G

Using the GNU Debugger

Objectives

In this appendix you'll learn:

- To use the `run` command to run a program in the debugger.
- To use the `break` command to set a breakpoint.
- To use the `continue` command to continue execution.
- To use the `print` command to evaluate expressions.
- To use the `set` command to change variable values during program execution.
- To use the `step`, `finish` and `next` commands to control execution.
- To use the `watch` command to see how a data member is modified during program execution.
- To use the `delete` command to remove a breakpoint or a watchpoint.

Outline

G.1 Introduction

In Chapter 2, you learned that there are two types of errors—compilation errors and logic errors—and you learned how to eliminate compilation errors from your code. Logic errors do not prevent a program from compiling successfully, but they can cause the program to produce erroneous results when it runs. GNU includes software called a **debugger** that allows you to monitor the execution of your programs so you can locate and remove logic errors.

The debugger is one of the most important program development tools. Many IDEs provide their own debuggers similar to the one included in GNU or provide a graphical user interface to GNU's debugger. This appendix demonstrates key features of GNU's debugger. Appendix F discusses the features and capabilities of the Visual Studio debugger.

G.2 Breakpoints and the `run`, `stop`, `continue` and `print` Commands

We begin by investigating **breakpoints**, which are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether a logic error exists. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Note that attempting to set a breakpoint at a line of code that is not executable (such as a comment) will actually set the breakpoint at the next executable line of code in that function.

To illustrate the features of the debugger, we use the program listed in Fig. G.1, which finds the maximum of three integers. Execution begins in `main` (lines 8–22 of Fig. G.1). The three integers are input with `scanf` (line 15). Next, the integers are passed to `maximum` (line 19), which determines the largest integer. The value returned is returned to `main` by the `return` statement in `maximum` (line 38). The value returned is then printed in the `printf` statement (line 19).

```
1  // Fig. H.1: figH_01.c
2  // Finding the maximum of three integers
3  #include <stdio.h>
```

Fig. G.1 | Finds maximum of three integers. (Part 1 of 2.)

```
4
5  int maximum( int x, int y, int z ); // function prototype
6
7  // function main begins program execution
8  int main( void )
9  {
10    int number1; // first integer
11    int number2; // second integer
12    int number3; // third integer
13
14    printf( "%s", "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    // number1, number2 and number3 are arguments
18    // to the maximum function call
19    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20 } // end main
21
22 // Function maximum definition
23 // x, y and z are parameters
24 int maximum( int x, int y, int z )
25 {
26    int max = x; // assume x is largest
27
28    if ( y > max ) { // if y is larger than max, assign y to max
29       max = y;
30    } // end if
31
32    if ( z > max ) { // if z is larger than max, assign z to max
33       max = z;
34    } // end if
35
36    return max; // max is largest value
37 } // end function maximum
```

Fig. G.1 | Finds maximum of three integers. (Part 2 of 2.)

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `number1` declared in line 10 of Fig. G.1

1. *Compiling the program for debugging.* To use the debugger, you must compile your program with the **`-g`** option, which generates additional information that the debugger needs to help you debug your programs. To do so, type

```
gcc -g figH_01.c
```

2. *Starting the debugger.* Type `gdb ./a.out` (Fig. G.2). The **`gdb` command** starts the debugger and displays the `(gdb)` prompt at which you can enter commands.
3. *Running a program in the debugger.* Run the program through the debugger by typing **`run`** (Fig. G.3). If you do not set any breakpoints before running your program in the debugger, the program will run to completion.

```
$ gdb ./a.out
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb)
```

Fig. G.2 | Starting the debugger to run the program.

