CHAPTER 5

Working with GNU Debugger

The debugger is one of the most important components of any development system. No programmer writes a code that works on the first attempt or that does not have any bugs in it. Most of the time you have to go through debugging processes of your programs time and again. Therefore no development system is complete without a debugger. The GNU compiler, Emacs editor and other utilities work very closely with GNU debugger, also known as `gdb`, which is the debugger of choice of all open source developers community as well as for many commercial products. Many commercial debuggers are also built on top of GNU debugger.

In addition to command line debugging features, you can find many GUI front ends to the GNU debugger. These front ends include the famous `xxgdb`.

There are many other debuggers available in the open source community. Some of these debuggers are also introduced at the end of this chapter. This chapter provides a comprehensive working knowledge of `gdb` and how to use it. The text is accompanied by many examples to elaborate concepts presented here. After going through this chapter you should be able to debug many types of programs.

5.1 Introduction to GDB

GNU debugger or more commonly known as GDB is the most commonly used debugger in open source as well as commercial development on UNIX type platforms. It can be used as a native as well as cross debugger. GNU debugger supports many object file formats. The most commonly used formats are as follows:

- ELF
- a.out
- S-record

It is part of almost all of the Linux distributions available these days. In its command line form, it is started using the gdb command. The command provides an interactive text based prompt for the user. The default prompt for the debugger is (gdb). You can use commands available on this command prompt. Most of the commands can be abbreviated as long as they are not confused with any other command. For example, the next command can be used as single character n. This abbreviation will be more clear when you go through examples in this book.

All examples in this book use the GNU debugger that came with RedHat 7.1. The chapter also includes information about how to get the latest version of the debugger and install it on your system.

5.2 Getting Started with GDB

The GNU debugger is started using the gdb command. Upon startup, it displays initial information about the platform. It stops at the (gdb) command prompt where GDB commands can be used. A typical GDB screen is as shown below:

```
[rrehman@laptop gdb]$ gdb
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".
(gdb)
```

General help about GDB commands can be displayed using the help command on (gdb) prompt as shown below:

```
(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)
```

You can stop `gdb` using `quit` command on `(gdb)` prompt. You can also stop `gdb` or the current running command by pressing the `Ctrl` and `C` keys simultaneously.

You need to open a file for debugging after starting `gdb` using `file` command. You can also start `gdb` by providing a file name on the command line. The following command starts `gdb` and uses `a.out` file for debugging purpose.

```
gdb a.out
```

Please note that `a.out` must be compiled using the `–g` command line switch with `gcc` compiler. If you don't use the command line switch `–g`, no debug information will be stored in the `a.out` file. The typical command line to compile a file with debug support is as follows:

```
gcc -g hello.c -o hello
```

The above command creates an output file `hello` that can be used to debug `hello.c` program using the following command:

```
gdb hello
```

If you have started `gdb` without any command line argument, you can load a file into `gdb` at any time using `file` command at the `(gdb)` prompt.

5.2.1 Most Commonly Used gdb Commands

The most commonly used GDB commands are listed in Table 5-1. These commands are used to start and stop `gdb` debugging, executing a program and displaying results. Note that you start `gdb` using the `gdb [filename]` command and then you get the `(gdb)` prompt where you can use these commands.

Note that there is a difference between the `next` and the `step` commands. The `next` command will move you to the next line in the current function. If the current line is a function

Table 5-1 Common gdb commands

Command	Description
`run`	Start a program execution inside `gdb` from the beginning
`quit`	Quit `gdb`
`print expr`	Print expression, where expression may be a variable name
`next`	Go to next line
`step`	Step into next line
`continue`	Continue from the current place until end of program reaches or you find a break point

call, it will not go into the function code and you will not see what happens inside the function code. This is equivalent to the *step over* action used in many debuggers. On the other hand if you use the `step` command, it will move to the next line but if the current line is a function call, it will go into the function code and will take you to the first line inside the function. This is also called a *step into* action in some debuggers.

5.2.2 A Sample Session with gdb

In this section you go through a sample `gdb` session. You will use the following C code in this session and will step through the code from top to bottom. This code is saved in a file `sum.c` and the output executable file is `sum`. The `sum.c` file is listed below.

```
#include <stdio.h>
main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = num1 + num2;

  printf("\nThe sum is : %d\n", total);
}
```

As you can see, the `sum.c` file takes two numbers as input. It then calculates the sum and displays it as standard output. The compilation is done using the following command.

```
gcc -g sum.c -o sum
```

Now let us debug the output file. Note that when starting `gdb`, you use the output file `sum` on the command line and `gdb` will automatically find out the source code file name (`sum.c`) from the information stored inside the executable file. An explanation of different `gdb` commands is provided at the end of this debug session. Note that the `next` command is abbreviated as `n` in this session.

```
[rr@conformix 5]$ gdb sum
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"...
(gdb) list
1  #include <stdio.h>
2  main ()
3  {
4    int num1, num2, total ;
5
6    printf("Enter first number : ");
7    scanf("%d", &num1);
8    printf("Enter second number : ");
9    scanf("%d", &num2);
10
(gdb) list
11    total = num1 + num2;
12
13    printf("\nThe sum is : %d\n", total);
14  }
(gdb) list 1,14
1  #include <stdio.h>
2  main ()
3  {
4    int num1, num2, total ;
5
6    printf("Enter first number : ");
7    scanf("%d", &num1);
8    printf("Enter second number : ");
9    scanf("%d", &num2);
10
11   total = num1 + num2;
12
13   printf("\nThe sum is : %d\n", total);
```

```
  }
(gdb) **list 5**
1  #include <stdio.h>
2  main ()
3  {
4    int num1, num2, total ;
5
6    printf("Enter first number : ");
7    scanf("%d", &num1);
8    printf("Enter second number : ");
9    scanf("%d", &num2);
10
(gdb) list 9
4    int num1, num2, total ;
5
6    printf("Enter first number : ");
7    scanf("%d", &num1);
8    printf("Enter second number : ");
9    scanf("%d", &num2);
10
11   total = num1 + num2;
12
13   printf("\nThe sum is : %d\n", total);
gdb) break sum.c:6
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) run
Starting program: /home/rr/5/sum

Breakpoint 1, main () at sum.c:6
6    printf("Enter first number : ");
(gdb) n
7    scanf("%d", &num1);
(gdb) n
Enter first number : 45
8    printf("Enter second number : ");
(gdb) n
9    scanf("%d", &num2);
(gdb) n
Enter second number : 56
11   total = num1 + num2;
(gdb) n
13   printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 101
14 }
(gdb) n
```

```
Program exited with code 022.
(gdb) quit
[rr@conformix 5]$
```

Here are some observations about this session.

- The `list` command lists lines in the source code file and moves the pointer for listing. So the first `list` command lists lines from 1 to 10. The next `list` command lists lines from 11 to 14 where 14 is the last line.
- You can use a number with the `list` command. The `list` command displays the line with the number in the middle. So the command `list 9` will list lines from 4 to 13 trying to keep line number 9 in the middle.
- To list a range of lines, you can specify that range as an argument to the `list` command using a comma. For example, `list 1,14` will list lines from line number 1 to line number 14.
- You can set a break point using the `break` command at a particular line number. When you use the `run` command, the execution will stop at the break point. It is important that you create at least one break point before using the `run` command. If you don't do so, the program will continue to run until it reaches the end and you may not be able to debug it. More on `break` points will be discussed later.
- After setting a break point, you can use the `run` command to start execution of the program. Note that simply loading a file into `gdb` does not start execution. When `gdb` comes across a break point, it will stop execution and you will get a `(gdb)` command prompt. At this point you can start tracing the program line by line or using some other method.
- The `next` command (or simply `n`) executes the current line and moves execution pointer to the next line. Note that the line number and the contents of line are displayed each time you use the `next` command.
- When the running program needs an input, `gdb` will stop and you can enter the required value. For example, I have entered 45 and 56 as values for `num1` and `num2`.
- `gdb` will print any output when a command is executed that prints something on the output device. You can see the result of addition printed.
- When `gdb` reaches the end of execution, it will print the exit code. After that you can use the `quit` command to end `gdb` session.

This was a simple `gdb` session. In next sections, you will go through more examples of how to use `gdb`.

