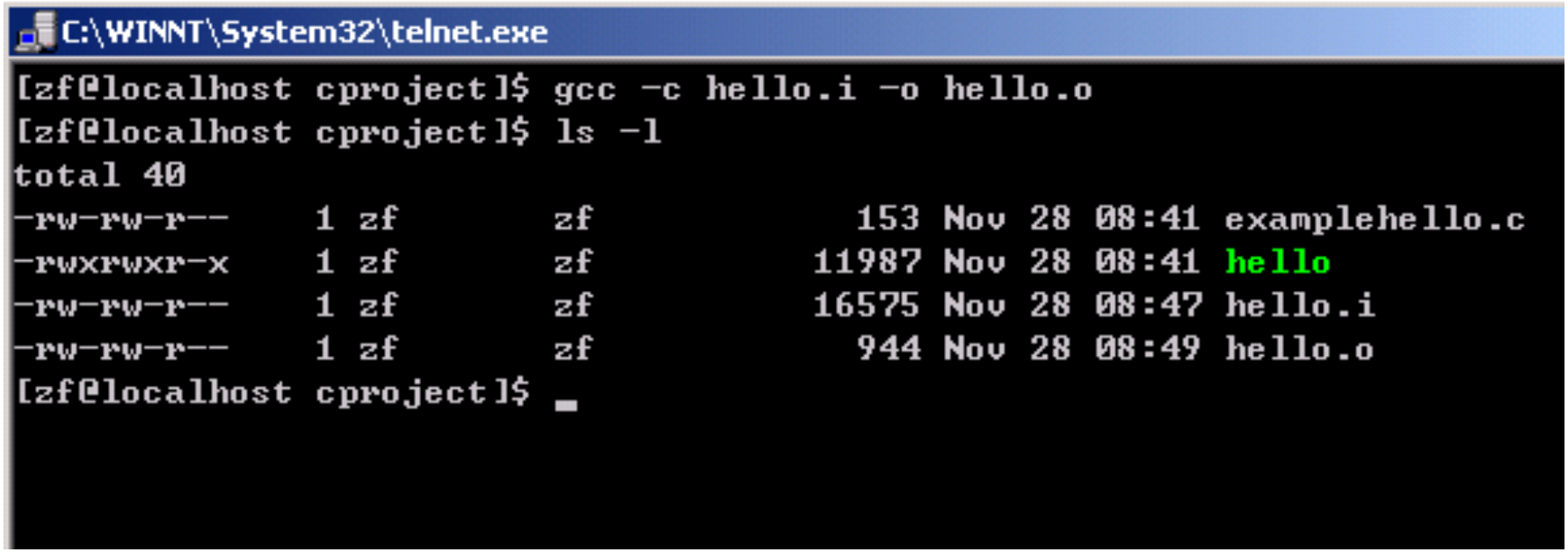
gcc


C:\WINNT\System32\telnet.exe
[zf@localhost cproject]$ gcc -E examplehello.c -o hello.i
[zf@localhost cproject]$ ls -l
total 36
-rw-rw-r-- 1 zf zf   153 Nov 28 08:41 examplehello.c
-rwxrwxr-x 1 zf zf 11987 Nov 28 08:41 hello
-rw-rw-r-- 1 zf zf 16575 Nov 28 08:47 hello.i
[zf@localhost cproject]$ more hello.i
1 "examplehello.c"
1 "/usr/i386-glibc-2.1-linux/i386-glibc21-linux/include/stdio.h" 1 3

gcc

- 继续键入下列命令：
- $gcc –c hello.i –o hello.o
- ls -l
- 第一行将hello.i编译为目标代码，因gcc识别“.i”文件为预处理后的C语言文件，因此，gcc自动跳过预处理而开始执行编译过程。

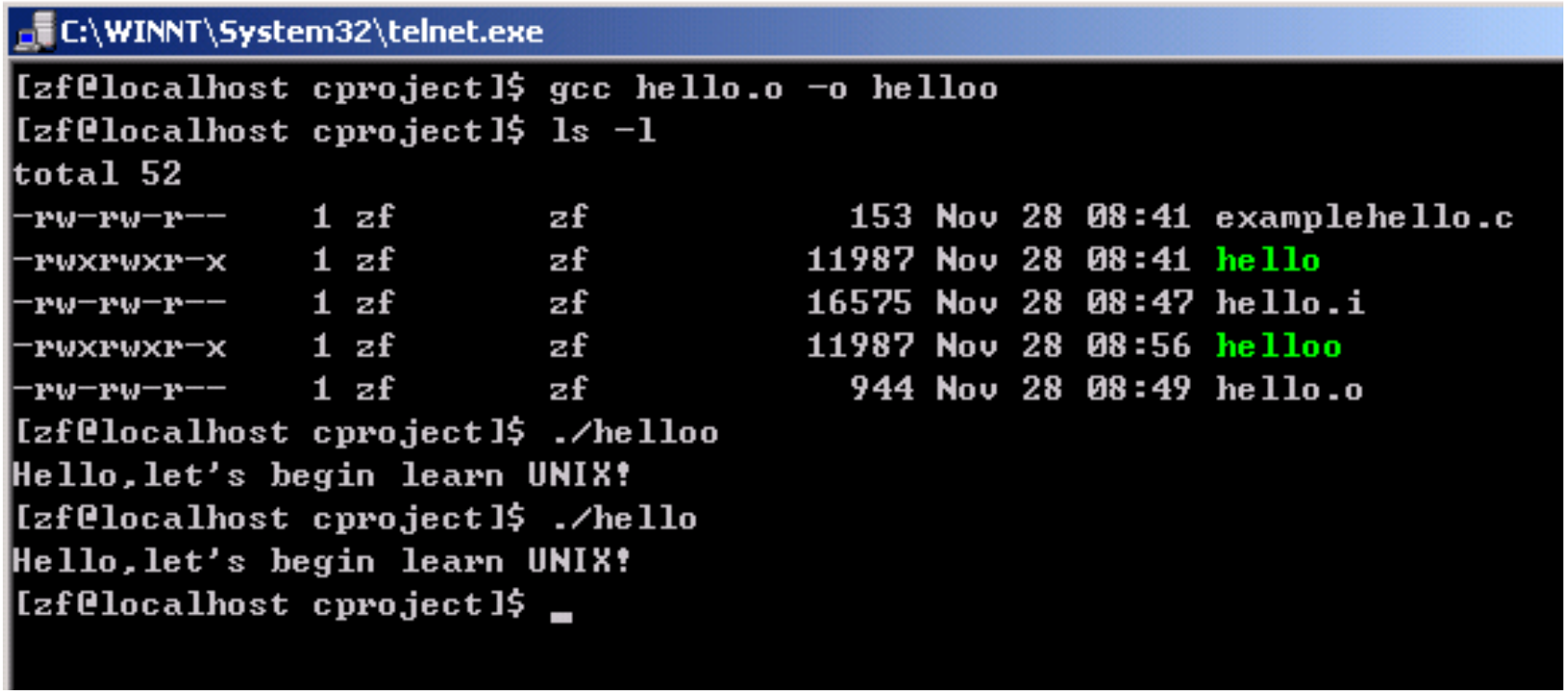
gcc


C:\WINNT\System32\telnet.exe
[zf@localhost cproject]$ gcc -c hello.i -o hello.o
[zf@localhost cproject]$ ls -l
total 40
-rw-rw-r-- 1 zf zf   153 Nov 28 08:41 examplehello.c
-rwxrwxr-x 1 zf zf 11987 Nov 28 08:41 hello
-rw-rw-r-- 1 zf zf 16575 Nov 28 08:47 hello.i
-rw-rw-r-- 1 zf zf   944 Nov 28 08:49 hello.o
[zf@localhost cproject]$

gcc

- 最后键入命令：
- $gcc hello.o –o helloo
- $ls –l
- $./helloo
- $./hello

gcc


C:\WINNT\System32\telnet.exe
[zf@localhost cproject]$ gcc hello.o -o helloo
[zf@localhost cproject]$ ls -l
total 52
-rw-rw-r-- 1 zf zf   153 Nov 28 08:41 examplehello.c
-rwxrwxr-x 1 zf zf 11987 Nov 28 08:41 hello
-rw-rw-r-- 1 zf zf 16575 Nov 28 08:47 hello.i
-rwxrwxr-x 1 zf zf 11987 Nov 28 08:56 helloo
-rw-rw-r-- 1 zf zf   944 Nov 28 08:49 hello.o
[zf@localhost cproject]$ ./helloo
Hello,let's begin learn UNIX!
[zf@localhost cproject]$ ./hello
Hello,let's begin learn UNIX!
[zf@localhost cproject]$

gcc

- 大多数C程序是由多个源代码文件组成，必须将源代码文件编译成目标代码后才能链接。
  - howdy.c使用了来自helper.c代码
  - helper.c程序代码：

```
/*
 *helper.c-Helper code for howdy.c
*/
#include <stdio.h>
void msg(void)
{
   printf(“This message sent from Jupier.\n”);
}
```

gcc

helper.c的头文件helper.h

```
/*
* helper.h – header for helper.c
*/
void meg(void)
```

gcc

howdy.c的源代码

```
/*
* howdy.c –Modifed “Hello, World!” program
*/
#include <stdio.h>
#include “helper.h”

int main(void)
{
   printf(“Hello, Linux programming world!\n”);
   msg();
   return 0;
}
```

gcc

- 正确编译howdy.c
- gcc howdy.c helper.c –o howdy
- 过程：
  - 预处理
  - 编译
  - 链接

gcc

- gcc主要参数选项：
- -x language　指定使用的语言（c、c++或汇编）
- -c　只对文件进行编译和汇编，但不链接
- -S　只对文件进行编译，但不汇编和链接
- -E　只对文件进行预处理，但不编译、汇编和链接
- -o file1 file2　将文件file2编译成可执行文件file1
- -l directory　用于指定所使用的库文件
- -I directory　为include文件的搜索指定路径
- -w　禁止警告信息
- -Wall　显示附加的警告信息

gcc

- -g 显示排错信息用于gdb调试
- -static 创建静态程序库，有利于程序的调试
- -O 优化编译代码
- -MM 输出一个make兼容的相关列表
- -v 输出gcc在编译过程中执行的每一个命令
- -werror 把所有警告转换为错误，以在警告发生时中止编译过程

- 注：更详细的选项可以提供执行man gcc或info gcc获得帮助。

gcc

- 需要注意：有些选项是多个字母，因此多个单字母选项不能组合使用，否则会产生歧义。例如：
- $gcc –p –g file.c
- $gcc –pg file.c

- 第一条gcc编译时为prof命令建立信息文件并把调试信息加入到可执行的文件中。
- 第二条gcc为gprof 命令建立信息文件。

gcc

- 增加函数库和包含文件
- -I{DIRNAME} 向GCC搜索包含（include）文件的路径中增加新目录
  - gcc myapp.c –I /home/fred/include –o myapp
- -L{DIRNAME} 同上，添加搜索路径，同时保证该路径比标准位置先被搜索。
  - gcc myapp.c –L/home/fred/lib –lnew –o myapp
  - -l 选项使得链接程序使用指定的函数库中的目标代码，把函数库命名为lib{名字}是UNIX的约定
  - 组合的用法：
  - gcc myapp.c –L/home/fred/lib –I/home/fred/include –lnew –o myapp

gcc

- 警告和出错选项
  - -pedantic 允许gcc发出遵循ANSI/ISO标准C语法的所有警告
  - -pedantic –errors 产生的错误是停止编译
  - -ansi 取消GNU扩展中所有与标准语法冲突部分
  - -Wall 能让gcc发出多种警告信息（如：从未所用过的变量）
  - -w 关闭所有警告信息

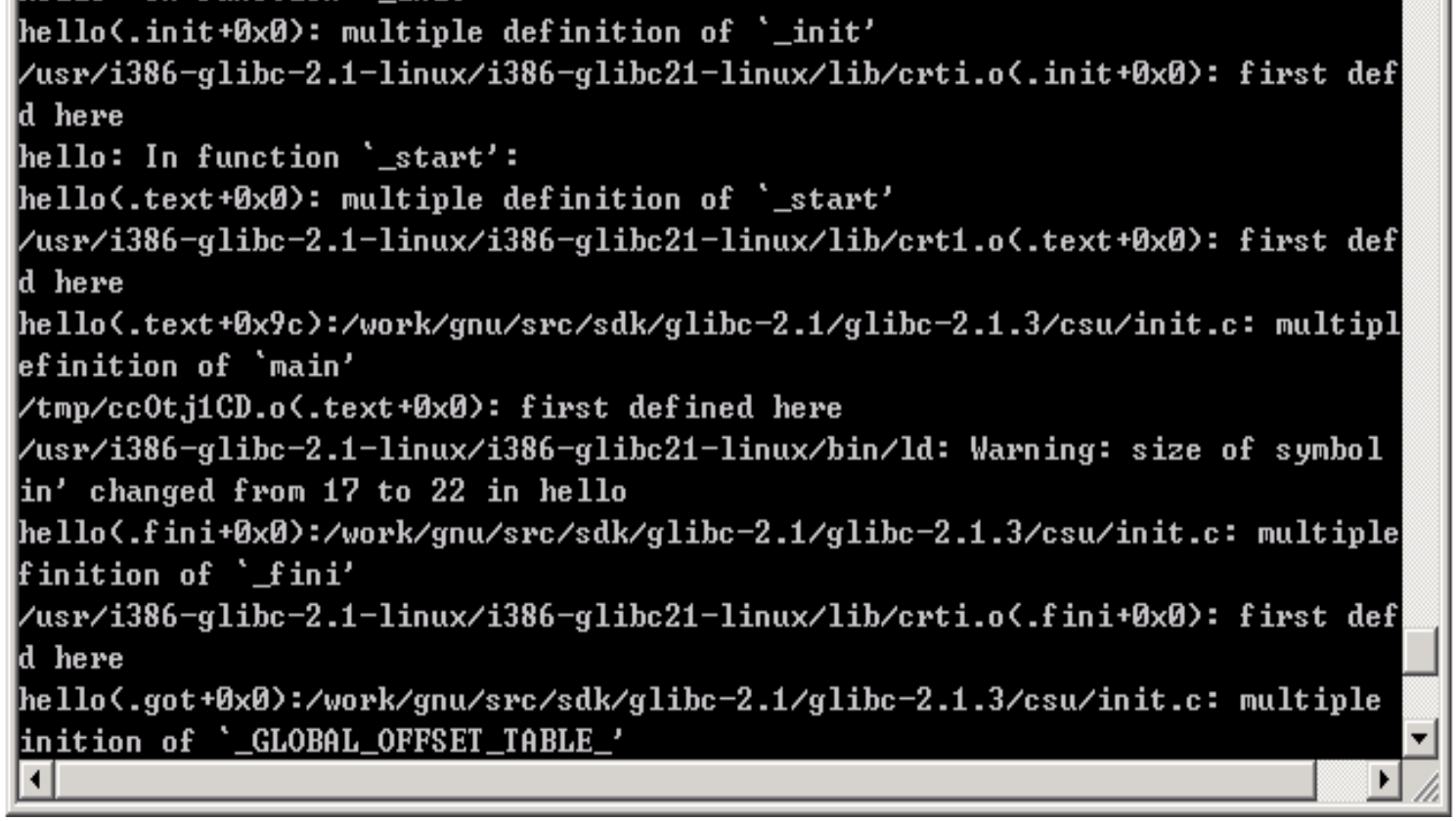
使用优化参数

- 当调用gcc编译C代码时，gcc试图用最少的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行次序，代码没有经过优化。可以采用gcc的优化选项控制gcc在编译时间花费和易于调试性的折中基础上产生更小更快的可执行文件。