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17
Max is 85

Program exited normally.
(gdb)
```

Fig. G.3 | Running the program with no breakpoints set.

4. ***Inserting breakpoints using the GNU debugger.*** Set a breakpoint at line 14 of `figH_01.c` by typing `break 14`. The **`break` command** inserts a breakpoint at the line number specified as its argument (i.e., `22`, `85`, and `17`). You can set as many breakpoints as necessary. Each breakpoint is identified by the order in which it was created. The first breakpoint is known as `Breakpoint 1`. Set another breakpoint at line 19 by typing `break 19` (Fig. G.4). This new breakpoint is known as `Breakpoint 2`. When the program runs, it suspends execution at any line that contains a breakpoint and the debugger enters **break mode**. Breakpoints can be set even after the debugging process has begun. [*Note:* If you do not have a numbered listing for your code, you can use the **`list` command** to output your code with line numbers. For more information about the `list` command type **`help list`** from the `gdb` prompt.]

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figH_01.c, line 14.
(gdb) break 19
Breakpoint 2 at 0x8048440: file figH_01.c, line 19.
```

Fig. G.4 | Setting two breakpoints in the program.

5. ***Running the program and beginning the debugging process.*** Type `run` to execute your program and begin the debugging process (Fig. G.5). The debugger enters break mode when execution reaches the breakpoint at line 14. At this point, the debugger notifies you that a breakpoint has been reached and displays the source code at that line (14), which will be the next statement to execute.

```
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main() at figH_01.c:14
14                    scanf("%d%d%d", &number1, &number2, &number3);
(gdb)
```

Fig. G.5 | Running the program until it reaches the first breakpoint.

6. ***Using the* `continue` *command to resume execution.*** Type `continue`. The **`continue` command** causes the program to continue running until the next breakpoint is reached (line 19). Enter `22`, `85`, and `17` at the prompt. The debugger notifies you when execution reaches the second breakpoint (Fig. G.6). Note that `figH_01`'s normal output appears between messages from the debugger.

```
(gdb) continue
Continuing.
Enter three integers: 22 85 17

Breakpoint 2, main() at figH_01.c:19
19                    printf("Max is %d\n", maximum(number1, number2,
number3));
( gdb)
```

Fig. G.6 | Continuing execution until the second breakpoint is reached.

7. ***Examining a variable's value.*** Type `print number1` to display the current value stored in the `number1` variable (Fig. G.7). The **`print` command** allows you to peek inside the computer at the value of one of your variables. This can be used to help you find and eliminate logic errors in your code. In this case, the variable's value is `22`—the value you entered that was assigned to variable `number1` in line 15 of Fig. G.1.

```
(gdb) print number1
$1 = 22
(gdb)
```

Fig. G.7 | Printing the values of variables.

8. ***Using convenience variables.*** When you use `print`, the result is stored in a convenience variable such as `$1`. Convenience variables are temporary variables created by the debugger that are named using a dollar sign followed by an integer. Convenience variables can be used to perform arithmetic and evaluate boolean expressions. Type `print $1`. The debugger displays the value of `$1` (Fig. G.8), which contains the value of `number1`. Note that printing the value of `$1` creates a new convenience variable—`$2`.

```
(gdb) print $1
$2 = 22
(gdb)
```

Fig. G.8 | Printing a convenience variable.

9. ***Continuing program execution.*** Type `continue` to continue the program's execution. The debugger encounters no additional breakpoints, so it continues executing and eventually terminates (Fig. G.9).

```
(gdb) continue
Continuing
Max is 85

Program exited normally
(gdb)
```

Fig. G.9 | Finishing execution of the program.

10. ***Removing a breakpoint.*** You can display a list of all of the breakpoints in the program by typing **`info break`**. To remove a breakpoint, type **`delete`**, followed by a space and the number of the breakpoint to remove. Remove the first breakpoint by typing `delete 1`. Remove the second breakpoint as well. Now type `info break` to list the remaining breakpoints in the program. The debugger should indicate that no breakpoints are set (Fig. G.10).

```
(gdb) info break
Num  Type        Disp   Enb  Address       What
1    breakpoint  keep   y    0x080483e5    in main at figH_01.c:14
         breakpoint already hit 1 time
2    breakpoint  keep   y    0x08048799    in main at figH_01.c:19
         breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints
(gdb)
```

Fig. G.10 | Viewing and removing breakpoints.

11. ***Executing the program without breakpoints.*** Type `run` to execute the program. Enter the values `22`, `85`, and `17` at the prompt. Because you successfully removed the two breakpoints, the program's output is displayed without the debugger entering break mode (Fig. G.11).
12. ***Using the `quit` command.*** Use the **`quit` command** to end the debugging session (Fig. G.12). This command causes the debugger to terminate.