5.2.3 Passing Command Line Arguments to the Program Being Debugged

Many program need command line arguments to be passed to them at the time of execution. There are multiple ways to pass these command line arguments when you debug a program. Let us look at the following program that prints two command line arguments. The program prints a message if exactly two command line arguments are not specified. Note that this may not be the best program for this purpose and is used only to demonstrate a debugging process. The program name is `arg.c` and it is listed below.

```
#include <stdio.h>
#include <string.h>
main (int argc, char **argv)
{
  if (argc != 3) {
    printf("You have to use two command line arguments\n");
    exit(-1);
  }
  printf("This program prints two command line arguments\n");
  printf("The first argument is : %s\n", argv[1]);
  printf("The second argument is : %s\n", argv[2]);
}
```

You have to create an executable file, `arg`, from `arg.c` source code file. If you run this program in `gdb` using the `run` command, it complains about command line arguments and terminates as expected. This is shown below.

```
[rr@conformix 5]$ gdb arg
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"...
(gdb) run
Starting program: /home/rr/5/arg
You have to use two command line arguments

Program exited with code 0377.
(gdb)
```

Note that no `break` point is set as we are not interested in stepping through the program line by line at this point.

The following lines of another `gdb` session output show that you can specify command line arguments with the `run` command. Any arguments for the `run` command are considered as program arguments. These arguments are “`test1`” and “`test2`” and are printed on the screen when you run the program.

```
[rr@conformix 5]$ gdb arg
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"...
(gdb) run test1 test2
Starting program: /home/rr/5/arg test1 test2
This program prints two command line arguments
The first argument is : test1
The second argument is : test2

Program exited with code 037.
(gdb)
```

Another method is to set the arguments using the `set` command as done in the following `gdb` session. You can display current arguments using the `show` command, which is also present in this `gdb` session. After setting the arguments, you can use the `run` command to start executing the program.

```
[rr@conformix 5]$ gdb arg
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"...
(gdb) set args test1 test2
(gdb) show args
Argument list to give program being debugged when it is
started is "test1 test2".
(gdb) run
Starting program: /home/rr/5/arg test1 test2
This program prints two command line arguments
The first argument is : test1
The second argument is : test2

Program exited with code 037.
(gdb)
```

5.3 Controlling Execution

Now you are already familiar with some of the gdb commands used for controlling execution of a program. The commands you have already used are run, quit and next commands. In this section you will learn some more commands that can be used to control execution of the program. The most commonly used commands to control execution of a program in gdb are as follows:

- The run command is used to start execution of a program. If the program is already running, you can restart the execution right from the beginning using the run command.
- The quit command will quit the debugger.
- The kill command stops debugging but does not quit the debugger.
- The continue command starts executing program from the current position. The difference between the continue and run commands is that the run commands starts execution from the entry point of the program while the continue command starts execution from the current location.
- The next command goes to the next line in the code. If the current line is a function call, it completes the function call without going into the function code.
- The step command goes to the next line in the code. If the current line is a function call, it goes to the first line inside that function.
- The finish command takes you out of the function call, if you are already inside one.
- The return command returns to the caller of the current frame in the stack. This means that you can return from a function without actually completing the function code execution.

There are many ways to control execution of a program. Mastery of these methods is necessary to debug a program more efficiently.

5.3.1 The step and finish Commands

Here we have a short demonstration of how to use finish and step commands when you are dealing with a function. Consider the following sumf.c program that adds two numbers and prints the result of addition. The addition is completed inside a function called sum().

```
#include <stdio.h>

int sum(int num1, int num2);

main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = sum(num1, num2);
  printf("\nThe sum is : %d\n", total);
}

int sum(int num1, int num2)
{
  int result;
  result = num1 + num2 ;
  printf("\nCalculation complete. Returning ...\n");
  return (result);
}
```

Let us create an executable `sumf` from this source code using the `gcc` compiler. This executable then can be debugged using GNU debugger as shown in the next session. Note that we use the `step` command (abbreviated as `s`) to step into the function code. We use the `finish` command to complete execution of code in the function `sum()` instead of tracing it line by line. After the `finish` command is executed, you go to the next line in the `main()` function just after the call to the `sum()` function.

```
[rr@conformix 5]$ gdb sumf
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"...
(gdb) bre main
Breakpoint 1 at 0x8048496: file sumf.c, line 9.
(gdb) run
Starting program: /home/rr/5/sumf

Breakpoint 1, main () at sumf.c:9
9  printf("Enter first number : ");
(gdb) n
10  scanf("%d", &num1);
(gdb) n
Enter first number : 4
11  printf("Enter second number : ");
(gdb) n
12  scanf("%d", &num2);
(gdb) n
```

```
Enter second number : 5
14  total = sum(num1, num2);
(gdb) s
sum (num1=4, num2=5) at sumf.c:21
21  result = num1 + num2 ;
(gdb) finish
Run till exit from #0  sum (num1=4, num2=5) at sumf.c:21

Calculation complete. Returning ...
0x080484ec in main () at sumf.c:14
14  total = sum(num1, num2);
Value returned is $1 = 9
(gdb) n
15  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 9
16}
(gdb) n

Program exited with code 020.
(gdb)
```

As you can see from the above example, all gdb commands can be abbreviated as long as they can't be confused with any other command. You can use n for next, bre for the break command and s for the step command. Also note that when you return from the function call, the return value of the function is displayed.

5.4 Working with the Stack

One of the most important tasks during the debugging process is to deal with the stack. The stack provides a lot of information about what happened in the past. You can backtrace to higher levels in the stack using gdb. Using the backtrace feature, you can move step-by-step backward to find out how you reached the current position. To demonstrate some of the commands used with stack, let us take the example of sumf.c program that you used in the last section. The program takes two numbers as input and then calls a function to sum() to add these two numbers. The function returns the result back to the main function.