-On选项

- 针对优化级别n生成不同的优化代码。其中，n是一个可有可无的整数。
- -O选项告诉gcc对源代码进行基本优化，提供程序执行的效率。

-On选项


C:\WINNT\System32\telnet.exe
[zf@localhost cproject]$ gcc examplehello.c -O2 hello
hello: In function `_init':
hello(.init+0x0): multiple definition of `_init'
/usr/i386-glibc-2.1-linux/i386-glibc21-linux/lib/crti.o(.init+0x0): first def
d here
hello: In function `_start':
hello(.text+0x0): multiple definition of `_start'
/usr/i386-glibc-2.1-linux/i386-glibc21-linux/lib/crt1.o(.text+0x0): first def
d here
hello(.text+0x9c):/work/gnu/src/sdk/glibc-2.1/glibc-2.1.3/csu/init.c: multipl
efinition of `main'
/tmp/ccOtj1CD.o(.text+0x0): first defined here
/usr/i386-glibc-2.1-linux/i386-glibc21-linux/bin/ld: Warning: size of symbol
in' changed from 17 to 22 in hello
hello(.fini+0x0):/work/gnu/src/sdk/glibc-2.1/glibc-2.1.3/csu/init.c: multiple
finition of `_fini'
/usr/i386-glibc-2.1-linux/i386-glibc21-linux/lib/crti.o(.fini+0x0): first def
d here
hello(.got+0x0):/work/gnu/src/sdk/glibc-2.1/glibc-2.1.3/csu/init.c: multiple
inition of `_GLOBAL_OFFSET_TABLE_'

使用调试选项

- gcc支持几种调试选项，其中最常用的是-g和-pg。-g告诉gcc产生被GNU调试程序gdb使用的信息以便程序调试。
- 如果使用-g和-O联用，可以在与最终的程序尽可能相近的情况下调试代码，同时使用这两个选项时必须清楚所写的某些源代码已经在优化时被gcc做了改动。
- -g gdb选项可以使代码包含gdb传有特性的调试信息以方便gdb的调试工作。
- 但任何一个调试选项都会使产生的可执行二进制文件的大小急剧增加，从而增加程序的执行开销。

使用调试选项

- -p选项和-pg，可以用在可执行的二进制文件中包含程序运行的一些统计信息。这些信息可以是程序员观察程序的各部分的运行特性，发现性能瓶颈，以便提高程序运行效率。
- -p在代码中键入prof工具能够识别的统计信息，而-pg则生成只能被GNU的prof(gprof)解释的信息。
- -a使gcc生成的可执行程序中包含计数代码块的记录执行次数的代码。
- -save-temps选项可以保存在编译过程中生成的中间文件，以利于调试。

Gcc的自定义符号

- 如果必须的程序中用到一些Linux独有的特性，可以把那些无法移植的程序代码以条件编译命令封装起来，用预定义符号_linux_完成：
- #ifdef _linux_
- /*与Linux特性相关的代码*/
- #endif

链接

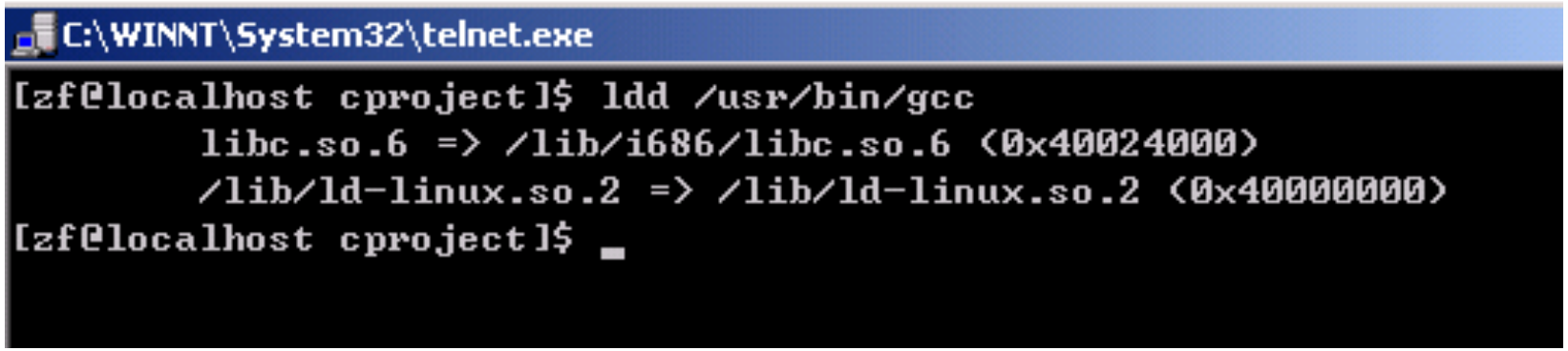
- 将所有步骤输出的目标文件组合起来。一般，库程序可以在/lib与/usr/lib/目录下找到。
- 当使用静态程序库时，链接程序会找出程序中所需的模块，然后将它们拷贝到执行文件内。
- 而共享程序库会在执行文件内留下一个标记，指明当程序执行到此处时，首先必须载入这个程序库。
- 共享程序库可以使执行文件变得更小，在程序运行时可以占用更小的内存。
- 对于glibc2，动态链接库为/lib/ld-linux.so.2。
- 要想保证系统总是连接共享程序库，应检查程序库是否在正确的位置，并且用户是否拥有读权限。

链接

- Linux中，静态程序库有类似与libname.a的名字，而共享程序库则是libname.so.x.y.z的形式。其中，x.y.z是版本序号。共享程序库通常都会有链接符号指向静态程序库和相关联的.so库。标准的程序库会包含共享与静态程序库两种形式。

链接

- 可以用ldd来查看程序需要用到的共享程序库。
- $ldd /usr/bin/gcc



链接

- Linux默认情况是链接共享程序库，如果Linux找不到这些共享程序库，会链接静态程序库。如果想在程序中强制使用静态库，用-static选项。
- 动态链接程序/装入程序ld.so会用到两个环境变量。$LD_LIBRARY_PATHZ在这些目录列表中可以搜索运行是的共享库。$LD_PRELOAD　是一个附件的有空格分隔的用户指定的共享库，它必须在所有库加载之前加载。

链接

- ld.so还通过两个配置文件来搜索共享库，标准目录/usr/lib和/lib是ld.so默认搜索目录。除此之外，文件/etc/ld.so.conf中列出了链接程序/ 装入程序搜索共享库时要查看的目录。/etc/ld.so.preload保存着环境变量$LD_PRELOAD的值，包含了一个由空格分隔的在执行程序之前要加载的共享库列表。

使用gcc的一个例子

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

void my_print1(char * string);
void my_print2(char * string);

void main()
{
        char string[] = "Hi, I want learn how to make c program!";

        my_print1(string);
        my_print2(string);
}

void my_print1(char * string)
{
        printf("The string is %s\n",string);
}

void my_print2(char * string)
{
        char * str;
        int size,size2,i;

        size = strlen(string);
        size2 = size-1;
        str = (char *) malloc (size+1);
        for ( i =0; i<size; i++)
                str[size2-i] = string[i];
        str[size]='\0';
        printf("The string printed backward is %s\n",str);
}
```

其它编译调试工具

- g++
- EGCS
- PGCC
- calls
- cproto
- indent
- gprof
- f2c和p2c

g++

- 是GNU提供的对C＋＋语言的编译器。
- g++与gcc具有相同的使用格式，即：
- g++ [options] [filenames]
- 其中，“options”与gcc的主要选项大体类似，而“filenames”则需要使用C++文件。
- 有几种扩展名：“.C”、“.cc”和“.cxx”。

EGCS

- EGCS(Experimental/Enhanced GNU Compiler System)
- 是gcc的发展方向，把Fortran的编译器集成进来。未来将加入Pascal。
- 构造清晰，把对gcc的各种改进都集成进来。

PGCC

- PGCC(Pentium gcc)
- 是针对Pentium CPU进行了编译器优化的Compiler 。使用JPEG压缩，解压缩测试最快比gcc快30％。
- 新版的PGCC都是基于EGCS，以patch的形式分布新版本。

calls

- calls调用gcc的与处理器来处理给出的源程序文件，然后输出这些文件里的函数调用树图。
- 当calls打印调用跟踪结果时，它在函数后面用中括号给出函数所在文件的文件名。
- main [test.c]

cproto

- cproto读入C源程序文件并自动为每个函数产生原型声明。
- 用它可以在写程序时节省大量用来定义函数原型的时间，并且输出可以重定向到一个调用函数原型的包含文件里。

indent

- 是Linux里包含的一个编程的实用工具。这个工具可以使你的代码产生美观的缩进格式以及指定如何格式化你的源代码。
- 联机帮助方式：
- indent -h

gprof

- Gprof是安装在Linux系统的/usr/bin目录下的一个程序。
- 它分析你的程序，可以告知程序的哪一个部分在执行时最费时间。可以告诉你程序里每个函数被调用的次数和每个函数执行时所占时间的百分比。
- 为了使用gprof，必须在编译程序时加上-pg选项。这将使程序在每次执行使产生一个gmon.out的文件，gprof使用这个文件产生分析信息。
- 在产生了gmon.out文件后，使用下列命令获得分析信息：
- gprof <filename>
- 参数filename是产生gmon.out文件的程序名。

f2c和p2c

- f2c和p2c是两个源代码转换程序，f2c把FORTRAN代码转换为C代码，p2c把Pascal代码转换成C代码。
- 在安装gcc时这两个程序都会被安装上去。
- 如果要转换的FORTRAN或Pascal程序比较小，可以直接使用f2c或p2c。而不需加然后选项。

使用gdb调试程序

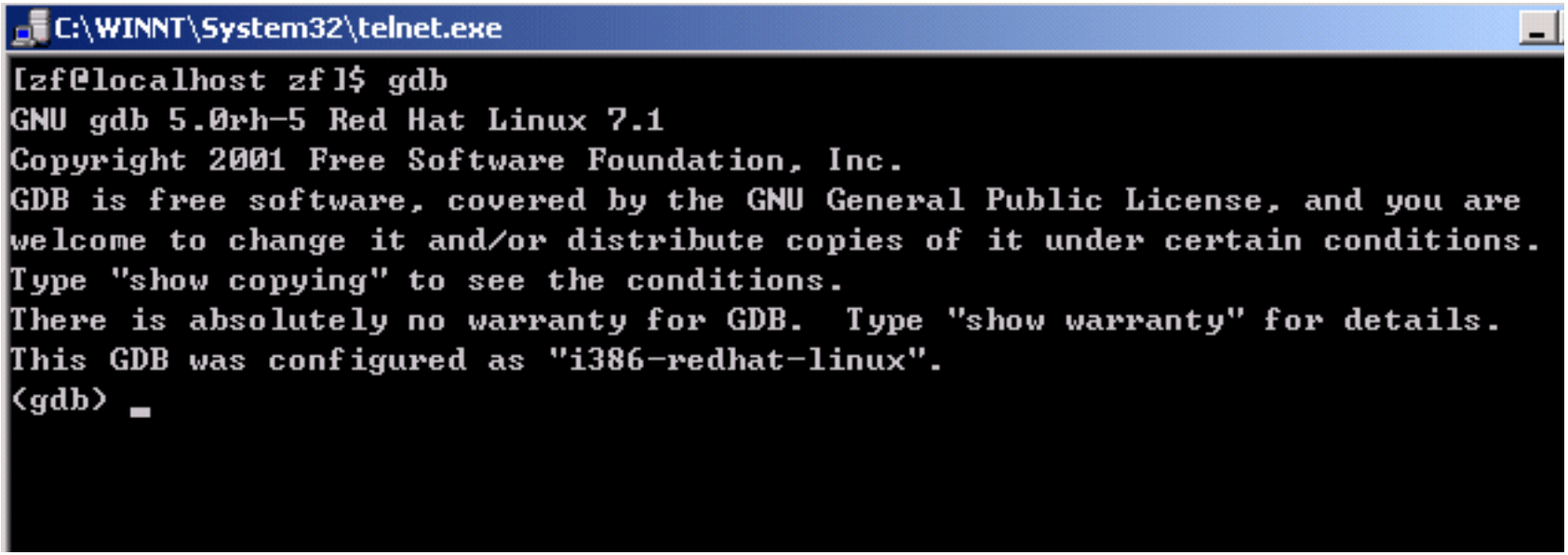
- Linux系统中包含了GNU的调试程序gdb，可以用来调试C和C++程序。
- **gdb的功能：**
- 控制程序，设置所有能影响程序运行的参数和环境；
- 控制程序在指定的条件下停止运行；
- 当程序停止时，可以检查程序的状态；
- 修正程序的错误，并重新运行程序；
- 动态监视程序中变量的值；
- 可以单步执行代码，观察程序的运行状态。

使用gdb调试程序

- GDB 的全称是 GNU Debuger。 是 linux 底下的一种免费的 debug 程式。GDB可以让你调试一个程序，包括让程序在你希望的地方停下，此时你可以查看变量，寄存器，内存及堆栈。更进一步可以修改变量及内存值。要使用 gdb ， 首先,在用户 compile 程式的時候, 要加上 -g 的选项。 （可以用-g, -g2, -g3）,g后面的数字越大, 可以 debug 的级别越高， 最高级别就是 -g3。

gdb的基本使用方法

- 启动gdb并获得版本和相关信息：
- $gdb



gdb的基本使用方法

- 如果指定想要调试的文件名，输入：
- $gdb filename
- gdb将装入名为filename的可执行文件。

gdb的基本使用方法

- 我們来学习 gdb。 用你喜爱的 editor 编辑一个叫做 debugit.c 的文件,内容如下：
- /*统计键盘输入的一行字符串中的单词个数*/
- #include <stdio.h>
- #include <stdlib.h>
- #include <string.h>
-
- int wordnum(char *s);
-
-

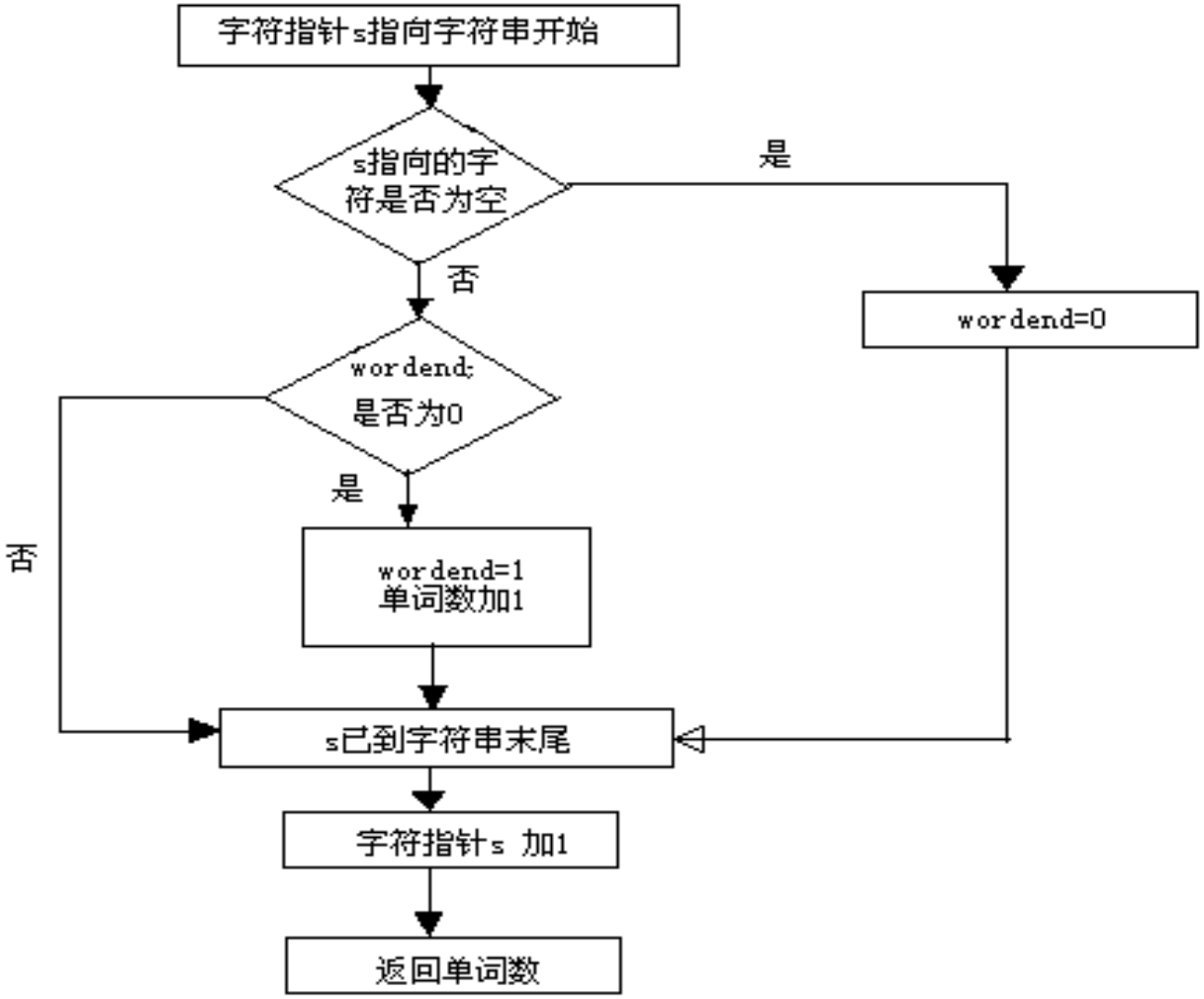
gdb的基本使用方法