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17
Max is 85

Program exited normally.
(gdb)
```

Fig. G.11 | Program executing with no breakpoints set.

```
(gdb) quit
$
```

Fig. G.12 | Exiting the debugger using the `quit` command.

In this section, you used the `gdb` command to start the debugger and the `run` command to start debugging a program. You set a breakpoint at a particular line number in the `main` function. The `break` command can also be used to set a breakpoint at a line number in another file or at a particular function. Typing `break`, then the filename, a colon and the line number will set a breakpoint at a line in another file. Typing `break`, then a function name will cause the debugger to enter the break mode whenever that function is called.

Also in this section, you saw how the `help list` command will provide more information on the `list` command. If you have any questions about the debugger or any of its commands, type `help` or `help` followed by the command name for more information.

Finally, you examined variables with the `print` command and removed breakpoints with the `delete` command. You learned how to use the `continue` command to continue execution after a breakpoint is reached and the `quit` command to end the debugger.

G.3 print and set Commands

In the preceding section, you learned how to use the debugger's `print` command to examine the value of a variable during program execution. In this section, you'll learn how to use the `print` command to examine the value of more complex expressions. You'll also learn the **`set` command**, which allows you to assign new values to variables. We assume you are working in the directory containing this appendix's examples and have compiled for debugging with the `-g` compiler option.

1. ***Starting debugging.*** Type `gdb /a.out` to start the GNU debugger.
2. ***Inserting a breakpoint.*** Set a breakpoint at line 19 in the source code by typing `break 19` (Fig. G.13).

```
(gdb) break 19
Breakpoint 1 at 0x8048412: file figH_01.c, line 19.
(gdb)
```

Fig. G.13 | Setting a breakpoint in the program.

3. ***Running the program and reaching a breakpoint.*** Type `run` to begin the debugging process (Fig. G.14). This will cause `main` to execute until the breakpoint at line 19 is reached. This suspends program execution and switches the program into break mode. The statement in line 19 is the next statement that will execute.

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17

Breakpoint 1, main() at figH_01.c:19
19          printf("Max is %d\n", maximum(number1, number2, number3));
(gdb)
```

Fig. G.14 | Running the program until the breakpoint at line 19 is reached.

4. ***Evaluating arithmetic and boolean expressions.*** Recall from Section G.2 that once the debugger enters break mode, you can explore the values of the program's variables using the `print` command. You can also use `print` to evaluate arithmetic and boolean expressions. Type `print number1 - 2`. This expression returns the value `20` (Fig. G.15), but does not actually change the value of `number1`. Type `print number1 == 20`. Expressions containing the `==` symbol return `0` if the statement is false and `1` if the statement is true. The value returned is `0` (Fig. G.15) because `number1` still contains `22`.

```
(gdb) print number1 - 2
$1 = 20
(gdb) print number1 == 20
$2 = 0
(gdb)
```

Fig. G.15 | Printing expressions with the debugger.

5. ***Modifying values.*** You can change the values of variables during the program's execution in the debugger. This can be valuable for experimenting with different values and for locating logic errors. You can use the debugger's `set` command to change a variable's value. Type `set number1 = 90` to change the value of `number1`, then type `print number1` to display its new value (Fig. G.16).

```
(gdb) set number1 = 90
(gdb) print number1
$3 = 90
(gdb)
```

Fig. G.16 | Setting the value of a variable while in break mode.

6. ***Viewing the program result.*** Type `continue` to continue program execution. Line 19 of Fig. G.1 executes, passing `number1`, `number2` and `number3` to function `max-`

imum. Function main then displays the largest number. Note that the result is 90 (Fig. G.17). This shows that the preceding step changed the value of number1 from the value 22 that you input to 90.

```
(gdb) continue
Continuing.
Max is 90

Program exited normally.
(gdb)
```

Fig. G.17 | Using a modified variable in the execution of a program.

7. *Using the* **quit** *command.* Use the quit command to end the debugging session (Fig. G.18). This command causes the debugger to terminate.

```
(gdb) quit
$
```

Fig. G.18 | Exiting the debugger using the quit command.