When you start debugging this program, first you set up a break point right at the beginning of the main function and then use the next command repeatedly until you reach the line where the function is called. At this point you use the step command instead of the next command to move into the function code. After entering the function code, some other commands related to stack are used in the following gdb session. These commands are explained at the end of this session listing.

```
[rr@conformix 5]$ gdb sumf
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"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sumf.c, line 9.
(gdb) run
Starting program: /home/rr/5/sumf

Breakpoint 1, main () at sumf.c:9
9  printf("Enter first number : ");
(gdb) next
10  scanf("%d", &num1);
(gdb) next
Enter first number : 30
11  printf("Enter second number : ");
(gdb) next
12  scanf("%d", &num2);
(gdb) next
Enter second number : 40
14  total = sum(num1, num2);
(gdb) step
sum (num1=30, num2=40) at sumf.c:21
21  result = num1 + num2 ;
(gdb) next
22  printf("\nCalculation complete. Returning ...\n");
(gdb) frame
#0  sum (num1=30, num2=40) at sumf.c:22
22  printf("\nCalculation complete. Returning ...\n");
(gdb) info frame
Stack level 0, frame at 0xbffffa08:
 eip = 0x804851b in sum (sumf.c:22); saved eip 0x80484ec
 called by frame at 0xbffffa38
 source language c.
 Arglist at 0xbffffa08, args: num1=30, num2=40
 Locals at 0xbffffa08, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa08, eip at 0xbffffa0c
(gdb) info args
num1 = 30
num2 = 40
```

```
(gdb) info locals
result = 70
(gdb) info reg
eax            0x4670
ecx            0x00
edx            0x401548e01075136736
ebx            0x401569e41075145188
esp            0xbffffa000xbffffa00
ebp            0xbffffa080xbffffa08
esi            0x40016b641073834852
edi            0xbffffaac-1073743188
eip            0x804851b0x804851b
eflags         0x312786
cs             0x2335
ss             0x2b43
ds             0x2b43
es             0x2b43
fs             0x00
gs             0x00
fctrl          0x37f895
fstat          0x00
ftag           0xffff65535
fiseg          0x2335
fioff          0x404b42151078673941
foseg          0x2b43
fooff          0x404c69681078749544
fop            0x2e9745
(gdb) info all-reg
eax            0x4670
ecx            0x00
edx            0x401548e01075136736
ebx            0x401569e41075145188
esp            0xbffffa000xbffffa00
ebp            0xbffffa080xbffffa08
esi            0x40016b641073834852
edi            0xbffffaac-1073743188
eip            0x804851b0x804851b
eflags         0x312786
cs             0x2335
ss             0x2b43
ds             0x2b43
es             0x2b43
fs             0x00
gs             0x00
st0            0(raw 0x00000000000000000000)
st1            0(raw 0x00000000000000000000)
st2            0(raw 0x00000000000000000000)
st3            0(raw 0x00000000000000000000)
```

```
st4            0(raw 0x00000000000000000000)
st5            0(raw 0x00000000000000000000)
st6            1.147239263803680981573807073914395(raw
0x3fff92d8bc775ca99ea0)
st7            3(raw 0x4000c000000000000000)
fctrl          0x37f895
fstat          0x00
ftag           0xffff65535
fiseg          0x2335
fioff          0x404b42151078673941
foseg          0x2b43
fooff          0x404c69681078749544
fop            0x2e9745
(gdb) backtrace
#0  sum (num1=30, num2=40) at sumf.c:22
#1  0x080484ec in main () at sumf.c:14
#2  0x40048177 in __libc_start_main (main=0x8048490 <main>,
argc=1, ubp_av=0xbffffaac, init=0x8048308 <_init>,
    fini=0x8048580 <_fini>, rtld_fini=0x4000e184 <_dl_fini>,
stack_end=0xbffffa9c)
    at ../sysdeps/generic/libc-start.c:129
(gdb) info frame
Stack level 0, frame at 0xbffffa08:
 eip = 0x804851b in sum (sumf.c:22); saved eip 0x80484ec
 called by frame at 0xbffffa38
 source language c.
 Arglist at 0xbffffa08, args: num1=30, num2=40
 Locals at 0xbffffa08, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa08, eip at 0xbffffa0c
(gdb) up
#1  0x080484ec in main () at sumf.c:14
#2  0x40048177 in __libc_start_main (main=0x8048490 <main>,
argc=1, ubp_av=0xbffffaac, init=0x8048308 <_init>,
    fini=0x8048580 <_fini>, rtld_fini=0x4000e184 <_dl_fini>,
stack_end=0xbffffa9c)
    at ../sysdeps/generic/libc-start.c:129
(gdb) info frame
Stack level 0, frame at 0xbffffa08:
 eip = 0x804851b in sum (sumf.c:22); saved eip 0x80484ec
 called by frame at 0xbffffa38
 source language c.
 Arglist at 0xbffffa08, args: num1=30, num2=40
 Locals at 0xbffffa08, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa08, eip at 0xbffffa0c
(gdb) up
#1  0x080484ec in main () at sumf.c:14
```

```
14  total = sum(num1, num2);
(gdb) info frame
Stack level 1, frame at 0xbffffa38:
 eip = 0x80484ec in main (sumf.c:14); saved eip 0x40048177
 called by frame at 0xbffffa78, caller of frame at 0xbffffa08
 source language c.
 Arglist at 0xbffffa38, args:
 Locals at 0xbffffa38, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa38, eip at 0xbffffa3c
(gdb) down
#0  sum (num1=30, num2=40) at sumf.c:22
22  printf("\nCalculation complete. Returning ...\n");
(gdb) info frame
Stack level 0, frame at 0xbffffa08:
 eip = 0x804851b in sum (sumf.c:22); saved eip 0x80484ec
 called by frame at 0xbffffa38
 source language c.
 Arglist at 0xbffffa08, args: num1=30, num2=40
 Locals at 0xbffffa08, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa08, eip at 0xbffffa0c
(gdb) down
Bottom (i.e., innermost) frame selected; you cannot go down.
(gdb) n

Calculation complete. Returning ...
23  return (result);
(gdb) n
24}
(gdb) n
main () at sumf.c:15
15  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 70
16}
(gdb) n

Program exited with code 021.
(gdb) quit
[rr@conformix 5]$
```

- The `frame` command shows the current frame of execution for the program. In simple terms, you can consider a frame as a block of commands. For example, commands in one function are a frame. When you use the `frame` command, it displays the current frame starting point, the file name and current execution pointer.

- The `info frame` command shows more information about the current frame, including some register values. It also shows the stack pointer for the previous frame. These values are taken from the stack.
- The `info args` command displays arguments passed to this frame. This is also taken from the stack.
- The `info locals` command displays the values of local variables. These variable have a scope limited to the current frame.
- The `info reg` command displays values of register values.
- The `info all-reg` command displays register values, including math registers.
- The `up` command takes you one level up in the stack. This means if you are inside a function call, the up command will take you to the function that called the current function. The `down` command is opposite to the `up` command.
- You can use `backtrace`, `up` and `down` commands to move around in different frames. These commands are useful for looking into stack data.

A combination of all of these commands used with other execution control command can be used to display a lot of information. If you want to effectively use `GNU` debugger, knowledge of commands related to stack is a must.

5.5 Displaying Variables

Using the GNU debugger, you can display environment variables as well as your program variables during the program execution. You can control display of some variables so that the value of these variables is displayed with each command. Using this feature you can easily track changes taking place to these variables when you step through the program. You can also modify the program as well as environment variables. This section shows examples of how to carry out these tasks.

5.5.1 Displaying Program Variables

The following session uses the `sum.c` program that you already used earlier in this chapter. Go through the following `gdb` session and then see the discussion at the end of this session about actions taking place.

```
[rr@conformix 5]$ gdb sum
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"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) run
Starting program: /home/rr/5/sum

Breakpoint 1, main () at sum.c:6
6  printf("Enter first number : ");
(gdb) print num2
$1 = 134518424
(gdb) print total
$2 = 134513777
(gdb) n
7  scanf("%d", &num1);
(gdb) n
Enter first number : 45
8  printf("Enter second number : ");
(gdb) n
9  scanf("%d", &num2);
(gdb) n
Enter second number : 33
11  total = num1 + num2;
(gdb) print num2
$3 = 33
(gdb) n
13  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 78
14}
(gdb) n

Program exited with code 021.
(gdb)
```

- First set a `break` point where function `main` starts.
- Start the program execution using the `run` command.
- Print values of `num2` and `total` variables. Since these variables are not yet initialized, random numbers are present at these two locations.
- Execute the program line by line and enter values of `num1` and `num2` variables and then calculate the value of `total`.
- Print the value of `num2` variable which is now exactly what you have entered.

Note that in this session, `gdb` does not display the values of variables automatically when a value changes. You have to use the `print` command again and again to display variable values. In the next section, you will see how to display these values automatically.

5.5.2 Automatic Displaying Variables with Each Command

The `display` command displays the value of a variable and keeps a record of the variable so that its value is displayed after each command. The following session shows how this feature works. This is an easy way to closely watch the value of variables with the execution of each line of code. It is especially useful when you are tracing through a loop where you need to know the values of some variables in each cycle.

```
[rr@conformix 5]$ gdb sum
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"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) run
Starting program: /home/rr/5/sum

Breakpoint 1, main () at sum.c:6
6  printf("Enter first number : ");
(gdb) display num1
1: num1 = 134518424
(gdb) disp total
2: total = 134513777
(gdb) n
7  scanf("%d", &num1);
2: total = 134513777
1: num1 = 134518424
(gdb) n
Enter first number : 3
8  printf("Enter second number : ");
2: total = 134513777
1: num1 = 3
(gdb) n
9  scanf("%d", &num2);
2: total = 134513777
1: num1 = 3
(gdb) n
Enter second number : 67
11  total = num1 + num2;
2: total = 134513777
1: num1 = 3
```

```
(gdb) n
13  printf("\nThe sum is : %d\n", total);
2: total = 70
1: num1 = 3
(gdb) n

The sum is : 70
14}
2: total = 70
1: num1 = 3
(gdb) n

Program exited with code 021.
(gdb)
```

You can use the following commands to control the display of variables and get information about variables that are being displayed.

- The `undisplay` command removes a variable from the list of displayed items.
- The `disable display` command keeps the variable in the display list but disables its continuous display.
- The `enable display` command enables display of a variable that was previously disabled.
- The `info display` command displays a list of expressions or variables currently in the display list.