```
main()
{
   char  str[80];
   int   num;

   printf(“Please input a string to count words…\n”);
   gets(str);
   num=wordnum(str);
   printf(“The number of words is %d\n”,num);
}
```

gdb的基本使用方法

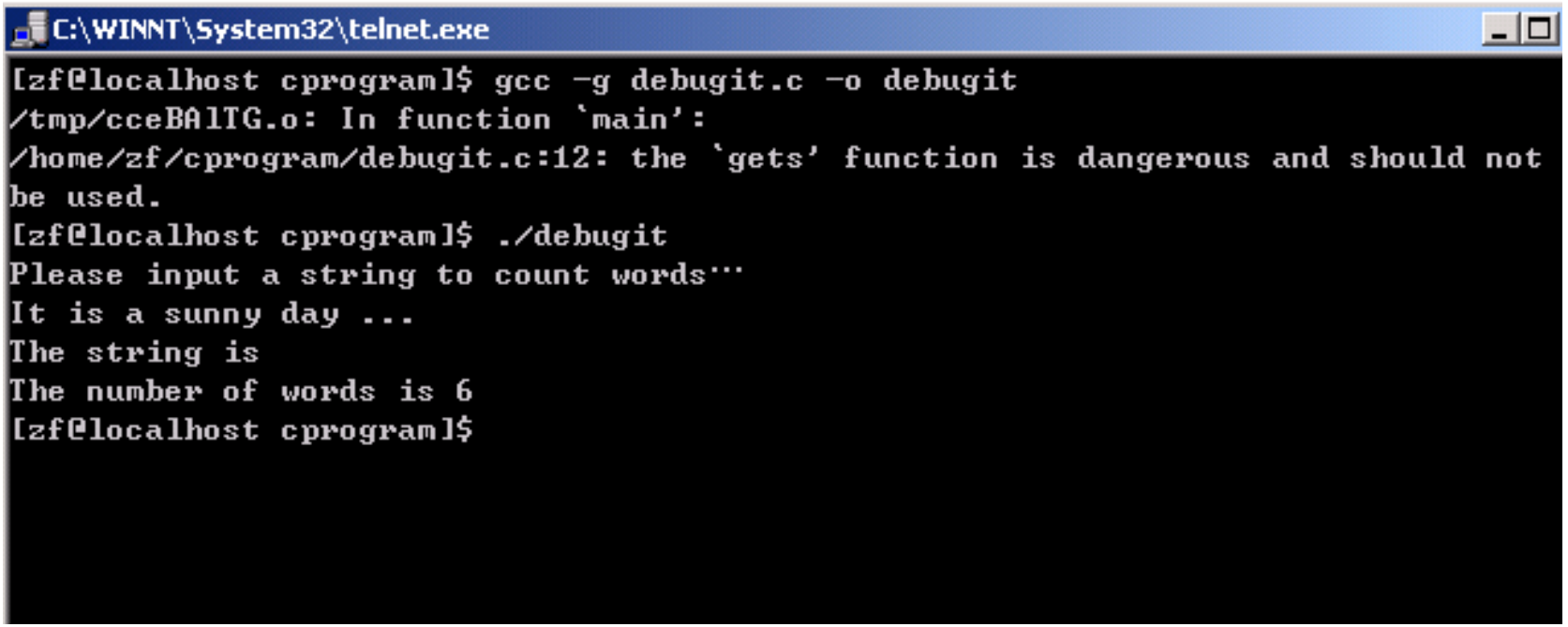
```
int wordnum(char *s)
{
   int j=0;
   int wordend=0;

   for (; *s!='\0'; s++)
   {
            if (*s ==' ')
                     wordend=0;
            else
                     if (!wordend)
                     {
                              wordend=1;
                              j++;
                     }
   }
   printf("The string is %s\n",s);
   return j;
}
```

gdb的基本使用方法


字符指针s指向字符串开始
s指向的字符是否为空
是
否
wordend=0
wordend;是否为0
是
否
wordend=1
单词数加1
s已到字符串末尾
字符指针s 加1
返回单词数

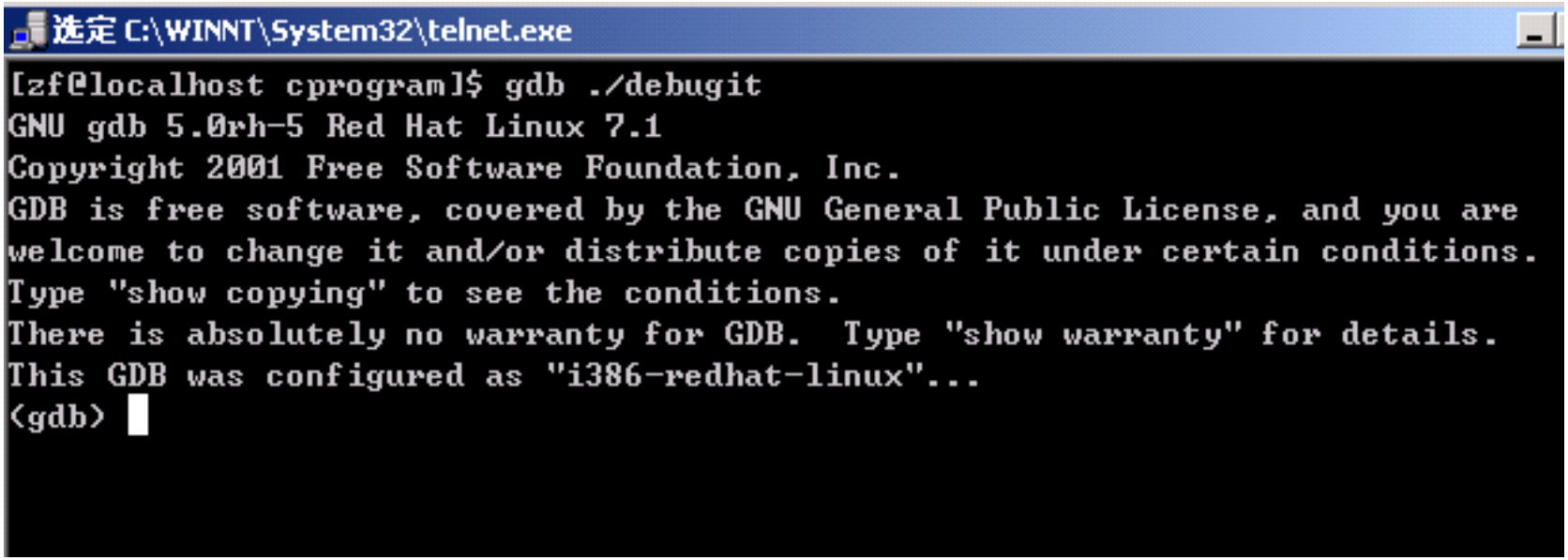
gdb的基本使用方法

- 经gcc编译、链接后可以执行，结果是：



gdb的基本使用方法

- 从上面信息中可以看出，程序在输出用户提供的字符串是有问题，输出了空串。
- 下面进行调试：



gdb的基本使用方法

- gdb在显示了版权信息后等待用户的命令输入，提示符是(gdb)。
- 使用list命令可以观察程序的源代码，list的格式是:
- list m,n
- 表示显示m行开始到n行结束的源代码。如果不带参数，则只显示十行代码。如果在gdb的提示符下键入回车键，则重复执行上条命令。

gdb的基本使用方法

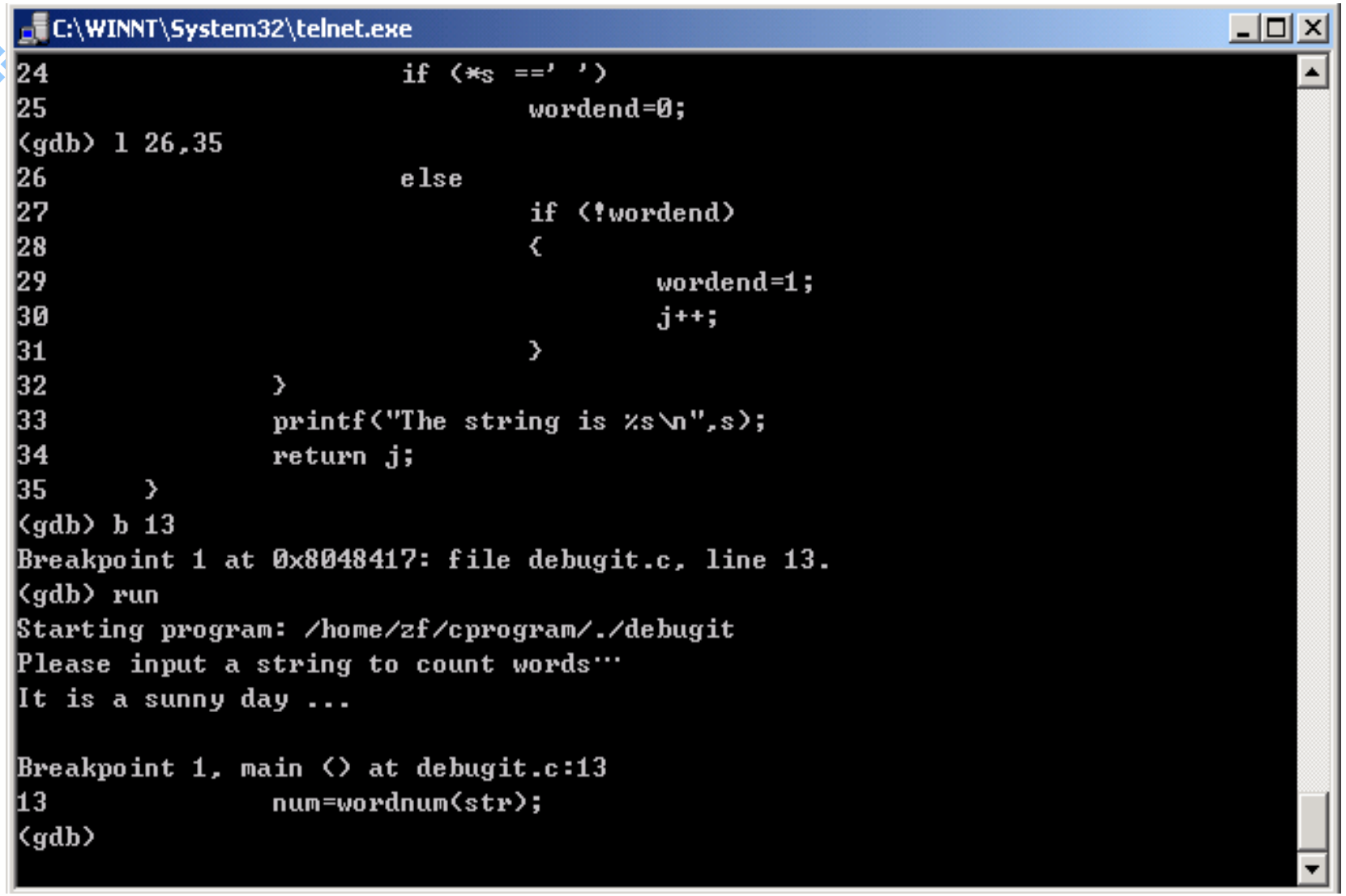
C:\WINNT\System32\telnet.exe