In this section, you used the debugger's print command to evaluate arithmetic and boolean expressions. You also learned how to use the set command to modify the value of a variable during your program's execution.

G.4 Controlling Execution Using the step, finish and next Commands

Sometimes you'll need to execute a program line by line to find and fix errors. Walking through a portion of your program this way can help you verify that a function's code executes correctly. The commands in this section allow you to execute a function line by line, execute all the statements of a function at once or execute only the remaining statements of a function (if you've already executed some statements within the function).

1. *Starting the debugger.* Start the debugger by typing gdb ./a.out.
2. *Setting a breakpoint.* Type break 19 to set a breakpoint at line 19.
3. *Running the program.* Run the program by typing run, then enter 22, 85 and 17 at the prompt. The debugger then indicates that the breakpoint has been reached and displays the code at line 19. The debugger then pauses and waits for the next command to be entered.
4. *Using the* **step** *command.* The **step command** executes the next statement in the program. If the next statement to execute is a function call, control transfers to the called function. The step command enables you to enter a function and study its individual statements. For instance, you can use the print and set commands to view and modify the variables within the function. Type step to enter the maximum function (Fig. G.1). The debugger indicates that the step has been

completed and displays the next executable statement (Fig. G.19)—in this case, line 28 of `figH_01.c`.

```
(gdb) step
maximum (x=22, y=85, z=17) at figH_03.c:28
28      int max = x;
(gdb)
```

Fig. G.19 | Using the `step` command to enter a function.

5. ***Using the `finish` command.*** After you've stepped into the `debit` member function, type **`finish`**. This command executes the remaining statements in the function and returns control to the place where the function was called. The `finish` command executes the remaining statements in member function `debit`, then pauses at line 19 in `main` (Fig. G.20). The value returned by function `maximum` is also displayed. In lengthy functions, you may want to look at a few key lines of code, then continue debugging the caller's code. The `finish` command is useful for situations in which you do not want to step through the remainder of a function line by line.

```
(gdb) finish
Run till exit from #0   maximum ( x = 22, y = 85, z = 17) at figH_03.c:28
0x0804842b in main() at figH_03.c:19
19          printf("Max is %d\n", maximum(number1, number2, number3));
Value returned is $1 = 85
(gdb)
```

Fig. G.20 | Using the `finish` command to complete execution of a function and return to the calling function.

6. ***Using the `continue` command to continue execution.*** Enter the `continue` command to continue execution until the program terminates.
7. ***Running the program again.*** Breakpoints persist until the end of the debugging session in which they are set. So, the breakpoint you set in *Step 2* is still set. Type `run` to run the program and enter `22`, `85` and `17` at the prompt. As in *Step 3*, the program runs until the breakpoint at line 19 is reached, then the debugger pauses and waits for the next command (Fig. G.21).

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17

Breakpoint 1, main() at figH_03.c:19
19          printf("Max is %d\n", maximum(number1, number2, number3));
(gdb)
```

Fig. G.21 | Restarting the program.

8. *Using the* **`next`** *command.* Type **`next`**. This command behaves like the `step` command, except when the next statement to execute contains a function call. In that case, the called function executes in its entirety and the program advances to the next executable line after the function call (Fig. G.22). In *Step 4*, the `step` command enters the called function. In this example, the `next` command executes function `maximum` and outputs the largest of the three integers. The debugger then pauses at line 21.

```
(gdb) next
Max is 85
21        return 0; // indicates successful termination
(gdb)
```

Fig. G.22 | Using the `next` command to execute a function in its entirety.

9. *Using the* **`quit`** *command.* Use the `quit` command to end the debugging session (Fig. G.23). While the program is running, this command causes the program to immediately terminate rather than execute the remaining statements in `main`.

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

Fig. G.23 | Exiting the debugger using the `quit` command.

In this section, you used the debugger's `step` and `finish` commands to debug functions called during your program's execution. You saw how the `next` command can step over a function call. You also learned that the `quit` command ends a debugging session.

G.5 watch Command

The **watch command** tells the debugger to watch a data member. When that data member is about to change, the debugger will notify you. In this section, you'll use the `watch` command to see how the variable `number1` is modified during execution.

1. *Starting the debugger.* Start the debugger by typing `gdb ./a.out`.
2. *Setting a breakpoint and running the program.* Type `break 14` to set a breakpoint at line 14. Then, run the program with the command `run`. The debugger and program will pause at the breakpoint at line 14 (Fig. G.24).

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figH_03.c, line 14.
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main () at figH_03.c:14
14          printf("Enter three integers: ");
(gdb)
```

Fig. G.24 | Running the program until the first breakpoint.