5.5.3 Displaying Environment Variables

Environment variables play a significant role in debugging a program. They are used to locate files and other tasks related to the file system as well as shell command execution. The following command displays the value of the `PATH` variable:

```
(gdb) show path
Executable and object file path: /usr/kerberos/bin:/bin:/usr/
bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin
```

The following command shows the current directory:

```
(gdb) pwd
Working directory /home/rr/5.
```

All environment variables can be displayed using the following command:

```
(gdb) show environment
PWD=/home/rr/5
HOSTNAME=conformix.conformix.net
PVM_RSH=/usr/bin/rsh
QTDIR=/usr/lib/qt-2.3.0
LESSOPEN=|/usr/bin/lesspipe.sh %s
```

```
XPVM_ROOT=/usr/share/pvm3/xpvm
KDEDIR=/usr
USER=rr
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=4
0;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=01;32:*.c
md=01;32:*.exe=01;32:*.com=01;32:*.btm=01;32:*.bat=01;32:*.sh=
01;32:*.csh=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01
;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.b
z2=01;31:*.bz=01;31:*.tz=01;31:*.rpm=01;31:*.cpio=01;31:*.jpg=
01;35:*.gif=01;35:*.bmp=01;35:*.xbm=01;35:*.xpm=01;35:*.png=01
;35:*.tif=01;35:
MACHTYPE=i386-redhat-linux-gnu
OLDPWD=/home/rr
MAIL=/var/spool/mail/rr
INPUTRC=/etc/inputrc
BASH_ENV=/home/rr/.bashrc
LANG=en_US
DISPLAY=192.168.2.115:1
LOGNAME=rr
SHLVL=1
SHELL=/bin/bash
HOSTTYPE=i386
OSTYPE=linux-gnu
HISTSIZE=1000
LAMHELPFILE=/etc/lam/lam-helpfile
PVM_ROOT=/usr/share/pvm3
TERM=xterm
HOME=/home/rr
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
PATH=/usr/kerberos/bin:/bin:/usr/bin:/usr/local/bin:/usr/bin/
X11:/usr/X11R6/bin
_=/usr/bin/gdb
(gdb)
```

You can also use the `set env` command and the `unset env` command to modify environment variables or to create or delete variables.

5.5.4 Modifying Variables

Using the `set` command, you can modify any program variable during the debugging process. The following session shows an example where you entered a value 34 for program variable `num1` but modified it to 11 using `set` command before the sum was calculated. You can see that the sum calculation process takes into account the new value.

```
Enter first number : 34
11  printf("Enter second number : ");
(gdb) n
12  scanf("%d", &num2);
```

```
(gdb) n
Enter second number : 12
14  total = sum(num1, num2);
(gdb) print num1
$1 = 34
(gdb) set num1=11
(gdb) print num1
$2 = 11
(gdb) print num2
$3 = 12
(gdb) n

Calculation complete. Returning ...
15  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 23
16 }
(gdb)
```

5.6 Adding Break Points

When you start debugging a program, you use the `run` command. This command executes the program until the end of the program or a break point is met. A break point is a place in your source code file where you temporarily want to stop execution of the program being debugged.

Break points in GNU debugger can be placed using the `break` command. Look at the following list of the source code file `sum.c` which you already have used:

```
(gdb) list
1  #include <stdio.h>
2  main ()
3  {
4    int num1, num2, total ;
5
6    printf("Enter first number : ");
7    scanf("%d", &num1);
8    printf("Enter second number : ");
9    scanf("%d", &num2);
10
(gdb)
```

To place a break point at line number 6 in file `sum.c` (displayed above), you can use the following command:

```
(gdb) break sum.c:6
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb)
```

As you can see from the above lines, when you set a break point, GNU debugger will display its information in the next line. This information contains the number of the breakpoint, memory address, file name and line number. You can also see a list of currently set break points using the following command:

```
(gdb) **info break**
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
(gdb)
```

Break points can also be set on function names. The following command sets a break point where function `main` starts:

```
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb)
```

Note that although the function `main()` starts at line number 2, the break point is set at line number 6. This is because the first executable instruction of the function `main` is located at this line number.

You can also set a break point at a particular line number in the currently loaded file. The following command creates a break point at line number 8:

```
(gdb) break 8
Breakpoint 2 at 0x80484ba: file sum.c, line 8.
(gdb)
```

In a multi-source file project, you set up a break point by including the file name and line number on the command line. The following command sets up a break point at line number 9 in file `sum.c`.

```
(gdb) break sum.c:9
Breakpoint 3 at 0x80484ca: file sum.c, line 9.
(gdb)
```

You can also use an offset value to set up a break point. For example if the execution pointer is on line number 6, you can set up a break point at line number 9 using the following command. Note that you can also use a minus symbol to specify an offset.

```
6  printf("Enter first number : ");
(gdb) break +3
Note: breakpoint 3 also set at pc 0x80484ca.
Breakpoint 4 at 0x80484ca: file sum.c, line 9.
(gdb)
```

All break points can be displayed using the `info` command. The following command displays three break points that we have specified:

```
(gdb) info break
Num Type           Disp Enb Address    What
```

```
1   breakpoint     keep y   0x08048496 in main at sum.c:6
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep y   0x080484ca in main at sum.c:9
(gdb)
```

If you want to disable all break points, use the `disable` command without any argument. Similarly if you want to enable all disabled break points, use the `enable` command without any argument. Normal arguments allow you to specify which break point to enable and disable.

5.6.1 Continuing from Break Point

You can continue execution of your program when you reach a break point in many ways. You can start tracing the program line by line using `next` or `step` commands. You can also use the `continue` command that will run the program from its current location until the program reaches its end or you find another break point on the way. You have already seen how to use the `step` and `next` commands. The following session creates a break point in the start of the `main` function and then uses the `continue` command to start execution from there.

```
[rr@conformix 5]$ gdb sum
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"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) run
Starting program: /home/rr/5/sum

Breakpoint 1, main () at sum.c:6
6  printf("Enter first number : ");
(gdb) continue
Continuing.
Enter first number : 34
Enter second number : 45

The sum is : 79

Program exited with code 021.
(gdb) quit
[rr@conformix 5]$
```

5.6.2 Disabling Break Points

Break points can be disabled temporarily. You can disable a break point using the `disable` command with a number as its argument. These numbers can be displayed using the `info break` command. The following command lists currently available break points. As you can see under the `Enb` column heading, all lines contain a `y` which shows that all break points are currently enabled.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
breakpoint already hit 1 time
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep y   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
```

To disable break point number 3 at line number 9 in file `sum.c` and then display the status of break points again, use the following two commands:

```
(gdb) dis 3
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
breakpoint already hit 1 time
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep n   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
(gdb)
```

Note that you can disable all break points in a single step if you don’t mention any number as an argument to the `disable` command.

5.6.3 Enabling Break Points

You can enable previously disabled break points using the `enable` command. The following command shows that breakpoint at line number 9 is currently disabled because it shows an `n` under the `Enb` column.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
    breakpoint already hit 1 time
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep n   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
```

The following two commands enable the break point number 3 and display the status of all break points.

```
(gdb) enab 3
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
breakpoint already hit 1 time
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep y   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
(gdb)
```

Enabling and disabling break points is useful when you want to cycle quickly through loops for a certain number of times and then start tracing once again.

5.6.4 Deleting Break Points

You can also delete break points using the `delete` command. The following session shows that you have four break points present. You delete two of these and then you can again display break points to make sure that these break points are actually deleted.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048496 in main at sum.c:6
    breakpoint already hit 1 time
2   breakpoint     keep y   0x080484ba in main at sum.c:8
3   breakpoint     keep y   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
(gdb) del 1
(gdb) del 2
(gdb) info break
Num Type           Disp Enb Address    What
3   breakpoint     keep y   0x080484ca in main at sum.c:9
6   breakpoint     keep y   0x08048496 in main at sum.c:5
(gdb)
```

Note that there is a difference between deleting and disabling a break point. The disabled break point stays there although it has no impact on execution of the program being debugged. The deleted break point is gone forever and you have to create it again if needed.

5.7 Debugging Optimized Code

You can use multiple levels of optimization while building the output binaries. The generated executable code may be different for each level of optimization. If you step through the optimized code in `gdb`, the `gdb` may not step as you expected in some cases. Let us compile the `sumopt.c` program and see how optimization does affect. Listing below is the program source code:

```
#include <stdio.h>
main ()
{
  int num1, num2, total ;

  num1 = 4;
  num2 = 6;
  total = num1 + num2;

  printf("\nThe sum is : %d\n", total);
}
```

Now let us compile the code without optimization using the following command and then step through it.

```
gcc -g sumopt.c -o sumopt
```

Following is the `gdb` session to step through the code. As you can see, it is quite straightforward. `gdb` executes all lines in order until it reaches end of the program.