```
(gdb) l
2       #include <stdlib.h>
3       #include <string.h>
4
5       int wordnum(char *s);
6       main()
7       {
8               char  str[80];
9               int   num;
10
11              printf("Please input a string to count words···\n");
(gdb)
12              gets(str);
13              num=wordnum(str);
14              printf("The number of words is %d\n",num);
15      }
16
17      int wordnum(char *s)
18      {
19              int j=0;
20              int wordend=0;
21
(gdb)
```

gdb的基本使用方法

- 可以看出，在函数wordnum()中输出了用户提供的字符串，但是结果却是空串，因此必须观察函数内部的运行情况，为此，我们在13行，函数调用位置设置一个断点：
- (gdb) b 13

gdb的基本使用方法


C:\WINNT\System32\telnet.exe
24 if (*s ==' ')
25 wordend=0;
(gdb) l 26,35
26 else
27 if (!wordend)
28 {
29 wordend=1;
30 j++;
31 }
32 }
33 printf("The string is %s\n",s);
34 return j;
35 }
(gdb) b 13
Breakpoint 1 at 0x8048417: file debugit.c, line 13.
(gdb) run
Starting program: /home/zf/cprogram/./debugit
Please input a string to count words…
It is a sunny day ...
Breakpoint 1, main () at debugit.c:13
13 num=wordnum(str);
(gdb)

gdb的基本使用方法

- break（缩写为b）可以在某一个源代码行设置断点，程序运行到此时将停止，等待用户的进一步输入。Break命令的格式是：
- (gdb) b n
- 其中，n为要设置的断点的行号。
- 设置完断点后，就可以运行程序，而且程序应在断点处停止。

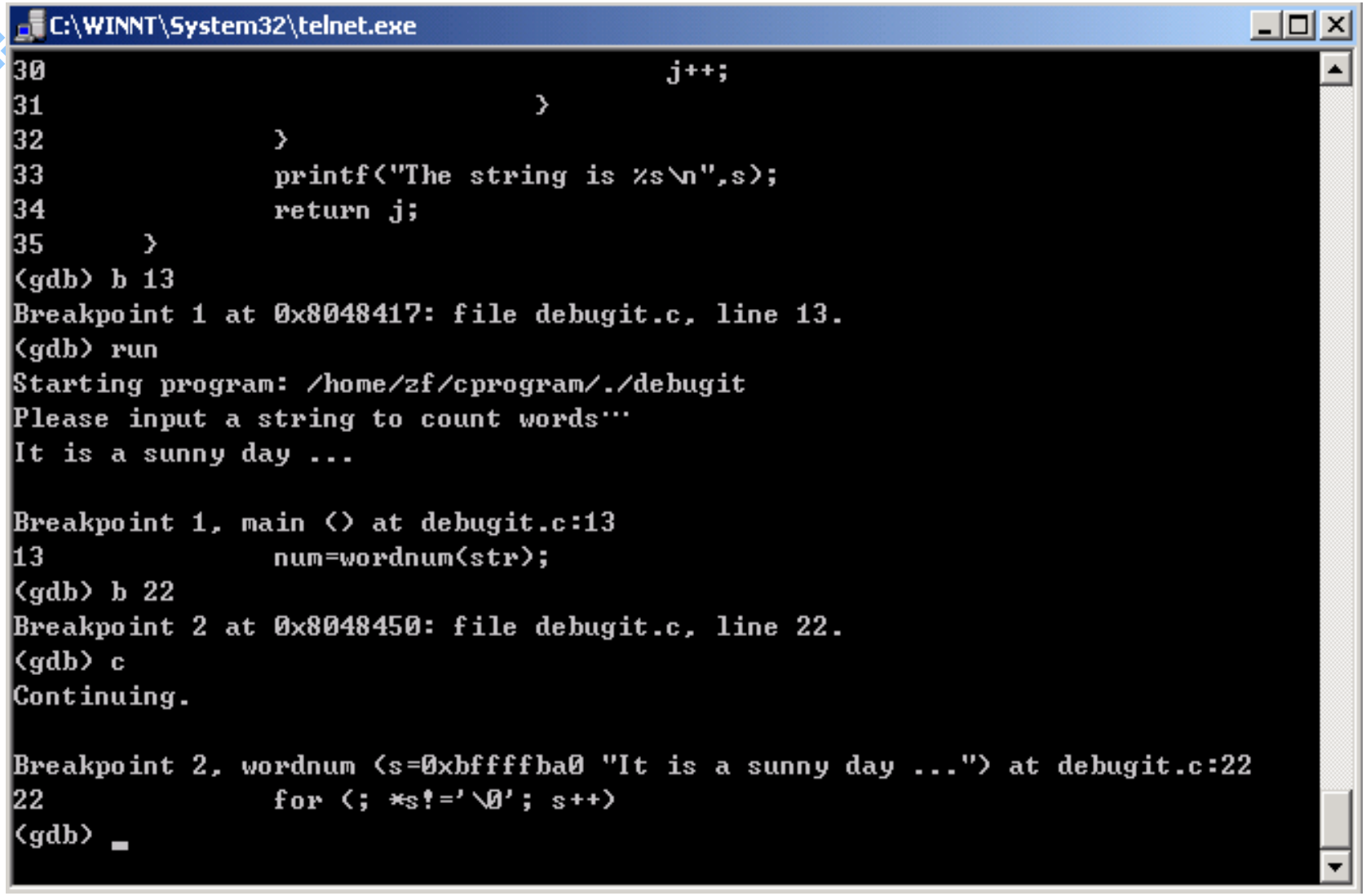
gdb的基本使用方法

- run命令可以在gdb中执行程序，如果程序运行需要参数，可以用set　args命令指定，也可以在run的命令行中指定。命un命名执行程序时将使用上一次run或set args命令指定的参数，如果想取消上次的参数，则执行不带参数的set args命令。

gdb的基本使用方法

- 为了检查函数内部的运行，就必须跟踪进入函数内部，可以使用step命令完成。
- 下面将跟踪函数wordnum的执行情况，为此，在函数内部设置断点以便观察相应的数据变化情况。
- 根据函数程序代码结构，在for循环体开始处即22行设置代码断点比较合适，这样可以每次循环开始时观察字符串s的值。

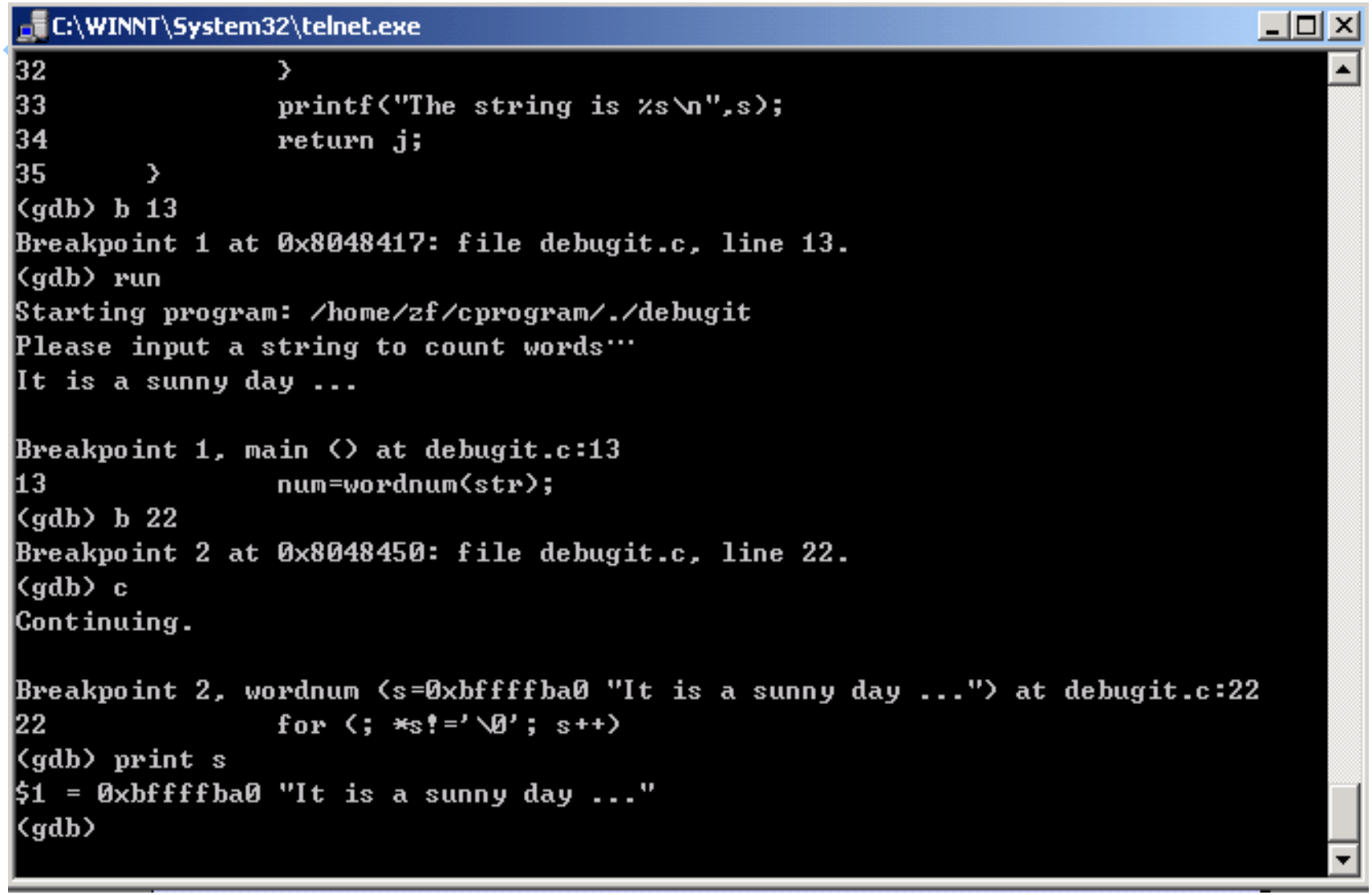
gdb的基本使用方法


C:\WINNT\System32\telnet.exe
30 j++;
31 }
32 }
33 printf("The string is %s\n",s);
34 return j;
35 }
(gdb) b 13
Breakpoint 1 at 0x8048417: file debugit.c, line 13.
(gdb) run
Starting program: /home/zf/cprogram/./debugit
Please input a string to count words···
It is a sunny day ...
Breakpoint 1, main () at debugit.c:13
13 num=wordnum(str);
(gdb) b 22
Breakpoint 2 at 0x8048450: file debugit.c, line 22.
(gdb) c
Continuing.
Breakpoint 2, wordnum (s=0xbffffba0 "It is a sunny day ...") at debugit.c:22
22 for (; *s!='\0'; s++)
(gdb)

gdb的基本使用方法

- 使用命令continue（缩写为c）可以使暂停的程序继续执行，而run命令则使程序从开始处运行。
- 此时，可以查看一下函数指针s指向的字符串的值，print后面加上变量的表达式，可以打印出此变量表达式的值，例如print s输出字符串s的值，而print *s则输出s指向的字符值。

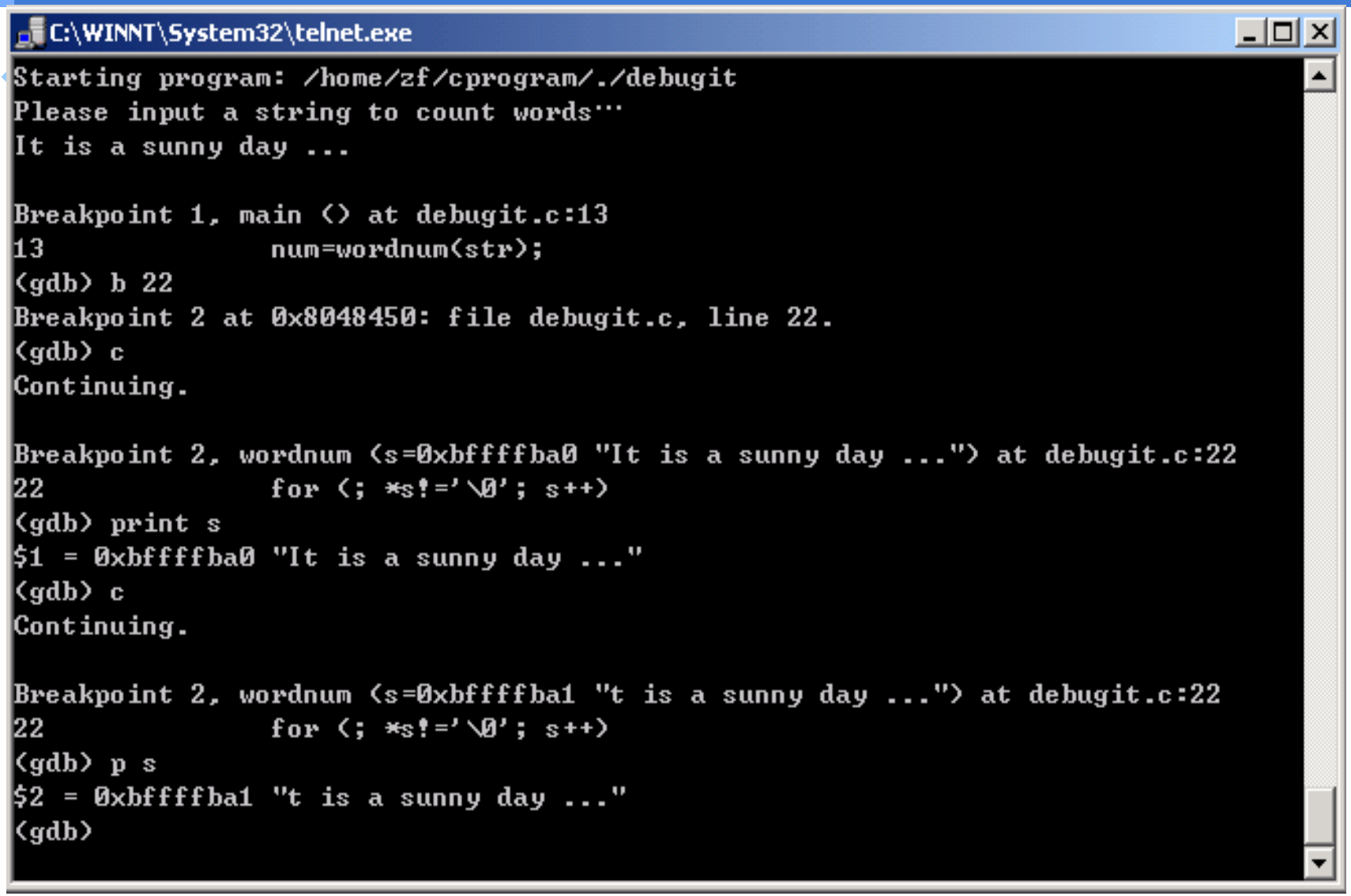
gdb的基本使用方法


C:\WINNT\System32\telnet.exe
32 }
33 printf("The string is %s\n",s);
34 return j;
35 }
(gdb) b 13
Breakpoint 1 at 0x8048417: file debugit.c, line 13.
(gdb) run
Starting program: /home/zf/cprogram/./debugit
Please input a string to count words···
It is a sunny day ...
Breakpoint 1, main () at debugit.c:13
13 num=wordnum(str);
(gdb) b 22
Breakpoint 2 at 0x8048450: file debugit.c, line 22.
(gdb) c
Continuing.
Breakpoint 2, wordnum (s=0xbffffba0 "It is a sunny day ...") at debugit.c:22
22 for (; *s!='\0'; s++)
(gdb) print s
$1 = 0xbffffba0 "It is a sunny day ..."
(gdb)

gdb的基本使用方法

- 被输入的字符串正确地存储在字符指针s指向的内存缓冲区内。$1表示变量的标号，你也可以通过命令print $num输出第num个变量值。不同标号的变量可以代表同一个程序中的同一个变量。ddb为用户保存了变量在运行中的动态值。
- 继续运行程序，指定停止在下一次循环开始处。

gdb的基本使用方法


C:\WINNT\System32\telnet.exe
Starting program: /home/zf/cprogram/./debugit
Please input a string to count words···
It is a sunny day ...

Breakpoint 1, main () at debugit.c:13
13   num=wordnum(str);
(gdb) b 22
Breakpoint 2 at 0x8048450: file debugit.c, line 22.
(gdb) c
Continuing.

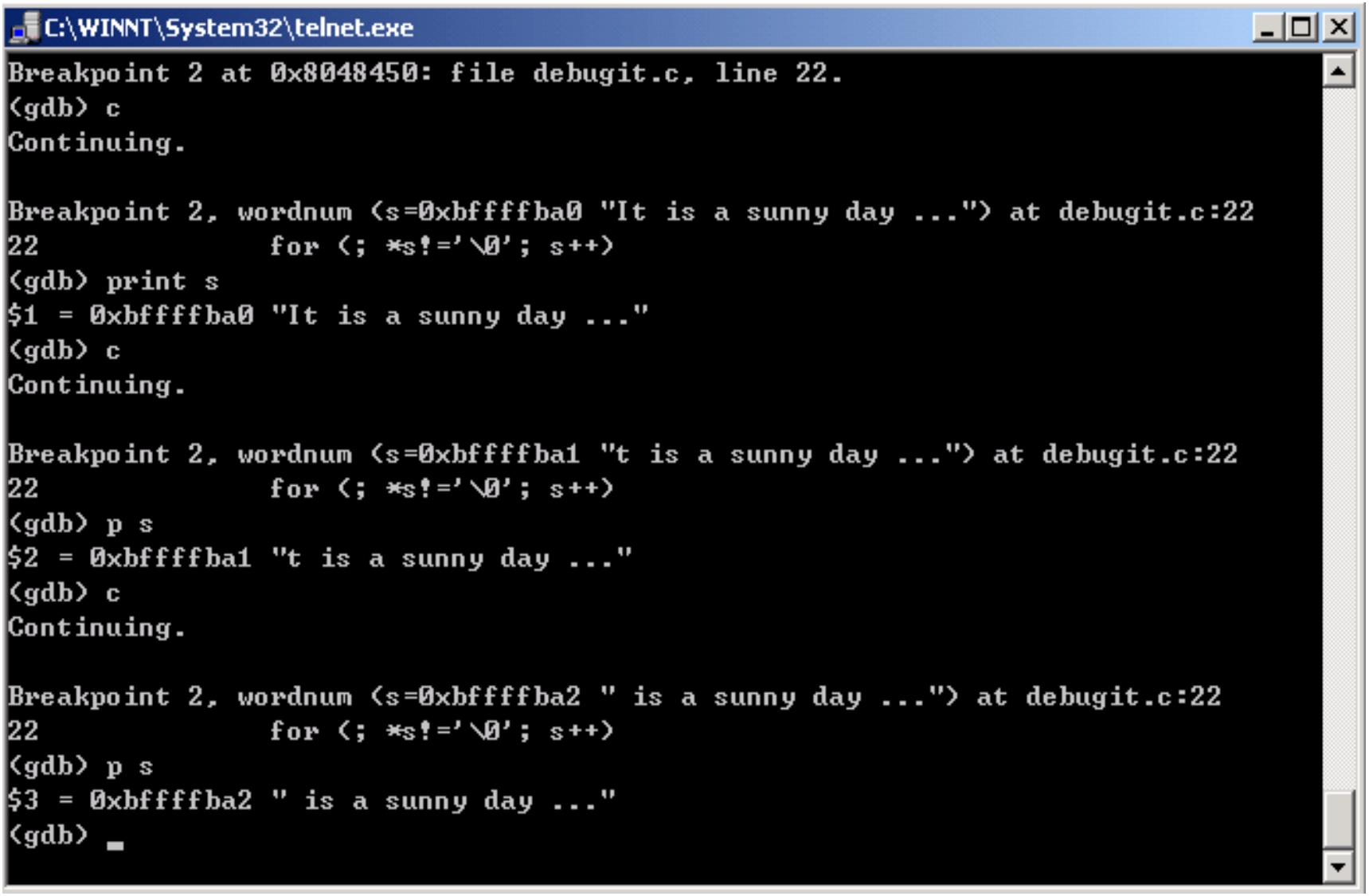
Breakpoint 2, wordnum (s=0xbffffba0 "It is a sunny day ...") at debugit.c:22
22   for (; *s!='\0'; s++)
(gdb) print s
$1 = 0xbffffba0 "It is a sunny day ..."
(gdb) c
Continuing.

Breakpoint 2, wordnum (s=0xbffffba1 "t is a sunny day ...") at debugit.c:22
22   for (; *s!='\0'; s++)
(gdb) p s
$2 = 0xbffffba1 "t is a sunny day ..."
(gdb)

gdb的基本使用方法

- 可以看到，s指向的字符串少了一个字符，原因是在每次循环时字符串指针向前移动了一个字符，从而导致字符丢失。
- 为了验证这个推测，可以继续运行程序到下一次循环开始处并观察字符串的值。

gdb的基本使用方法


C:\WINNT\System32\telnet.exe
Breakpoint 2 at 0x8048450: file debugit.c, line 22.
(gdb) c
Continuing.
Breakpoint 2, wordnum (s=0xbffffba0 "It is a sunny day ...") at debugit.c:22
22 for (; *s!='\0'; s++)
(gdb) print s
$1 = 0xbffffba0 "It is a sunny day ..."
(gdb) c
Continuing.
Breakpoint 2, wordnum (s=0xbffffba1 "t is a sunny day ...") at debugit.c:22
22 for (; *s!='\0'; s++)
(gdb) p s
$2 = 0xbffffba1 "t is a sunny day ..."
(gdb) c
Continuing.
Breakpoint 2, wordnum (s=0xbffffba2 " is a sunny day ...") at debugit.c:22
22 for (; *s!='\0'; s++)
(gdb) p s
$3 = 0xbffffba2 " is a sunny day ..."
(gdb)

gdb的基本使用方法

- 由此可以得知，当函数循环运行结束时，s指针将指向字符串结尾，因此在32行输出字符串时将得到空串。修改的办法是在函数内不改变参数s的值，使用另外一个字符串指针索引字符串中的字符。
- 修改程序，并重新编译和链接、运行程序，来验证所进行的修改是否有效。

gdb的基本使用方法

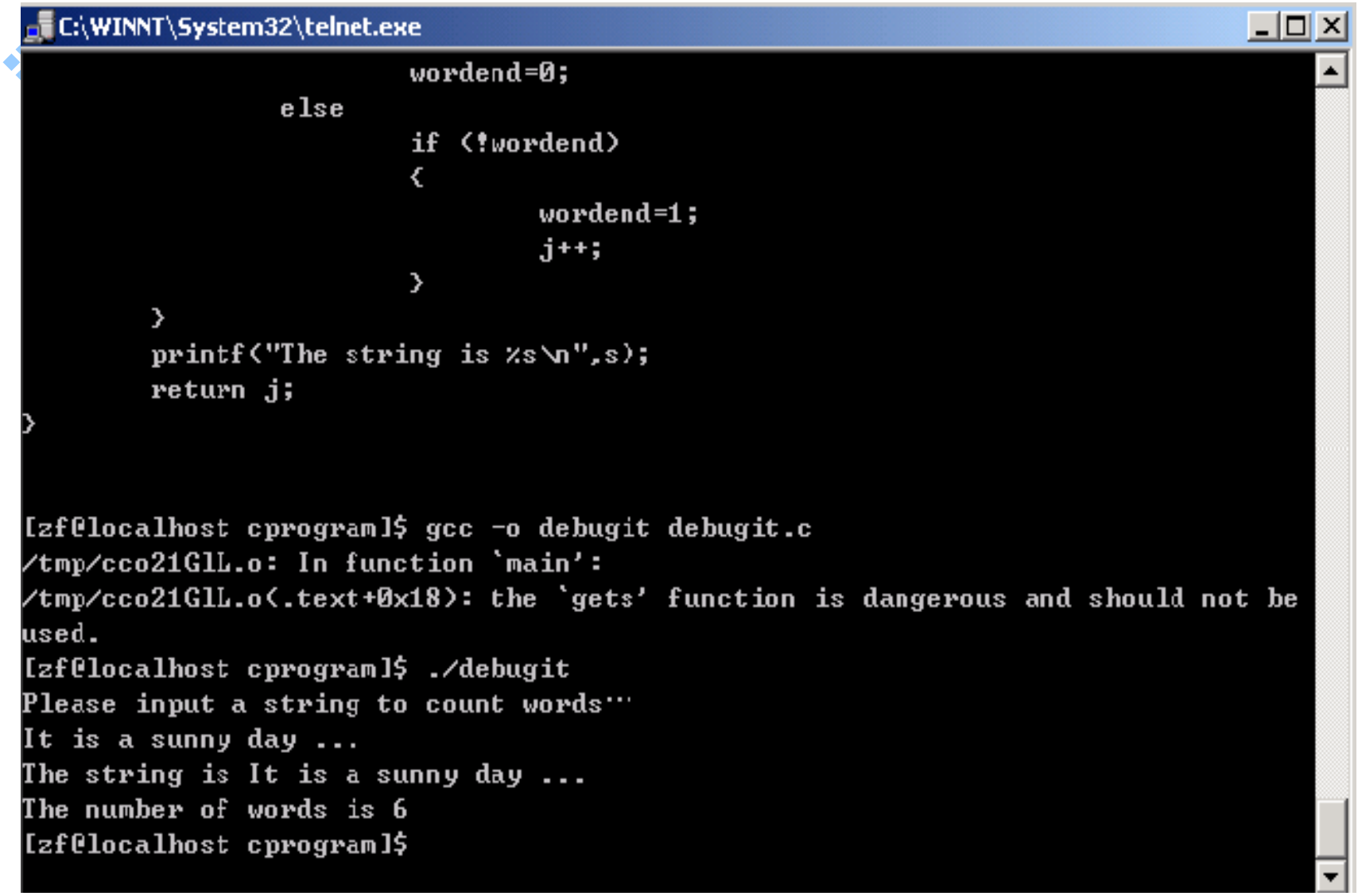
- 子函数的程序修改为：

