I

Using the GNU C++ Debugger

Objectives

In this appendix you'll:

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

Outline

I.1 Introduction
I.2 Breakpoints and the `run`, `stop`, `continue` and `print` Commands
I.3 `print` and `set` Commands
I.4 Controlling Execution Using the `step`, `finish` and `next` Commands
I.5 `watch` Command
I.6 Wrap-Up

I.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. For this appendix, we used GNU C++ on Ubuntu Linux.

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 H discusses the features and capabilities of the Visual Studio debugger. Appendix J discusses the features and capabilities of the Xcode debugger. Our C++ Resource Center (`www.deitel.com/cplusplus/`) provides links to tutorials that can help students and instructors familiarize themselves with the debuggers provided with various other development tools.

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

We begin our study of the debugger 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 class `Account` (Figs. I.1–I.2) and the program listed in Fig. I.3, which creates and manipulates an object of class `Account`. Execution begins in `main` (lines 8–25 of Fig. I.3). Line 10 creates an `Account` object with an initial balance of $50.00. `Account`'s constructor (lines 8–20 of Fig. I.2) accepts one argument, which specifies the `Account`'s initial `balance`. Line 13 of Fig. I.3 outputs the initial account balance using `Account` member function `getBalance`. Line 15 declares a local variable `withdrawalAmount` which stores a withdrawal amount input by the user. Line 17 prompts the user for the withdrawal amount; line 18 inputs the `withdrawalAmount`. Line 21 uses the `Account`'s `debit` member function to subtract the `withdrawalAmount` from the `Account`'s `balance`. Finally, line 24 displays the new `balance`.

```
// Fig. I.1: Account.h
// Definition of Account class.
class Account
{
public:
   Account( int ); // constructor initializes balance
   void credit( int ); // add an amount to the account balance
   void debit( int ); // subtract an amount from the account balance
   int getBalance(); // return the account balance
private:
   int balance; // data member that stores the balance
}; // end class Account
```

Fig. I.1 | Header file for the `Account` class.

```
// Fig. I.2: Account.cpp
// Member-function definitions for class Account.
#include <iostream>
#include "Account.h" // include definition of class Account
using namespace std;

// Account constructor initializes data member balance
Account::Account( int initialBalance )
{
   balance = 0; // assume that the balance begins at 0

   // if initialBalance is greater than 0, set this value as the
   // balance of the Account; otherwise, balance remains 0
   if ( initialBalance > 0 )
      balance = initialBalance;

   // if initialBalance is negative, print error message
   if ( initialBalance < 0 )
      cout << "Error: Initial balance cannot be negative.\n" << endl;
} // end Account constructor

// credit (add) an amount to the account balance
void Account::credit( int amount )
{
   balance = balance + amount; // add amount to balance
} // end function credit

// debit (subtract) an amount from the account balance
void Account::debit( int amount )
{
   if ( amount <= balance ) // debit amount does not exceed balance
      balance = balance - amount;
   else // debit amount exceeds balance
      cout << "Debit amount exceeded account balance.\n" << endl;
} // end function debit

```

Fig. I.2 | Definition for the `Account` class. (Part 1 of 2.)

```
// return the account balance
int Account::getBalance()
{
   return balance; // gives the value of balance to the calling function
} // end function getBalance
```

Fig. I.2 | Definition for the `Account` class. (Part 2 of 2.)

```
1  // Fig. I.3: figI_03.cpp
2  // Create and manipulate Account objects.
3  #include <iostream>
4  #include "Account.h"
5  using namespace std;
6
7  // function main begins program execution
8  int main()
9  {
10    Account account1( 50 ); // create Account object
11
12    // display initial balance of each object
13    cout << "account1 balance: $" << account1.getBalance() << endl;
14
15    int withdrawalAmount; // stores withdrawal amount read from user
16
17    cout << "\nEnter withdrawal amount for account1: "; // prompt
18    cin >> withdrawalAmount; // obtain user input
19    cout << "\nattempting to subtract " << withdrawalAmount
20       << " from account1 balance\n\n";
21    account1.debit( withdrawalAmount ); // try to subtract from account1
22
23    // display balances
24    cout << "account1 balance: $" << account1.getBalance() << endl;
25 } // end main
```

Fig. I.3 | Test class for debugging.

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in line 15 of Fig. I.3.

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

```
g++ -std=c++11 -g -o figI_03 figI_03.cpp Account.cpp
```

2. *Starting the debugger.* Type gdb `figI_03` (Fig. I.4). 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. I.5). If you do not set any breakpoints before running your program in the debugger, the program will run to completion.

```
paul deitel@ubuntu:~/Documents/examples/appI$ gdb figI_03
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/pauldeitel/Documents/examples/appI/figI_03...done.
(gdb)
```

Fig. I.4 | Starting the debugger to run the program.

```
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

[Inferior 1 (process 27091) exited normally]
(gdb)
```

Fig. I.5 | Running the program with no breakpoints set.