```
[rr@conformix 5]$ gdb sumopt
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"...
(gdb) break main
Breakpoint 1 at 0x8048466: file sumopt.c, line 6.
(gdb) run
Starting program: /home/rr/5/sumopt

Breakpoint 1, main () at sumopt.c:6
6  num1 = 4;
(gdb) n
7  num2 = 6;
(gdb) n
8  total = num1 + num2;
(gdb) n
10  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 10
11}
(gdb) n

Program exited with code 021.
(gdb)
```

CH05.fm Page 162 Monday, October 7, 2002 9:00 PM

Now let us compile the program using the –O2 option for optimization. The command line for this is shown below:

```
gcc –O2 -g sumopt.c -o sumopt
```

Now let us trace through the optimized program. The gdb session for this purpose is listed below:

```
[rr@conformix 5]$ gdb sumopt
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"...
(gdb) break main
Breakpoint 1 at 0x8048466: file sumopt.c, line 10.
(gdb) run
Starting program: /home/rr/5/sumopt

Breakpoint 1, main () at sumopt.c:10
10  printf("\nThe sum is : %d\n", total);
(gdb) n

The sum is : 10
11 }
(gdb) n

Program exited with code 021.
(gdb)
```

This is quite different from what you may have expected. The first difference is in setting up the break point. We used the same command to set up the break point but now gdb set it up at line number 10 instead of line number 6 as it had done before. When we started execution of the program, it jumped to line number 10 and did not go to any line before that. This is because when we optimized code at the compile time, the compiler was smart enough to know that all values are static so there is no need to assign two static numbers to variables at the run time and then calculate their sum. All this can be done at the compile time, which results in shorter code and better execution time.

The bottom line is that if you find that gdb is not executing some lines when you debug a program, it may be due to the optimization level used when compiling a program. A fun trick might be to look at the variable values during such a debug session as well, but it's not necessary!

5.8 Files and Shared Libraries

You can find out information about the currently loaded file, its type, ELF segments, memory addresses and other things. The following command displays such information about the `sum` program.

```
(gdb) info files
Symbols from "/home/rr/5/sum".
Unix child process:
  Using the running image of child process 1860.
  While running this, GDB does not access memory from...
Local exec file:
  `/home/rr/5/sum', file type elf32-i386.
  Entry point: 0x8048390
  0x080480f4 - 0x08048107 is .interp
  0x08048108 - 0x08048128 is .note.ABI-tag
  0x08048128 - 0x08048160 is .hash
  0x08048160 - 0x080481f0 is .dynsym
  0x080481f0 - 0x0804828b is .dynstr
  0x0804828c - 0x0804829e is .gnu.version
  0x080482a0 - 0x080482d0 is .gnu.version_r
  0x080482d0 - 0x080482d8 is .rel.got
  0x080482d8 - 0x08048308 is .rel.plt
  0x08048308 - 0x08048320 is .init
  0x08048320 - 0x08048390 is .plt
  0x08048390 - 0x08048540 is .text
  0x08048540 - 0x0804855e is .fini
  0x08048560 - 0x080485aa is .rodata
  0x080495ac - 0x080495bc is .data
  0x080495bc - 0x080495c0 is .eh_frame
  0x080495c0 - 0x080495c8 is .ctors
  0x080495c8 - 0x080495d0 is .dtors
  0x080495d0 - 0x080495f8 is .got
  0x080495f8 - 0x08049698 is .dynamic
  0x08049698 - 0x080496b0 is .bss
```

You can also display information about shared libraries used by the `sum` program using the following command:

```
(gdb) info share
From        To          Syms Read   Shared Object Library
0x40047fa0  0x40140a9b  Yes         /lib/i686/libc.so.6
0x40001db0  0x4001321c  Yes         /lib/ld-linux.so.2
(gdb)
```

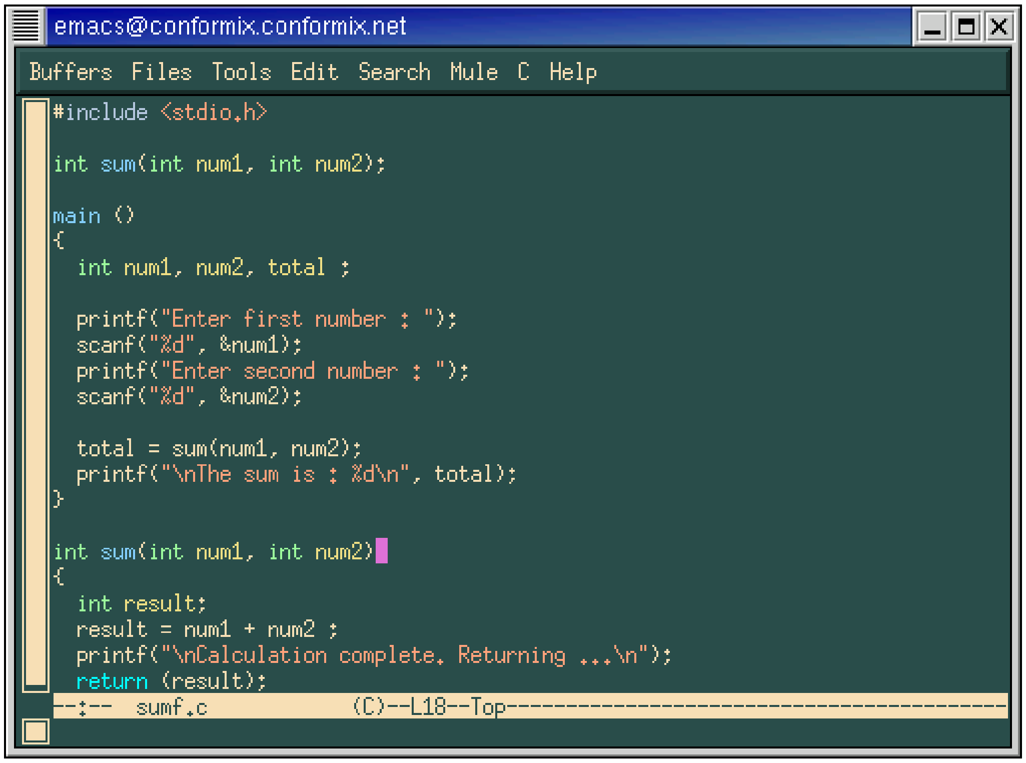
The command shows that the program is using `libc.so.6` and `ld-linux.so.2` during the debug session. The `ld-linux.so` is the dynamic loader, and will always be present in dynamically linked files.

CH05.fm Page 164 Monday, October 7, 2002 9:00 PM

5.9 Using gdb With GNU Emacs

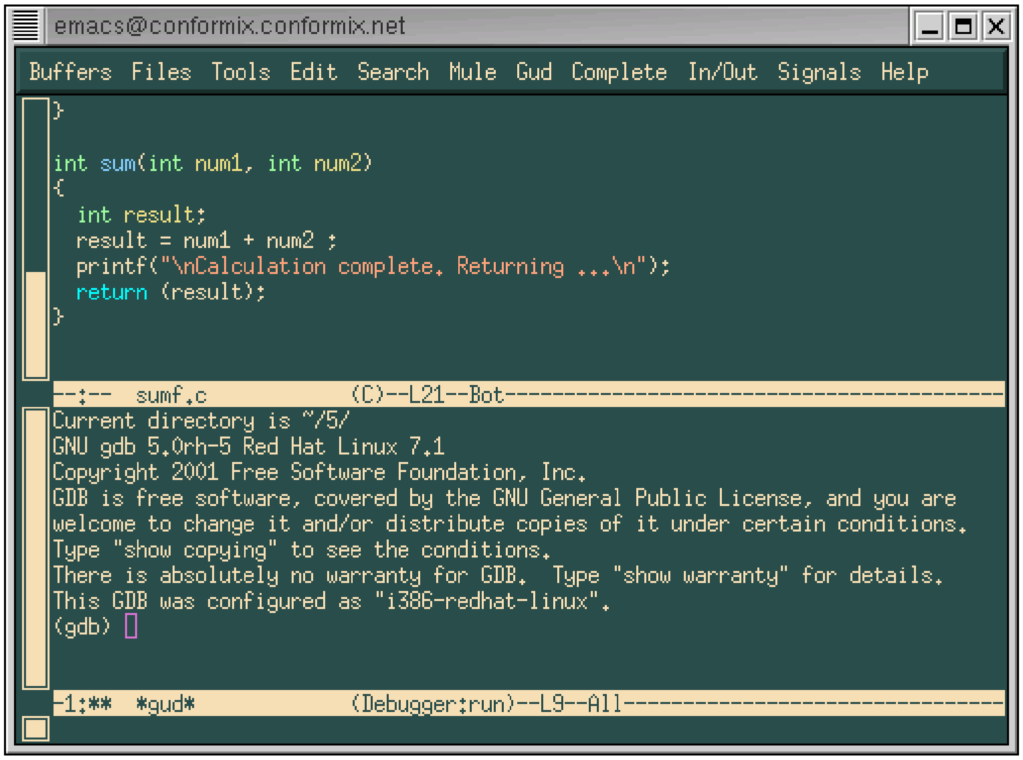
If you use GNU Emacs to edit your programs, you can also create a debug window within Emacs using the `M-x gdb` command. Let us suppose that you are editing `sumf.c` program in GNU Emacs as shown in Figure 5-1.

Now when you use `M-x gdb` command, you will see that the Emacs window is split and the bottom part is used as the `gdb` window. The screen shot in Figure 5-2 shows this new Emacs window. You can also see that new menus appear in the menu bar at the top of the screen. You can use these menus for the usual debugging functions like creating break points, starting and stopping the debug process and so on. The Gud menu is especially of interest for program debugging.


emacs@conformix.conformix.net
Buffers Files Tools Edit Search Mule C Help