```
int wordnum(char *s)
{
    int j=0;
    int wordend=0;
    char *p;

    for (p=s; *p!='\0'; p++)
    {
            if (*p ==' ')
                    wordend=0;
            else
```

gdb的基本使用方法

```
                    if (!wordend)
                    {
                            wordend=1;
                            j++;
                    }
    }
    printf("The string is %s\n",s);
    return j;
}
```

gdb的基本使用方法


C:\WINNT\System32\telnet.exe
wordend=0;
else
if (!wordend)
{
wordend=1;
j++;
}
}
printf("The string is %s\n",s);
return j;
}
[zf@localhost cprogram]$ gcc -o debugit debugit.c
/tmp/cco21GlL.o: In function `main':
/tmp/cco21GlL.o(.text+0x18): the `gets' function is dangerous and should not be used.
[zf@localhost cprogram]$ ./debugit
Please input a string to count words···
It is a sunny day ...
The string is It is a sunny day ...
The number of words is 6
[zf@localhost cprogram]$

gdb的基本命令

- gdb是功能强大的调试器，支持的调试命令非常丰富，可以实现不同的功能。这些命令包括从简单的文件装入到允许检查所调用的堆栈内容的复杂命令。
- 1、启动GDB
  2、载入想要调试的可执行文件 ： file
- 3、退出GDB ： quit
- 4、运行程序 ： run
- 5、查看程序信息 ： info
- 6、查看断点信息： info br
- 7、查看当前源程序：info source
- 8、查看堆栈信息：info stack
  9、列出源一段源程序 ： list
- 10、列出某个函数：list FUNCTION
- 11、接着前一次继续显示：list

gdb的基本命令

- 12、显示前一次之前的源程序：list –
- 13、显示另一个文件的一段程序：list FILENAME:FUNCTION
- 14、设置断点 ： break
- 15、在函数入口设置断点：break FUNCTION
- 16、在另一个源文件的某一行上设置断点break FILENAME:LINENUM
  17、在某个地址上设置断点 ：break *ADDRESS
- 18、其它断点操作
- 有效(Enabled)
  禁止(Disabled)
  一次有效(Enabled once)
  有效后删除(Enabled for deletion)
- 19、终止正在调试的程序 kill
- 20、不退出gdb而重新产生可执行文件 make

gdb的基本命令举例

```
#include <stdio.h>

int main()
{
        int i,j;

        j=0;
        for (i=0; i<10; i++)
        {
                j+=5;
                printf("now j == %d\n",j);
        }
        return 0;
}
[zf@localhost test]$ ./watch
now j == 5
now j == 10
now j == 15
now j == 20
now j == 25
now j == 30
now j == 35
now j == 40
now j == 45
now j == 50
[zf@localhost test]$
```

gdb的基本命令举例

C:\WINNT\System32\telnet.exe

```
(gdb) list
Line number 15 out of range; watch_test.c has 14 lines.
(gdb) list 1,10
1       #include <stdio.h>
2
3       int main()
4       {
5               int i,j;
6
7               j=0;
8               for (i=0; i<10; i++)
9               {
10                      j+=5;
(gdb)
11                      printf("now j == %d\n",j);
12              }
13              return 0;
14      }
(gdb) break 8
Breakpoint 1 at 0x80483d1: file watch_test.c, line 8.
(gdb) r
Starting program: /home/zf/test/watch

Breakpoint 1, main () at watch_test.c:8
8               for (i=0; i<10; i++)
(gdb)
```

gdb的基本命令举例

使用awatch可以为一个表达式设置观察点，在表达式值发生改变时，程序就会停止运行。

```
(gdb) awatch j
Hardware access (read/write) watchpoint 2: j
(gdb) cont
Continuing.
```

```
now j == 5
Hardware access (read/write) watchpoint 2: j
```

gdb的基本命令举例

使用clear可以用来清除断点。

```
(gdb) l
3       int main()
4       {
5               int i,j;
6
7               j=0;
8               for (i=0; i<10; i++)
9               {
10                      j+=5;
11                      printf("now j == %d\n",j);
12              }
(gdb) clear 8
Deleted breakpoint 1
(gdb) break 7
Breakpoint 4 at 0x80483ca: file watch_test.c, line 7.
(gdb) clear 8
No breakpoint at 8.
(gdb)
```

gdb的基本命令举例

- 可以使用condition在满足一定条件的情况下才在指定的行上设置断点。

```
(gdb) clear 8
Deleted breakpoint 1
(gdb) break 7
Breakpoint 4 at 0x80483ca: file watch_test.c, line 7.
(gdb) clear 8
No breakpoint at 8.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/zf/test/watch

Breakpoint 4, main () at watch_test.c:7
7                 j=0;
(gdb) condition 4 1
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/zf/test/watch

Breakpoint 4, main () at watch_test.c:7
7                 j=0;
(gdb) condition 3 1
No breakpoint number 3.
(gdb)
```

gdb的基本命令举例

- 用ignore可以在一定范围内忽略用户设定的断点。

```
(gdb) break 10
Breakpoint 1 at 0x80483e0: file watch_test.c, line 10.
(gdb) break 11
Breakpoint 2 at 0x80483e4: file watch_test.c, line 11.
(gdb) r
Starting program: /home/zf/test/watch

Breakpoint 1, main () at watch_test.c:10
10                      j+=5;
(gdb) ignore 1 3
Will ignore next 3 crossings of breakpoint 1.
(gdb) continue
Continuing.