3. *Watching a class's data member*. Set a watch on `number1` by typing `watch number1` (Fig. G.25). This watch is labeled as `watchpoint 2` because watchpoints are labeled with the same sequence of numbers as breakpoints. You can set a watch on any variable or data member of an object currently in scope. Whenever the value of a watched variable changes, the debugger enters break mode and notifies you that the value has changed. .

```
(gdb) watch number1
Hardware watchpoint2: number1
(gdb)
```

Fig. G.25 | Setting a watchpoint on a data member.

4. *Continuing execution.* Type `continue` to continue execution and enter three integers at the prompt. The debugger notifies you that the value of `number1` has changed and enters break mode (Fig. G.26). The old value of `number1` is its value before initialization. This value may be different each time the program executes. This unpredictable (and often undesirable) value demonstrates why it's important to initalize all C variables before they are used.

```
(gdb) continue
Continuing.
Enter three integers: 22 85 17
Hardware watchpoint 2: number1

Old value = -1208401328
New value = 22
0xb7e6c692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

Fig. G.26 | Entering break mode when a variable is changed.

5. *Continuing execution.* Type `continue`—the program will finish executing function `main`. The debugger removes the watch on `number1` because `number1` goes out of scope when function `main` ends. Removing the watchpoint causes the debugger to enter break mode. Type `continue` again to finish execution of the program (Fig. G.27).
6. *Restarting the debugger and resetting the watch on the variable.* Type `run` to restart the debugger. Once again, set a watch on `number1` by typing `watch number1`. This watchpoint is labeled as `watchpoint 3`. Type `continue` to continue execution (Fig. G.28).
7. *Removing the watch on the data member.* Suppose you want to watch a data member for only part of a program's execution. You can remove the debugger's watch on variable number1 by typing `delete 3` (Fig. G.29). Type `continue`—the program will finish executing without reentering break mode.

```
(gdb) continue
Continuing.
Max is 85

Watchpoint 2 is deleted because the program has left the block in
which its expression is valid.
0xb7e4aab7 in exit() from /lib/i686/cmov/libc.so.6
(gdb) continue
Continuing

Program exited normally
(gdb)
```

Fig. G.27 | Continuing to the end of the program.

```
(gdb) run
Starting program: /home/users/AppJ/a.out

Breakpoint 1, main () at figH_03.c:14
14         printf("Enter three integers: ");
(gdb) watch number1
Hardware watchpoint 3: number1
(gdb) continue
Continuing
Hardware watchpoint 3: number1

Old value = -1208798640
New value = 22
0xb7e0b692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

Fig. G.28 | Resetting the watch on a data member.

```
(gdb) delete 3
(gdb) continue
Continuing.
Max is 85

Program exited normally.
(gdb)
```

Fig. G.29 | Removing a watch.

In this section, you used the `watch` command to enable the debugger to notify you when the value of a variable changes. You used the `delete` command to remove a watch on a data member before the end of the program.

G.6 Wrap-Up

In this appendix, you learned how to insert and remove breakpoints in the debugger. Breakpoints allow you to pause program execution so you can examine variable values with

the debugger's `print` command, which can help you locate and fix logic errors. You used the `print` command to examine the value of an expression, and you used the `set` command to change the value of a variable. You also learned debugger commands (including the `step`, `finish` and `next` commands) that can be used to determine whether a function is executing correctly. You learned how to use the `watch` command to keep track of a data member throughout the scope of that data member. Finally, you learned how to use the `info break` command to list all the breakpoints and watchpoints set for a program and the `delete` command to remove individual breakpoints and watchpoints.