```
#include <stdio.h>

int sum(int num1, int num2);

main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = sum(num1, num2);
  printf("\nThe sum is : %d\n", total);
}

int sum(int num1, int num2)
{
  int result;
  result = num1 + num2 ;
  printf("\nCalculation complete. Returning ...\n");
  return (result);
--:--  sumf.c          (C)--L18--Top----------------------------------------
```

Figure 5-1 Editing `sumf.c` program in GNU Emacs.


emacs@conformix.conformix.net
Buffers Files Tools Edit Search Mule Gud Complete In/Out Signals Help

```
}

int sum(int num1, int num2)
{
  int result;
  result = num1 + num2 ;
  printf("\nCalculation complete. Returning ...\n");
  return (result);
}

--:--  sumf.c            (C)--L21--Bot----------------------------------------
Current directory is ~/5/
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".
(gdb) 

-1:**  *gud*             (Debugger:run)--L9--All-------------------------------
```

Figure 5-2 The `gdb` split window inside GNU Emacs.

5.10 Debugging Running Processes

In addition to stand-alone programs, running processes can also be debugged using GNU debugger. The procedure is as follows:

1. Start the process that you want to debug.
2. Use the `ps` command to find the process ID of the running process.
3. Start `gdb`.
4. Load the file containing the symbol table into `gdb`.
5. Use the `attach` command to attach to the running process to `gdb`.
6. Continue debugging as usual.

In this section we shall attach to a process created by executing the `sumf` program that we created earlier in this chapter. This program requires input of two numbers. After starting, the process waits for the user to input these two numbers. You can open a new terminal window at this point and start gdb in the new window to attach the process. The following command starts the process which is waiting for the user to enter the first number.

```
[rr@conformix 5]$ ./sumf
Enter first number :
```

Open a new terminal window and find out the process ID of the sumf process. This can be easily done using the ps command as follows:

```
[rr@conformix 5]$ ps -a|grep sumf
4272 pts/4    00:00:00 sumf
[rr@conformix 5]$
```

The process ID for the running process is 4272. After getting this information, you can start gdb and load the sumf file that contains the symbol table. This is done using the file command. The following sequence of commands shows this process:

```
[rr@conformix 5]$ gdb
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".
(gdb) file sumf
Reading symbols from sumf...done.
(gdb)
```

Now you can attach to the program with ID 4272. Note that the program is waiting for user input and there are many functions calls from loaded shared libraries. To come back to this point in the main source file, you have to make a number of finish commands.

```
 (gdb) attach 4272
Attaching to program: /home/rr/5/sumf, process 4272
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0x40105f44 in __libc_read () from /lib/i686/libc.so.6
(gdb) n
Single stepping until exit from function __libc_read,
which has no line number information.
_IO_file_read (fp=0x401548e0, buf=0x40019000, size=1024) at
fileops.c:764
764  fileops.c: No such file or directory.
  in fileops.c
```

```
(gdb) finish
Run till exit from #0  _IO_file_read (fp=0x401548e0,
buf=0x40019000, size=1024) at fileops.c:764
0x400a68dd in _IO_new_file_underflow (fp=0x401548e0) at
fileops.c:467
467  in fileops.c
Value returned is $1 = 2
(gdb) finish
Run till exit from #0  0x400a68dd in _IO_new_file_underflow
(fp=0x401548e0) at fileops.c:467
_IO_default_uflow (fp=0x401548e0) at genops.c:389
389  genops.c: No such file or directory.
  in genops.c
Value returned is $2 = 52
(gdb) finish
Run till exit from #0  _IO_default_uflow (fp=0x401548e0) at
genops.c:389
0x400a7f52 in __uflow (fp=0x401548e0) at genops.c:345
345  in genops.c
Value returned is $3 = 52
(gdb) finish
Run till exit from #0  0x400a7f52 in __uflow (fp=0x401548e0)
at genops.c:345
0x4008bee5 in _IO_vfscanf (s=0x401548e0, format=0x80485d6
"%d", argptr=0xbffff9f4, errp=0x0)
    at vfscanf.c:610
610  vfscanf.c: No such file or directory.
  in vfscanf.c
Value returned is $4 = 52
(gdb) finish
Run till exit from #0  0x4008bee5 in _IO_vfscanf
(s=0x401548e0, format=0x80485d6 "%d",
    argptr=0xbffff9f4, errp=0x0) at vfscanf.c:610
scanf (format=0x80485d6 "%d") at scanf.c:40
40  scanf.c: No such file or directory.
  in scanf.c
Value returned is $5 = 1
(gdb) finish
Run till exit from #0  scanf (format=0x80485d6 "%d") at
scanf.c:40
0x080484b7 in main () at sumf.c:10
10    scanf("%d", &num1);
Value returned is $6 = 1
(gdb) n
11    printf("Enter second number : ");
(gdb) n
12    scanf("%d", &num2);
(gdb)
```

With every `finish` command, you can see the function that you were into and the return value of that function. When you are back in the `main` function, you can use the `next` command to start tracing the execution line by line. Note that when you trace through, the output of the process will appear in the window where you started the program. You will also enter the input values in the same window. All those functions we had to finish were just part of the standard C libraries, and you could actually step into them anytime from this program as well, but you normally wouldn't.

5.11 Installing GDB

GNU debugger is included in all Linux distributions in the development package. In certain cases, you may want to get a new copy in the source code form and build and install it manually. The compilation and building process is similar to other GNU development tools. By this time you have already built and installed the GNU compiler and the C library. With that experience, you should not find the compilation and installation of GNU debugger difficult.

5.11.1 Downloading and Building

You can download the latest version from ftp://ftp.gnu.org/gnu/gdb directory. The current version at the time of writing this book is 5.1.1. The source code is available in the compressed `tar` format. The file name is `gdb-5.1.1.tar.gz`. You have to uncompress it using the following command.

```
tar zxvf gdb-5.1.1.tar.gz
```

I have untarred this file in `/opt` directory. The `tar` program will create `/opt/gdb-5.1.1` directory and uncompress all source code tree under this directory. Move into this directory using the `cd` command. Run the `configure` script and the `make` command to build and install the debugger. This is a similar process as with all other GNU tools. The following commands will build `gdb`.

```
cd /opt/gdb-5.1.1
./configure --prefix=/opt/gcc-3.0.4
make
```

Note that `--prefix` shows the directory where the `gdb` binary and other files will be installed. Usually this is the same directory where your other development tools are installed.

5.11.2 Final Installation

The following command places the `gdb` binary and other files in the proper location. After running this command, you can start using the new debugger.