Breakpoint 2, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
(gdb) continue
Continuing.
now j == 5

Breakpoint 2, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
(gdb)
```

gdb的基本命令举例

```
(gdb) cont
Continuing.
now j == 10

Breakpoint 2, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
(gdb) cont
Continuing.
now j == 15

Breakpoint 2, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
(gdb) cont
Continuing.
now j == 20

Breakpoint 1, main () at watch_test.c:10
10                      j+=5;
(gdb)
```

gdb的基本命令举例

```
6
7                 j=0;
8                 for (i=0; i<10; i++)
9                 {
10                        j+=5;
(gdb) tbreak 10
Breakpoint 1 at 0x80483e0: file watch_test.c, line 10.
(gdb) run
Starting program: /home/zf/test/watch
main () at watch_test.c:10
10                        j+=5;
(gdb) continue
Continuing.
now j == 5
now j == 10
now j == 15
now j == 20
now j == 25
now j == 30
now j == 35
now j == 40
now j == 45
now j == 50

Program exited normally.
(gdb)
```

为程序设置断点和观察点

- 断点的作用是当程序运行到断点处时，可以中断当前的执行过程。可以针对每个断点设置复杂的条件以决定断点在何时起作用。
- 可以使用break命令在程序中设置断点，可以在指定的行、函数、甚至在某一个地址上设置断点。在支持异常处理的语言中，还可以在异常发生的地方设置断点，来观察错误的发生情况。

为程序设置断点和观察点

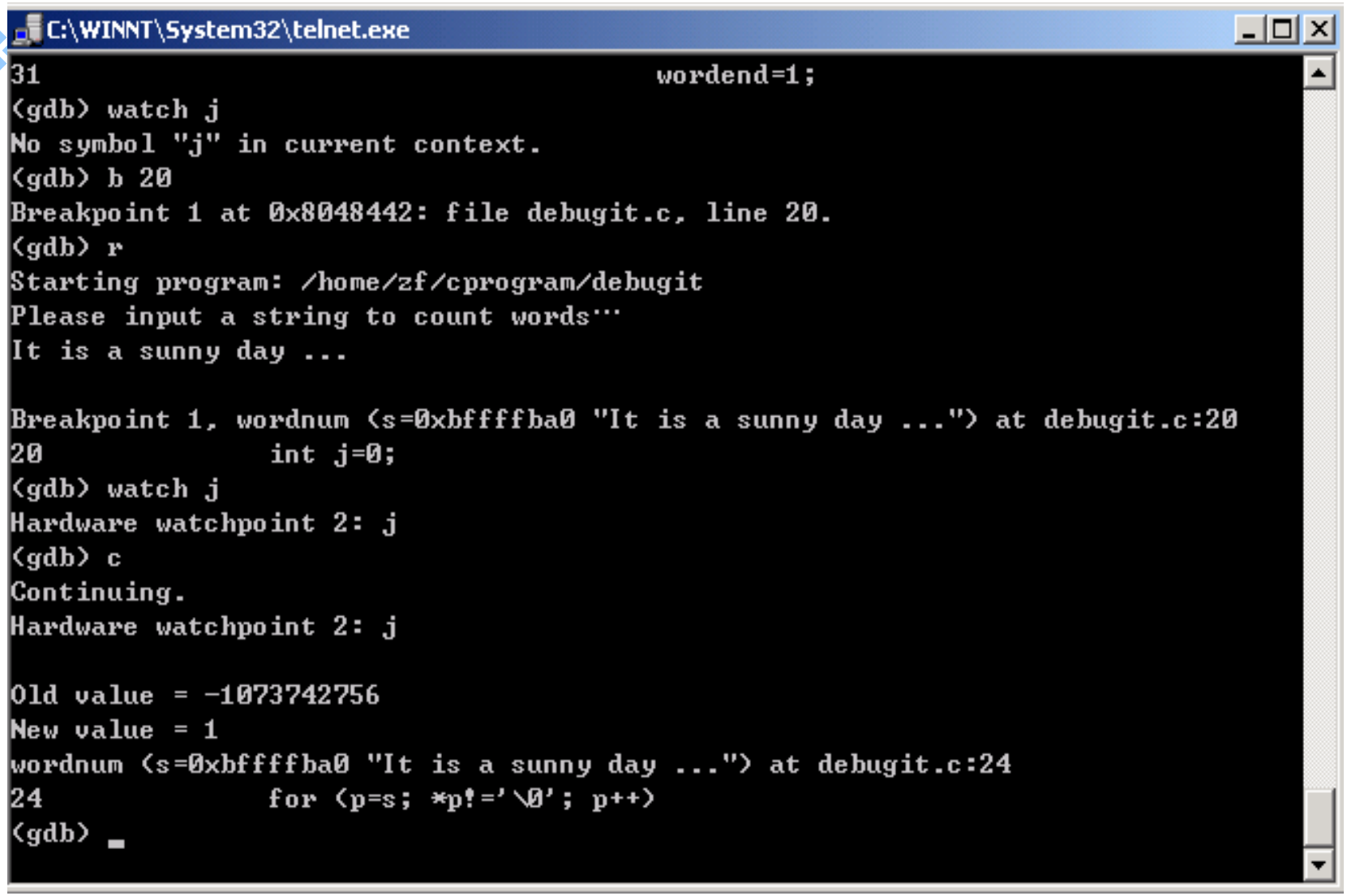
- 设置监视点动态观察一个变量的值： watch

  当你调试一个很大的程序，并且在跟踪一个关键的变量时，发现这个变量不知在哪儿被改动过，如何才能找到改动它的地方。这时你可以使用watch命令。简单地说，监视点可以让你监视某个表达式或变量，当它被读或被写时让程序断下。watch命令的用法如下：

  watch EXPRESSION

- 观察点在程序中某个表达式的值发生变化时，停止程序并显示表达式的值。

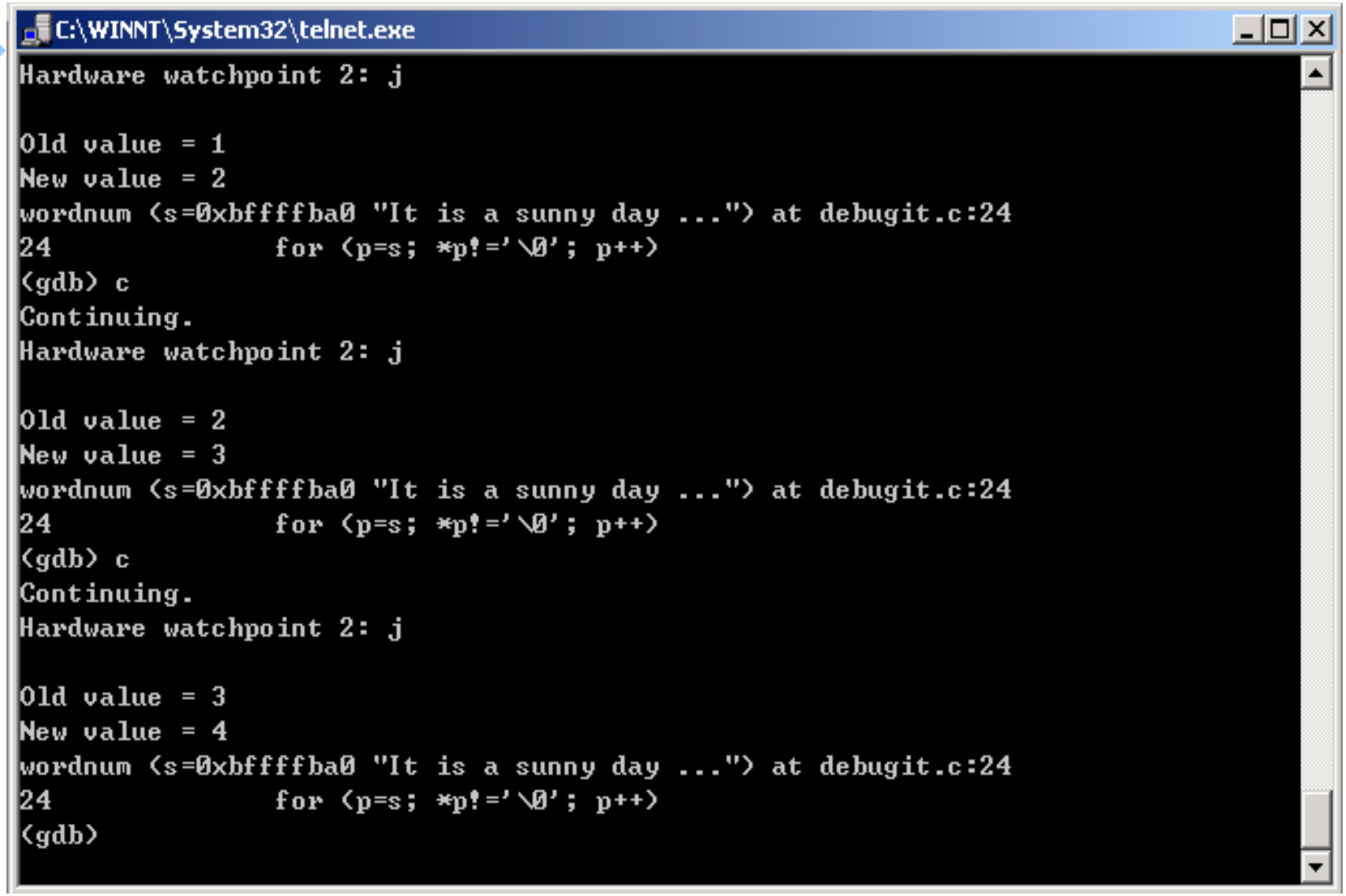
为程序设置断点和观察点



为程序设置断点和观察点

- 首先在主函数的执行开始处设置断点，并运行程序，然后对函数wordnum()中的变量j设置观察点。可以看到gdb无法在当前的执行上下文中找到变量j，为此在函数wordnum() 的开始处设置断点，并对变量j 设置观察点。
- 继续执行程序，当j值改变时，程序中断并输出j的值。

为程序设置断点和观察点



关于数据的命令

- 命令display用于显示表达式的值，使用该命令之前，应使用break设置断点。这样，程序每运行到这个断点时都会显示该表达式的值。

关于数据的命令

```
(gdb) break 11
Breakpoint 1 at 0x80483e4: file watch_test.c, line 11.
(gdb) display j*6
No symbol "j" in current context.
(gdb) r
Starting program: /home/zf/test/watch

Breakpoint 1, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
(gdb) display j*6
1: j * 6 = 30
(gdb) continue
Continuing.
now j == 5

Breakpoint 1, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
1: j * 6 = 60
(gdb) continue
Continuing.
now j == 10

Breakpoint 1, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
1: j * 6 = 90
(gdb)
```

关于数据的命令

命令info display用于显示当前所有要显示值表达式的有关信息。

```
(gdb) display j*3
1: j * 3 = 15
(gdb) display j*5
2: j * 5 = 25
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
2:   y  j * 5
1:   y  j * 3
(gdb) continue
Continuing.
now j == 5

Breakpoint 1, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
2: j * 5 = 50
1: j * 3 = 30
(gdb) continue
Continuing.
now j == 10

Breakpoint 1, main () at watch_test.c:11
11                      printf("now j == %d\n",j);
2: j * 5 = 75
1: j * 3 = 45
(gdb)
```

关于数据的命令

- 其它命令：
- delete display ：用于删除一个要显示值的表达式。
- disable display ：使一个要显示值的表达式暂时无效，但并不删除该表达式的显示。
- enable display ：和disable display命令功能相反，用于使显示值被屏蔽的表达式恢复显示。
- undisplay ：用于结束某个表达式的显示。
- whatis ：用于显示某个表达式的数据类型。
- print ：用于打印表达式的值，也可以用于打印内存中从某个变量开始一段区域的内容。

关于文件的命令

- 命令list
- list命令可使用的参数有：
- LINENUM
- FILE:LINUNUM
- FUNCTION
- FILE:FUNCTION
- ADDRESS

关于文件的命令

- 命令forward
- 用于从当前行开始向后查找第一个匹配某个字符串的程序行。
- 命令load
- 用于动态地往正在调试的程序中装入文件，并记录它的符号表，准备连接。
- 命令search
- 命令reverse-search

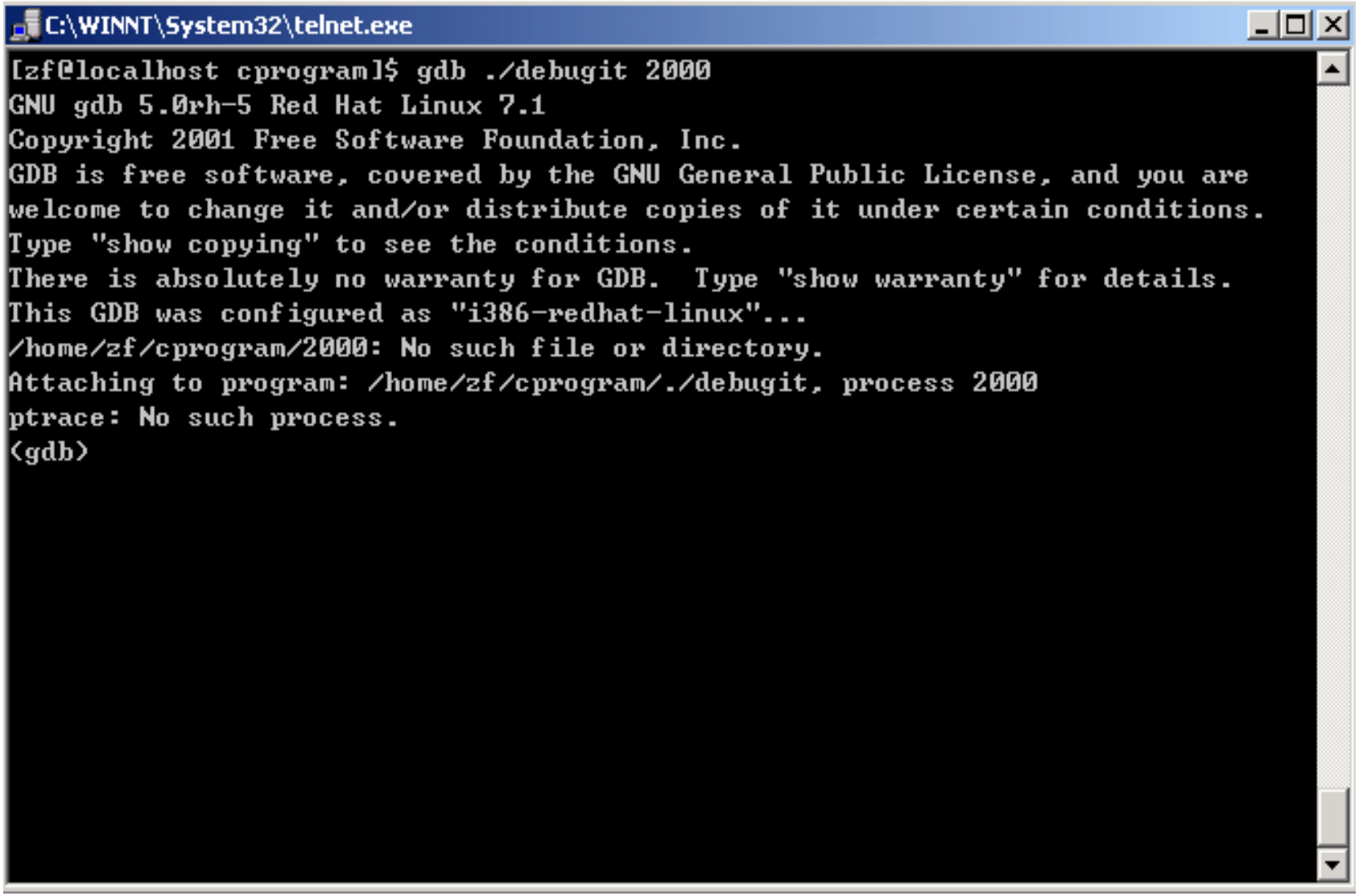
Gdb的命令特性

- gdb支持很多与shell程序一样的命令编辑特征。可以像在bash或tcsh那样按Tab键让gdb补齐一个唯一的命令，如果相关项不唯一，gdb会列出所有匹配的命令，也可以用光标键上下翻动历史命令。
- 一条gdb命令是一个单行的输入，长度没有限制。命令的后面可以跟参数。
- gdb命令只要没有二义性就可以被缩写。
- 在gdb的提示符下直接按enter表示重复上一条命令。但run命令除外，它不能被重复。

调用gdb

- 执行gdb时可以指定许多命令行参数，通过这些参数可以在gdb开始时就设置好程序运行和调试的环境。
- 一般调用格式：
- $gdb <可执行文件名>
- 可以为要执行的文件指定一个进程号：
- $gdb <可执行文件名> <进程号>

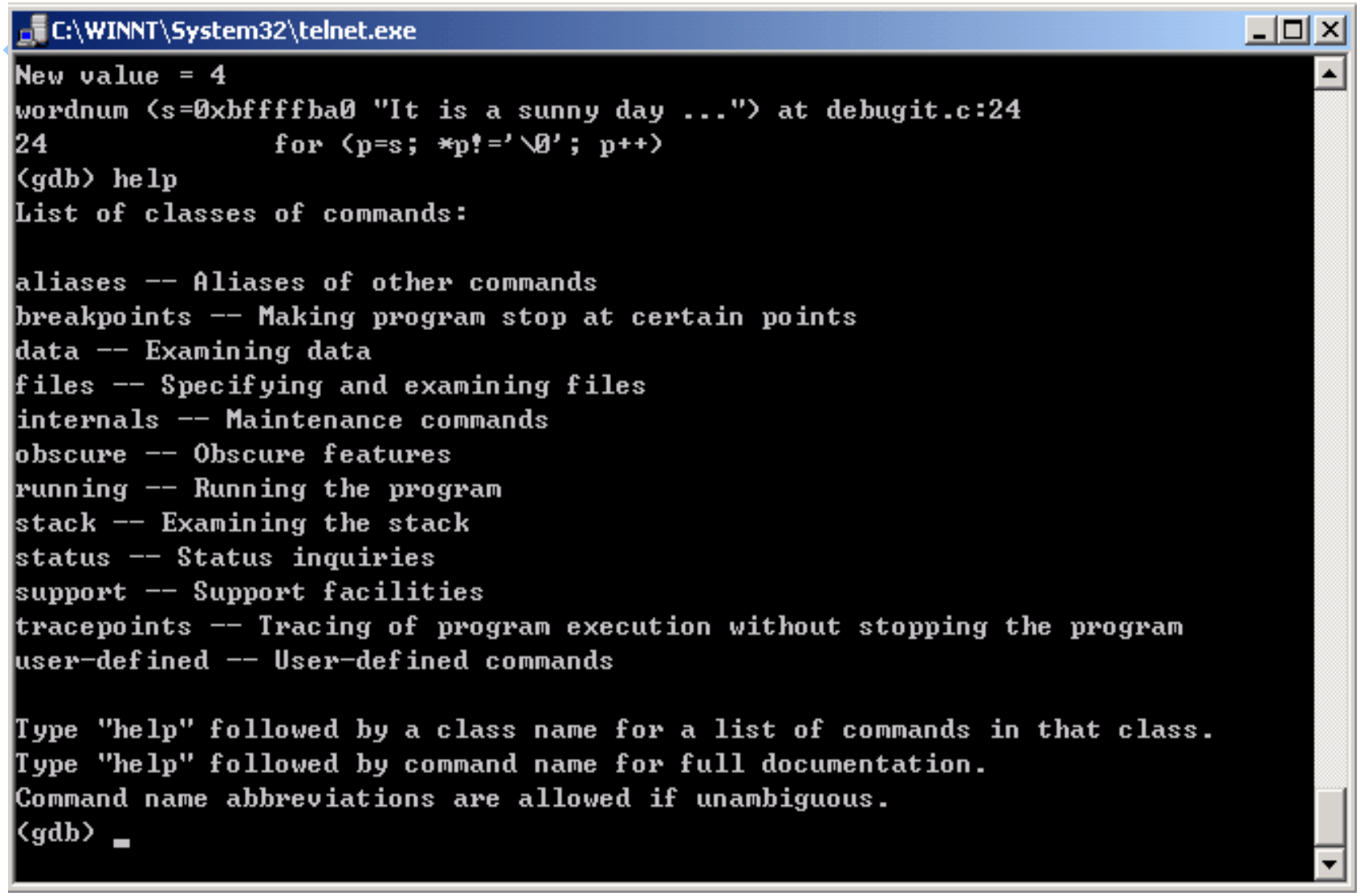
调用gdb


C:\WINNT\System32\telnet.exe
[zf@localhost cprogram]$ gdb ./debugit 2000
GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 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-redhat-linux"...
/home/zf/cprogram/2000: No such file or directory.
Attaching to program: /home/zf/cprogram/./debugit, process 2000
ptrace: No such process.
(gdb)

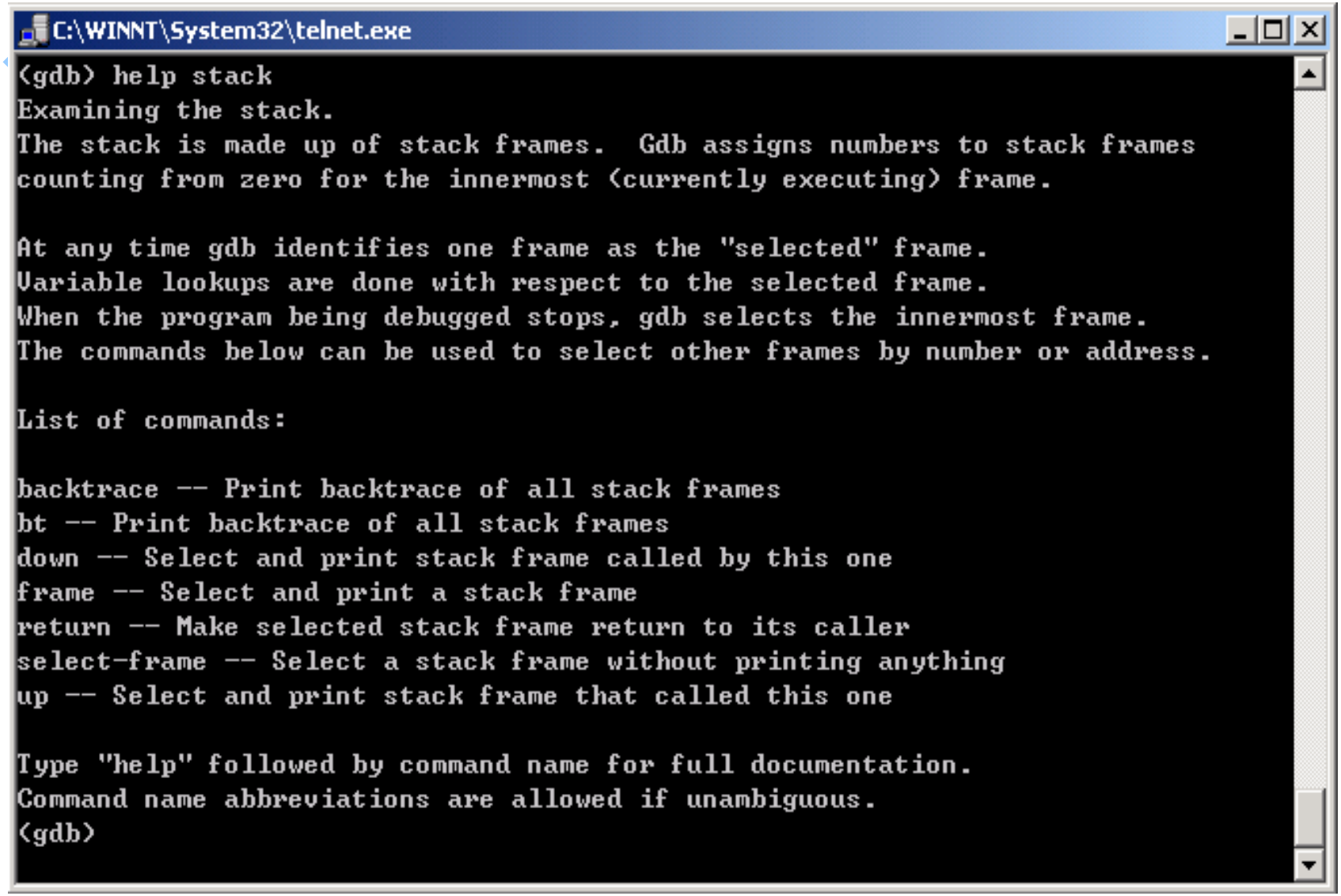
调用gdb

- 上例为debugit指定了进程号2000。
- 首先，gdb会寻找一个文件名为2000的文件，如果找不到，则把调试程序debugit的进程号（PID）设置为2000。
- 如果不希望看到gdb开始的提示信息，可以用gdb –silent执行调试工作，可以通过更多的选项，按自己的喜好定制gdb的行为。

在gdb中获得帮助


C:\WINNT\System32\telnet.exe
New value = 4
wordnum (s=0xbffffba0 "It is a sunny day ...") at debugit.c:24
24 for (p=s; *p!='\0'; p++)
(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)

在gdb中获得帮助


C:\WINNT\System32\telnet.exe
(gdb) help stack
Examining the stack.
The stack is made up of stack frames. Gdb assigns numbers to stack frames
counting from zero for the innermost (currently executing) frame.

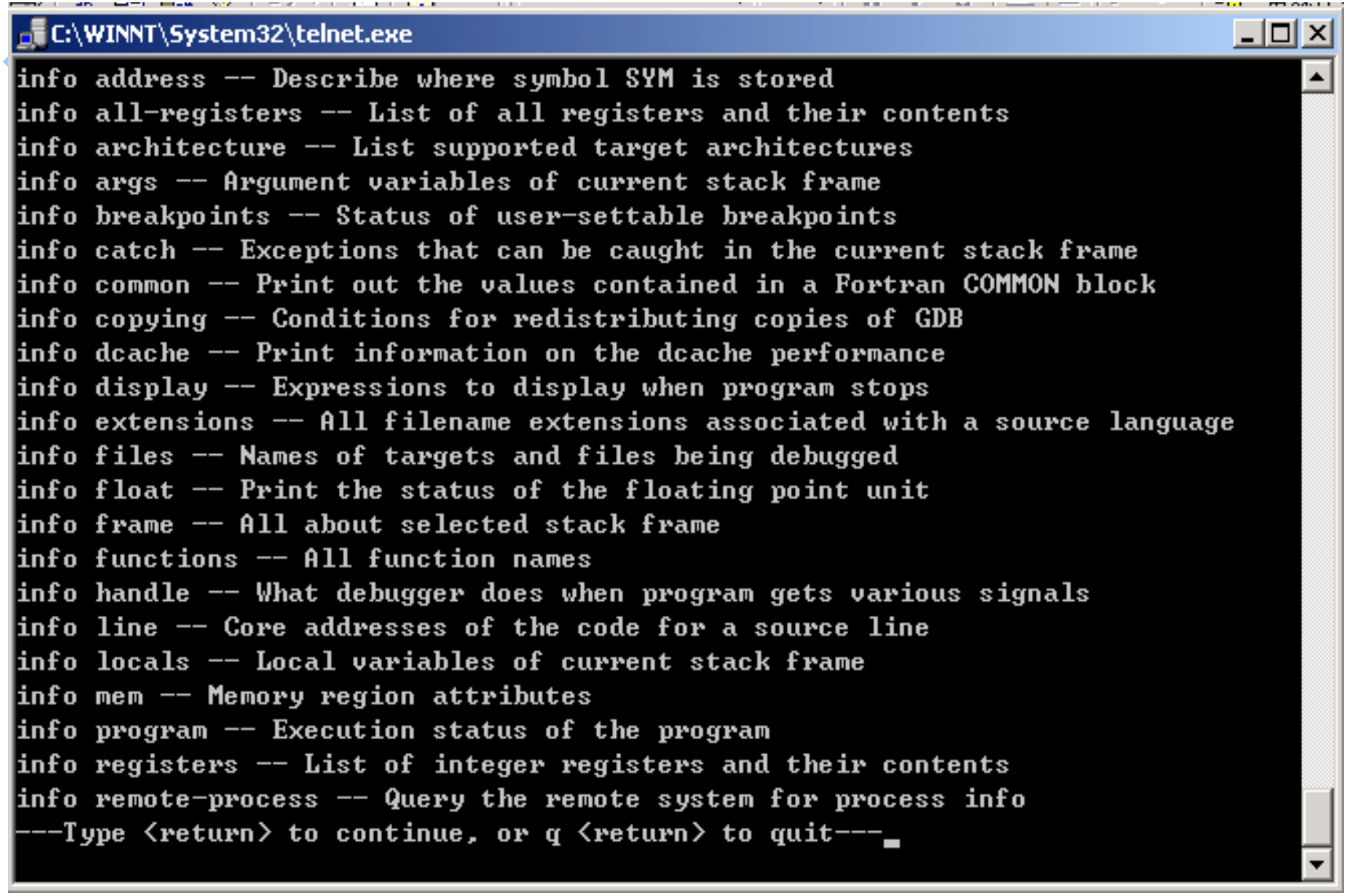
At any time gdb identifies one frame as the "selected" frame.
Variable lookups are done with respect to the selected frame.
When the program being debugged stops, gdb selects the innermost frame.
The commands below can be used to select other frames by number or address.

List of commands:

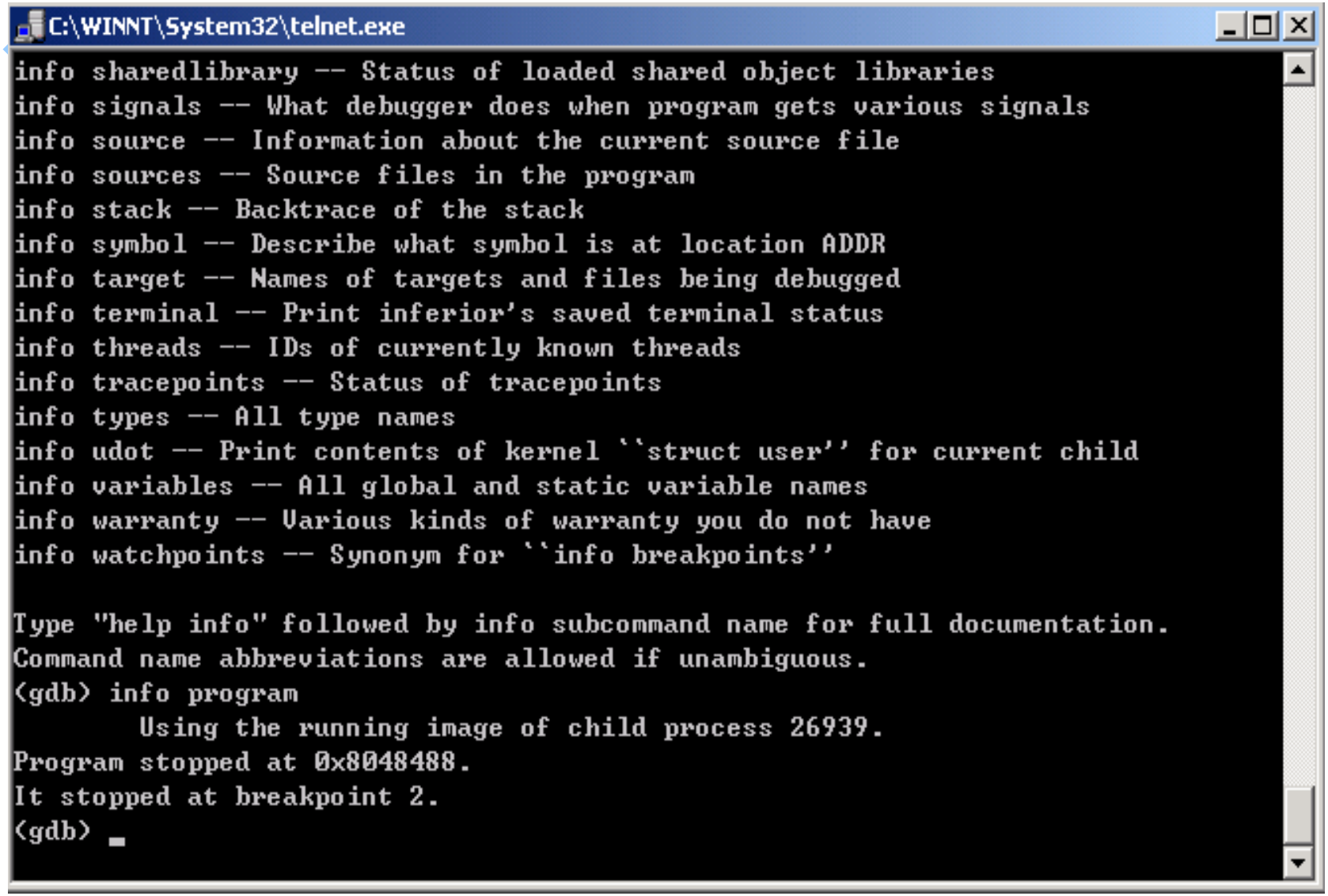
backtrace -- Print backtrace of all stack frames
bt -- Print backtrace of all stack frames
down -- Select and print stack frame called by this one
frame -- Select and print a stack frame
return -- Make selected stack frame return to its caller
select-frame -- Select a stack frame without printing anything
up -- Select and print stack frame that called this one

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

在gdb中获得帮助


C:\WINNT\System32\telnet.exe
info address -- Describe where symbol SYM is stored
info all-registers -- List of all registers and their contents
info architecture -- List supported target architectures
info args -- Argument variables of current stack frame
info breakpoints -- Status of user-settable breakpoints
info catch -- Exceptions that can be caught in the current stack frame
info common -- Print out the values contained in a Fortran COMMON block
info copying -- Conditions for redistributing copies of GDB
info dcache -- Print information on the dcache performance
info display -- Expressions to display when program stops
info extensions -- All filename extensions associated with a source language
info files -- Names of targets and files being debugged
info float -- Print the status of the floating point unit
info frame -- All about selected stack frame
info functions -- All function names
info handle -- What debugger does when program gets various signals
info line -- Core addresses of the code for a source line
info locals -- Local variables of current stack frame
info mem -- Memory region attributes
info program -- Execution status of the program
info registers -- List of integer registers and their contents
info remote-process -- Query the remote system for process info
---Type <return> to continue, or q <return> to quit---

在gdb中获得帮助


C:\WINNT\System32\telnet.exe
info sharedlibrary -- Status of loaded shared object libraries
info signals -- What debugger does when program gets various signals
info source -- Information about the current source file
info sources -- Source files in the program
info stack -- Backtrace of the stack
info symbol -- Describe what symbol is at location ADDR
info target -- Names of targets and files being debugged
info terminal -- Print inferior's saved terminal status
info threads -- IDs of currently known threads
info tracepoints -- Status of tracepoints
info types -- All type names
info udot -- Print contents of kernel ``struct user'' for current child
info variables -- All global and static variable names
info warranty -- Various kinds of warranty you do not have
info watchpoints -- Synonym for ``info breakpoints''
Type "help info" followed by info subcommand name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) info program
Using the running image of child process 26939.
Program stopped at 0x8048488.
It stopped at breakpoint 2.
(gdb)

Gdb的高级应用

- gdb变量的作用域的概念
- 函数堆栈操作
- 操作源文件
- 在gdb中与Shell进行通信

gdb变量的作用域的概念

- gdb把变量分为两类：处于活动状态的变量和处于非活动状态的变量。Gdb 只能访问、观察、操作活动变量。可以参考以下规则判断变量的状态：
- 程序的全局变量总是活动的，无论程序是否运行。
- 非全局变量是非活动的，除非此变量的定义域的程序段正在运行。例如，parent()调用函数child()，只要child()在运行，所有parent()和child()所定义的局部变量就是活动的。而child()返回，则只有parent()函数的局部变量是活动的。

gdb变量的作用域的概念

- 如果调试多个文件编译和链接组成的程序，gdb中采用如下的规则标识每个变量：
- domain_name::variable
- 其中，variable是想要引用的变量名，domain_name是含有variable变量的文件或函数。
- 例: (gdb) print ‘file1.c’::var
- 文件名必须包含在单引号中。
- 例： (gdb) print fun1::var
- (gdb) print fun2::var

函数堆栈操作

- gdb可以让开发者观察函数是如何调用堆栈的 。
- 例如，请看下面程序代码：