```
make install
```

5.12 Other Open Source Debuggers

In addition to gdb, there are many other debuggers available to the open source community. Many of these debuggers are built on top of the GNU debugger or they use gdb concepts. Some of these debuggers are introduced in this section. The most popular of these is the ddd debugger. xxgdb is a GUI interface to gdb and many people also use it. The kdbg comes with KDE on Linux. All of these debuggers use the same concepts as used by the gdb debugger.

5.12.1 The kdbg Debugger

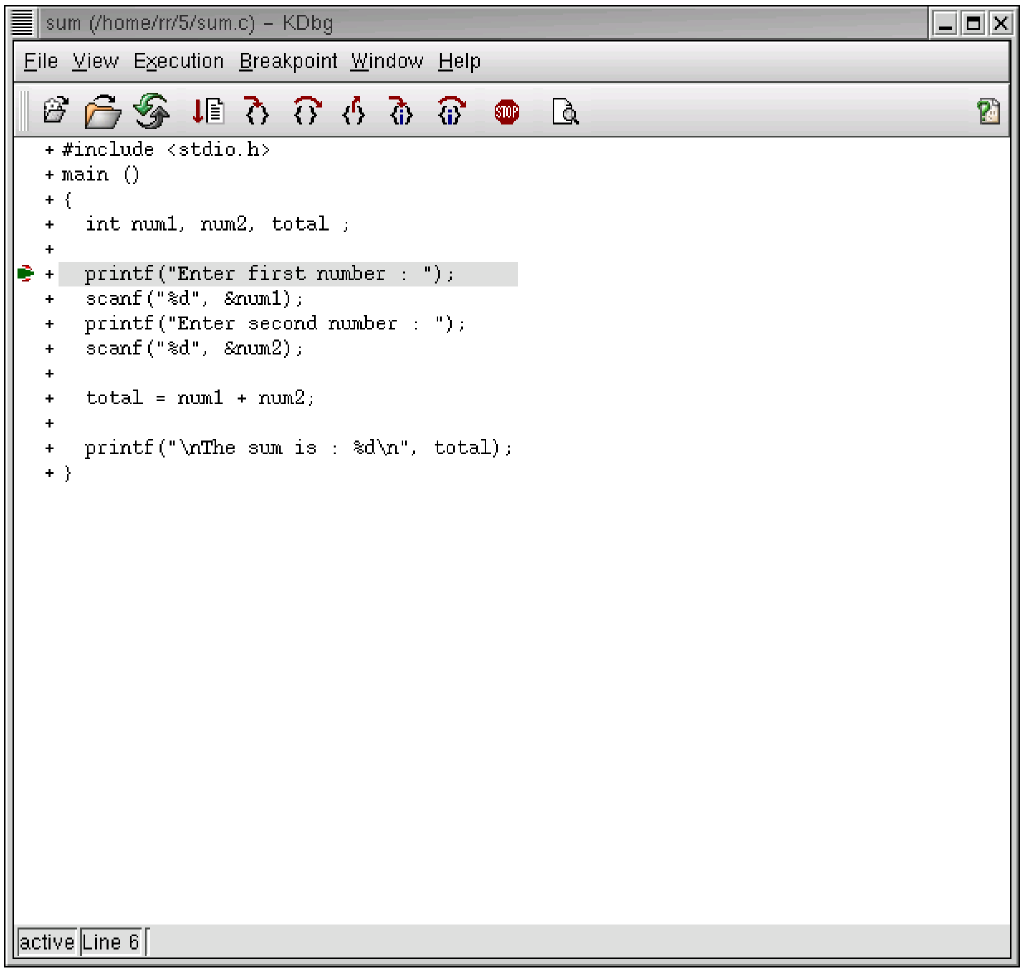
KDE debugger comes with KDE desktop development utilities. It is a GUI based upon GNU debugger. You can start it using the kdbg command or from a KDE menu. Using icons in the tool bar or menus, you can open files to be debugged. Once a file is loaded in the debugger window, you can click on a line to set or remove break points. To find out the purpose of an icon, you can move your cursor over the icon on the toolbar to display a short description.

The kdbg is a simple debugger but keep in mind that you may not find it as powerful as a native GNU debugger is. The window that appears when you start the debugger is shown in Figure 5-3. In this window the pointer is on the first printf statement where we have a break point set.

You can add a variable to the watch list using the watch window. This window can be opened by selecting the "Watched Expressions" option from the View menu. The window is shown in Figure 5-4 where two variables num1 and num2 are added to the watch list. You can click on a variable and then press the "Del" button to remove a variable from the watch list.

The kdbg also has an output window that is used to display any output from the program. This output window is actually an input window as well. You can enter input for the program in this window. The output window is started when you start the kdbg debugger. The output window is shown Figure 5-5.

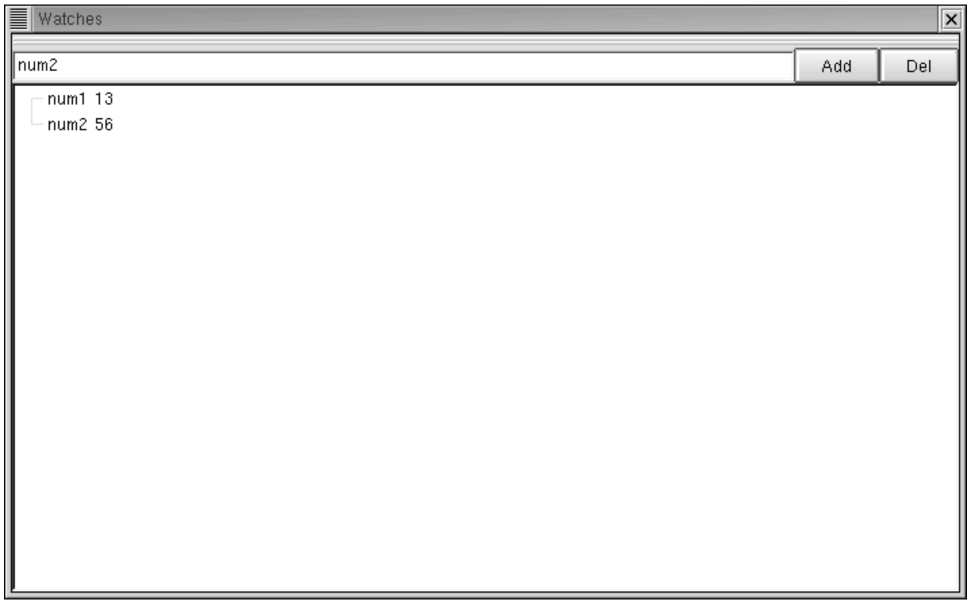
CH05.fm Page 170 Monday, October 7, 2002 9:00 PM


sum (/home/rr/5/sum.c) - KDbg
File View Execution Breakpoint Window Help
STOP
active
Line 6

```
+ #include <stdio.h>
+ main ()
+ {
+   int num1, num2, total ;
+
+   printf("Enter first number : ");
+   scanf("%d", &num1);
+   printf("Enter second number : ");
+   scanf("%d", &num2);
+
+   total = num1 + num2;
+
+   printf("\nThe sum is : %d\n", total);
+ }
```

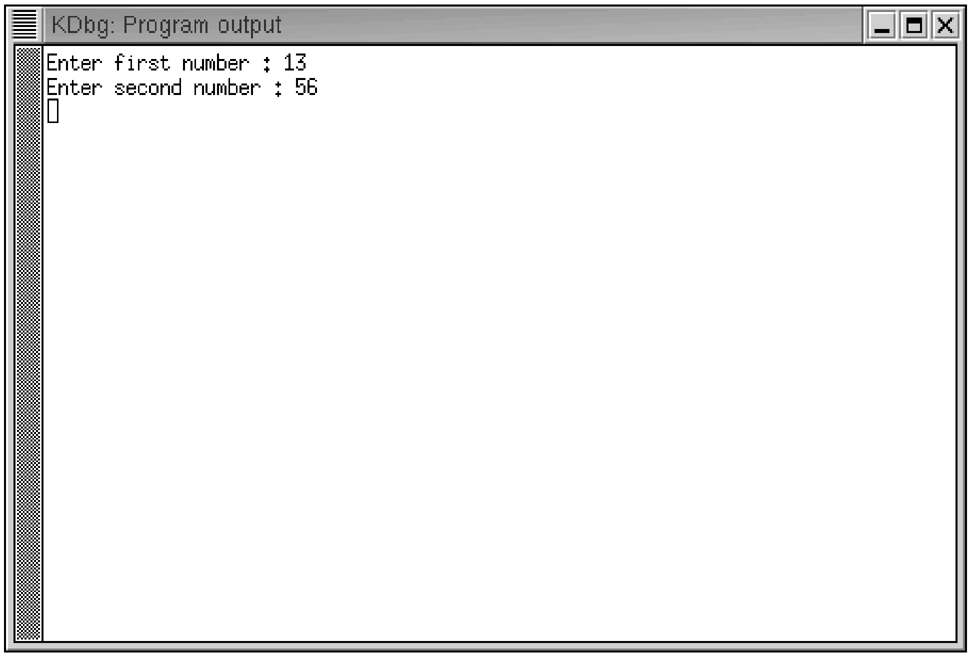
Figure 5-3 The `kdbg` window.

CH05.fm Page 171 Monday, October 7, 2002 9:00 PM