```
/*stack.c  program to illustrate the stack operation in gdb*/
#include <stdio.h>
#include <stdlib.h>

 void root(void);
 void subdir(void);
 void getfile(void);

 int main()
{
      printf("This is main!\n");
      root();
      return 0;
 }
```

函数堆栈操作

```
 void root(void)
{
        printf(“This is root!\n”);
        subdir();
}

 void subdir(void)
{
        printf(“This is subdir!\n”);
        getfile();
}

void getfile(void)
{
        printf(“This is getfile!\n”);
}
```

函数堆栈操作

- 编译链接stack.c程序并使用gdb进行调试：
- $gcc –g stack.c –o stack
- $gdb ./stack
- 在程序getfile()的第一行设置断点，并执行程序，当程序停止时，可以用where显示堆栈的情况。

函数堆栈操作

C:\WINNT\System32\telnet.exe

```
26
27      void getfile(void)
28      {
29              printf("This is getfile!\n");
30      }
(gdb) l
Line number 31 out of range; stack.c has 30 lines.
(gdb) b 9
Breakpoint 1 at 0x80483c4: file stack.c, line 9.
(gdb) r
Starting program: /home/zf/cprogram/./stack

Breakpoint 1, main () at stack.c:10
10      {
(gdb) n
main () at stack.c:11
11              printf("This is main!\n");
(gdb) n
This is main!
12              root();
(gdb)
This is root!
This is subdir!
This is getfile!
13              return 0;
(gdb)
```

函数堆栈操作

- 反映了函数调用链，getfile()函数被subdir()函数调用，subdir() 被root()调用， root() 被main()调用。
- 使用up或down命令可以显示当前堆栈的上一个或下一个堆栈。
- 使用堆栈检查命令和变量观察命令，可以使开发者从程序运行的底层开始跟踪函数的调用过程并可以随时检查不同调用上下文的局部变量的值，这将极大的方便调试程序的隐含错误。

函数堆栈操作

C:\WINNT\System32\telnet.exe

```
Breakpoint 1, subdir () at stack.c:24
24              getfile();
(gdb) where
#0  subdir () at stack.c:24
#1  0x080483f9 in root () at stack.c:18
#2  0x080483d9 in main () at stack.c:12
#3  0x40040177 in __libc_start_main (main=0x80483c4 <main>, argc=1,
    ubp_av=0xbffffc5c, init=0x80482a8 <_init>, fini=0x8048458 <_fini>,
    rtld_fini=0x4000e184 <_dl_fini>, stack_end=0xbffffc54)
    at ../sysdeps/generic/libc-start.c:129
(gdb) where full
#0  subdir () at stack.c:24
No locals.
#1  0x080483f9 in root () at stack.c:18
No locals.
#2  0x080483d9 in main () at stack.c:12
No locals.
#3  0x40040177 in __libc_start_main (main=0x80483c4 <main>, argc=1,
    ubp_av=0xbffffc5c, init=0x80482a8 <_init>, fini=0x8048458 <_fini>,
    rtld_fini=0x4000e184 <_dl_fini>, stack_end=0xbffffc54)
    at ../sysdeps/generic/libc-start.c:129
        ubp_av = (char **) 0xbffffc5c
        fini = (void (*)()) 0x40016b64 <_dl_debug_mask>
        rtld_fini = (void (*)()) 0x400
        ubp_ev = (char **) 0xbffffc64
(gdb) _
```

函数堆栈操作

C:\WINNT\System32\telnet.exe

```
#2  0x080483d9 in main () at stack.c:12
#3  0x40040177 in __libc_start_main (main=0x80483c4 <main>, argc=1,
    ubp_av=0xbffffc5c, init=0x80482a8 <_init>, fini=0x8048458 <_fini>,
    rtld_fini=0x4000e184 <_dl_fini>, stack_end=0xbffffc54)
    at ../sysdeps/generic/libc-start.c:129
(gdb) where full
#0  subdir () at stack.c:24
No locals.
#1  0x080483f9 in root () at stack.c:18
No locals.
#2  0x080483d9 in main () at stack.c:12
No locals.
#3  0x40040177 in __libc_start_main (main=0x80483c4 <main>, argc=1,
    ubp_av=0xbffffc5c, init=0x80482a8 <_init>, fini=0x8048458 <_fini>,
    rtld_fini=0x4000e184 <_dl_fini>, stack_end=0xbffffc54)
    at ../sysdeps/generic/libc-start.c:129
        ubp_av = (char **) 0xbffffc5c
        fini = (void (*)()) 0x40016b64 <_dl_debug_mask>
        rtld_fini = (void (*)()) 0x400
        ubp_ev = (char **) 0xbffffc64
(gdb) down
Bottom (i.e., innermost) frame selected; you cannot go down.
(gdb) up
#1  0x080483f9 in root () at stack.c:18
18              subdir();
(gdb) _
```

在gdb中访问多个文件

- 启动gdb时，可以用-d选项告诉gdb在哪里可以找到源代码。如果当前工作命令或程序的编译目录没有包含所有的源文件时，可以用这个选项开关指定一个或多个其他目录。
- 例如：
- $gdb -d /source1 –d /source2 progfile
- 在当前文件中定位一个特定字符串可以使用search命令，使用反向查找命令reverse-search来查找字符串上一次的出现。

在gdb中访问多个文件

C:\WINNT\System32\telnet.exe

```
#0  subdir () at stack.c:24
No locals.
#1  0x080483f9 in root () at stack.c:18
No locals.
#2  0x080483d9 in main () at stack.c:12
No locals.
#3  0x40040177 in __libc_start_main (main=0x80483c4 <main>, argc=1,
    ubp_av=0xbffffc5c, init=0x80482a8 <_init>, fini=0x8048458 <_fini>,
    rtld_fini=0x4000e184 <_dl_fini>, stack_end=0xbffffc54)
    at ../sysdeps/generic/libc-start.c:129
        ubp_av = (char **) 0xbffffc5c
        fini = (void (*)()) 0x40016b64 <_dl_debug_mask>
        rtld_fini = (void (*)()) 0x400
        ubp_ev = (char **) 0xbffffc64
(gdb) down
Bottom (i.e., innermost) frame selected; you cannot go down.
(gdb) up
#1  0x080483f9 in root () at stack.c:18
18              subdir();
(gdb) search printf
23              printf("This is subdir!\n");
(gdb) reverse-search printf
17              printf("This is root!\n");
(gdb)
11            printf("This is main!\n");
(gdb)
```

在gdb中执行Shell命令

- gdb的shell命令可以使开发者不需要离开gdb就可以执行shell命令。命令语法是：
- (gdb) shell　command

在gdb中执行Shell命令

C:\WINNT\System32\telnet.exe

```
(gdb) shell ls -l
total 52
-rwxrwxr-x    1 zf       zf          20710 Dec  2 17:17 debugit
-rw-rw-r--    1 zf       zf            487 Dec  2 16:31 debugit.c
-rwxrwxr-x    1 zf       zf          20236 Dec  3 09:35 stack
-rw-rw-r--    1 zf       zf            447 Dec  3 09:34 stack.c
(gdb)
```

xxgdb

- 你可以使用一些前端工具如XXGDB，DDD等。他们都有图形化界面，因此使用更方便，但它们仅是GDB的一层外壳。

Cproto为函数产生原形

- cproto读入C源程序文件并自动为每个函数产生原型声明。用cproto可以在写程序时节省大量定义函数原型的时间。
- 例如，程序有如下代码：

```
/*stack1.c program to illustrate the results of cproto*/
int main()
{
    printf("This is main!\n");
    root();
    return 0;
}
```

Cproto为函数产生原形

```
 void root(void)
{
        printf(“This is root!\n”);
        subdir();
}

 void subdir(void)
{
        printf(“This is subdir!\n”);
        getfile();
}

void getfile(void)
{
        printf(“This is getfile!\n”);
}
```

Cproto为函数产生原形

C:\WINNT\System32\telnet.exe

```
 void root(void)
{
      printf("This is root!\n");
      subdir();
}

 void subdir(void)
{
      printf("This is subdir!\n");
      getfile();
}

void getfile(void)
{
      printf("This is getfile!\n");
}
~
"stack1.c" 23L, 343C written
[zf@localhost cprogram]$ cproto stack1.c -o stackfunc.h
[zf@localhost cprogram]$ more stackfunc.h
/* stack1.c */
int main(void);
void root(void);
void subdir(void);
void getfile(void);
[zf@localhost cprogram]$ _
```

Cproto为函数产生原形

- cproto的另外一个作用是把旧C语言的函数声明转换为新的ANSI C语言的语法。
- 例：
- main(argc,argv)
- int argc; /*number of arguments*/
- char *argv[]; /*arguments*/
- 转换为：
- int main(
- int argc; /*number of arguments*/
- char *argv[]; /*arguments*/
-)

Thank You !

zhaofang@email.buptsse.cn