```
num2
num1 13
num2 56
```

Figure 5-4 Watch window of `kdbg` debugger.



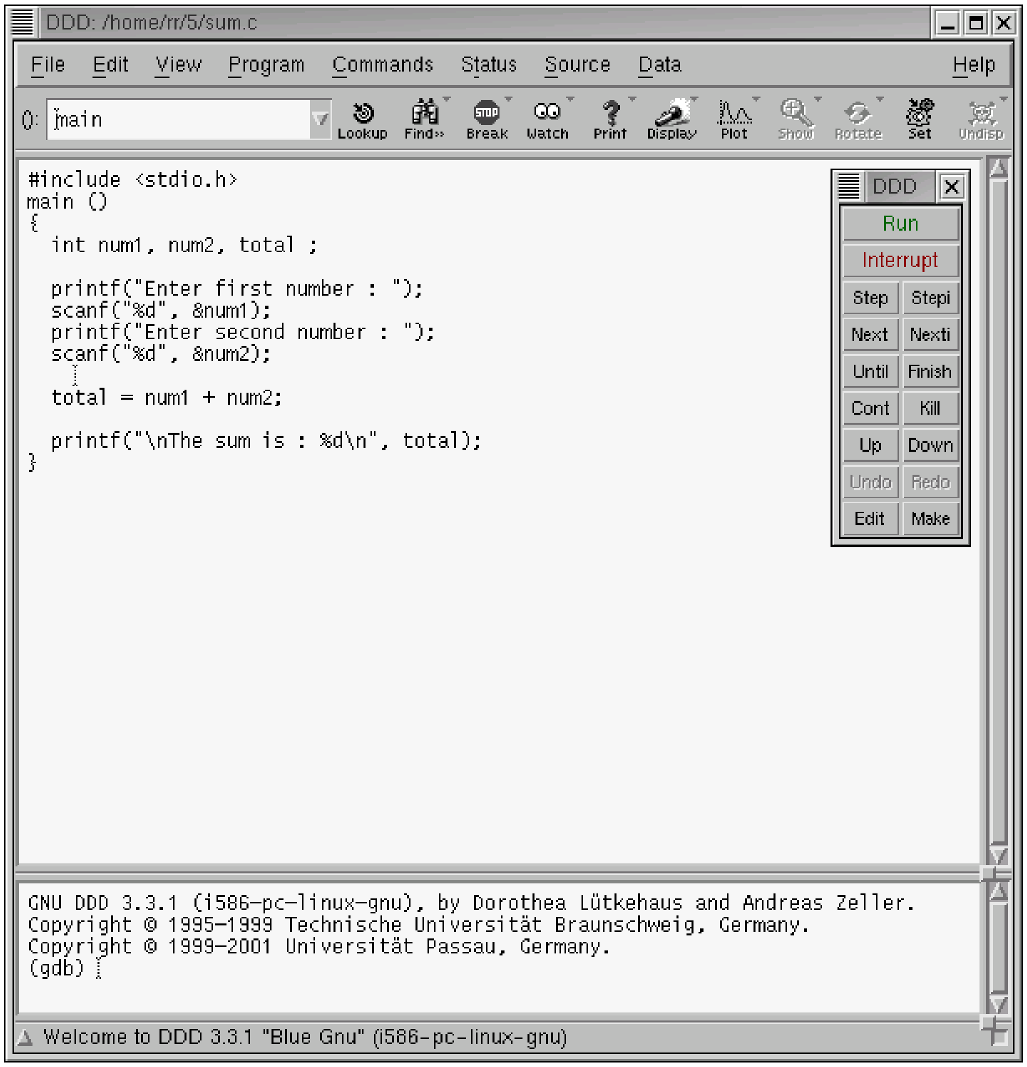
```
Enter first number : 13
Enter second number : 56
```

Figure 5-5 Output window for `kdbg`.

CH05.fm Page 172 Monday, October 7, 2002 9:00 PM

5.12.2 The ddd Debugger

The Data Display Debugger or `ddd` is a popular GUI front end to the GNU debugger. It has many additional features that are not present in native `gdb`. The main window of `ddd` debugger is shown in Figure 5-6. A complete discussion of this debugger is beyond the scope of this book.



```
#include <stdio.h>
main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = num1 + num2;

  printf("\nThe sum is : %d\n", total);
}
```

```
GNU DDD 3.3.1 (i586-pc-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
(gdb) 
```

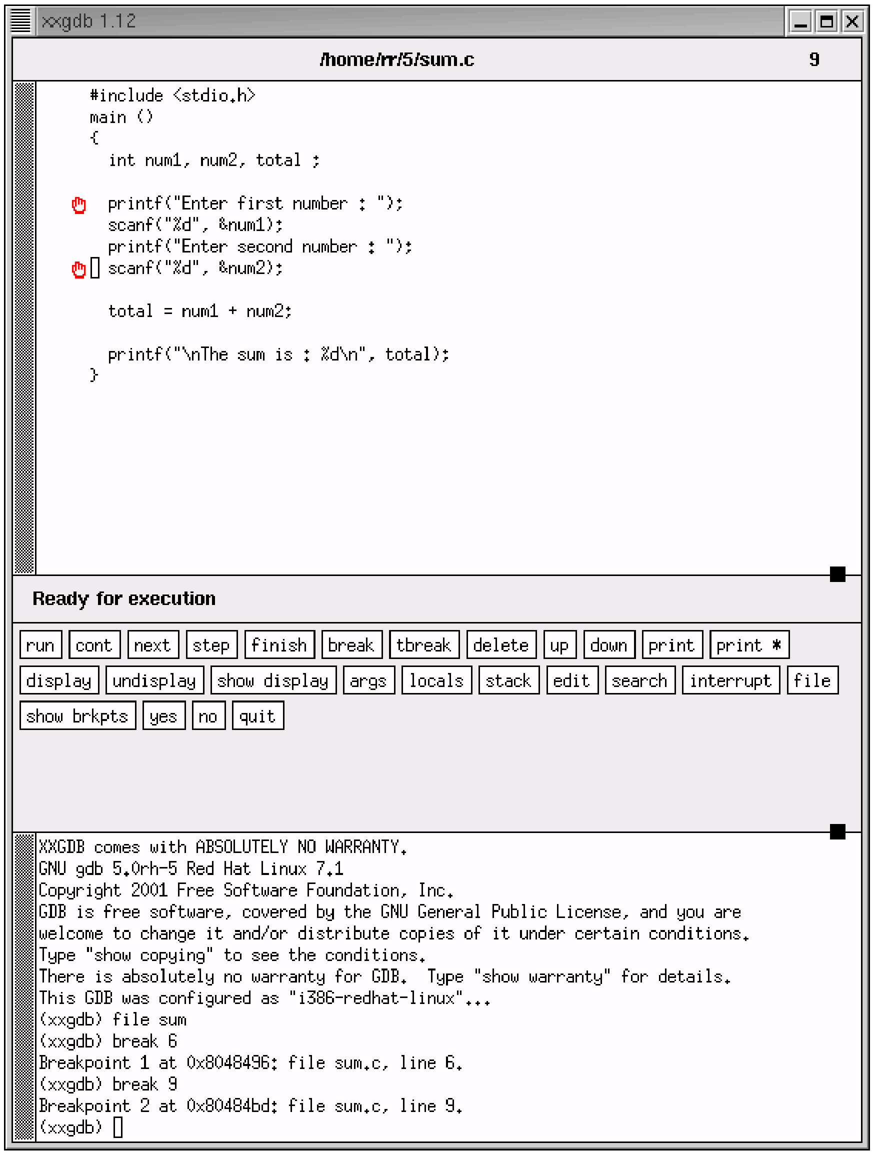
```
Welcome to DDD 3.3.1 "Blue Gnu" (i586-pc-linux-gnu)
```

Figure 5-6 Main window of `ddd` debugger.

CH05.fm Page 173 Monday, October 7, 2002 9:00 PM

5.12.3 The xxgdb Debugger

The `xxgdb` debugger is a GUI interface to the GNU debugger. You can use the command line window as well as GUI buttons with this debugger. The main screen is shown in Figure 5-7. The bottom part of the screen can be used to enter `gdb` commands while the top part can be used to scroll through source code and set or remove break points.



```
#include <stdio.h>
main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = num1 + num2;

  printf("\nThe sum is : %d\n", total);
}
```

```
XXGDB comes with ABSOLUTELY NO WARRANTY.
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"...
(xxgdb) file sum
(xxgdb) break 6
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(xxgdb) break 9
Breakpoint 2 at 0x80484bd: file sum.c, line 9.
(xxgdb) 
```

Figure 5-7 The `xxgdb` debugger main window.

For example, you can bring the cursor to a particular line and then press the break button to set up a break point at that line. Similarly you can use other buttons in the middle part of the window to carry out other tasks related to debugging a program.

5.13 References and Resources

1. GNU web site at http://www.gnu.org/
2. GNU debugger download web site ftp://ftp.gnu.org/gnu/gdb
3. DDD home page at http://www.gnu.org/software/ddd/
4. The gdb manual at http://www.gnu.org/manual/gdb/html_mono/gdb.html. You can find a detailed list of all commands and their explanation on this web site.