4. *Inserting breakpoints using the GNU debugger.* Set a breakpoint at line 13 of `FigI_03.cpp` by typing `break 13`. The **`break` command** inserts a breakpoint at the line number specified as its argument (i.e., `13`). 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 21 by typing `break 21` (Fig. I.6). 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 13
Breakpoint 1 at 0x804881a: file figI_03.cpp, line 13.
(gdb) break 21
Breakpoint 2 at 0x80488b4: file figI_03.cpp, line 21.
(gdb)
```

Fig. I.6 | 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. I.7). The debugger enters break mode when execution reaches the breakpoint at line 13. At this point, the debugger notifies you that a breakpoint has been reached and displays the source code at that line (13), which will be the next statement to execute.

```
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03

Breakpoint 1, main () at figI_03.cpp:13
13        cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

Fig. I.7 | 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 21). Enter `13` at the prompt. The debugger notifies you when execution reaches the second breakpoint (Fig. I.8). Note that `figI_03`'s normal output appears between messages from the debugger.

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 2, main () at figI_03.cpp:21
21        account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.8 | Continuing execution until the second breakpoint is reached.

7. ***Examining a variable's value.*** Type `print withdrawalAmount` to display the current value stored in the `withdrawalAmount` variable (Fig. I.9). 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 `13`—the value you entered that was assigned to variable `withdrawalAmount` in line 18 of Fig. I.3. Next, use `print` to display the contents of the `account1` object. When an object is displayed with `print`, braces are placed around the object's data members. In this case, there is a single data member—`balance`—which has a value of `50`.

```
(gdb) print withdrawalAmount
$1 = 13
(gdb) print account1
$2 = {balance = 50}
(gdb)
```

Fig. I.9 | 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. I.10), which contains the value of `withdrawalAmount`. Note that printing the value of `$1` creates a new convenience variable—`$3`.

```
(gdb) print $1
$3 = 13
(gdb)
```

Fig. I.10 | 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. I.11).

```
(gdb) continue
Continuing.
account1 balance: $37
[Inferior 1 (process 13291) exited normally]
(gdb)
```

Fig. I.11 | 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. I.12).
11. ***Executing the program without breakpoints.*** Type `run` to execute the program. Enter the value `13` at the prompt. Because you successfully removed the two breakpoints, the program's output is displayed without the debugger entering break mode (Fig. I.13).

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804881a in main() at figI_03.cpp:13
        breakpoint already hit 1 time
2       breakpoint     keep y   0x080488b4 in main() at figI_03.cpp:21
        breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Fig. I.12 | Viewing and removing breakpoints.

```
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37
[Inferior 1 (process 13366) exited normally]
(gdb)
```

Fig. I.13 | Program executing with no breakpoints set.

12. *Using the `quit` command.* Use the **`quit` command** to end the debugging session (Fig. I.14). This command causes the debugger to terminate.

```
(gdb) quit
pauldeitel@ubuntu:~/Documents/examples/appI$
```

Fig. I.14 | 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 remove 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.

I.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 figI_03` to start the GNU debugger.
2. *Inserting a breakpoint.* Set a breakpoint at line 21 in the source code by typing `break 21` (Fig. I.15).

```
(gdb) break 21
Breakpoint 1 at 0x8048799: file FigI_03.cpp, line 21.
(gdb)
```

Fig. I.15 | Setting a breakpoint in the program.

3. *Running the program and reaching a breakpoint.* Type `run` to begin the debugging process (Fig. I.16). This will cause `main` to execute until the breakpoint at line 21 is reached. This suspends program execution and switches the program into break mode. The statement in line 25 is the next statement that will execute.

```
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at figI_03.cpp:21
21        account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.16 | Running the program until the breakpoint at line 25 is reached.

4. *Evaluating arithmetic and boolean expressions.* Recall from Section I.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 withdrawalAmount - 2`. This expression returns the value `11` (Fig. I.17), but does not actually change the value of `withdrawalAmount`. Type `print withdrawalAmount == 11`. Expressions containing the `==` symbol return `bool` values. The value returned is `false` (Fig. I.17) because `withdrawalAmount withdrawalAmount` still contains `13`.

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

Fig. I.17 | 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 withdrawalAmount = 42` to change the value of `withdrawalAmount`, then type `print withdrawalAmount` to display its new value (Fig. I.18).

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

Fig. I.18 | Setting the value of a variable while in break mode.

6. *Viewing the program result.* Type `continue` to continue program execution. Line 21 of Fig. I.3 executes, passing `withdrawalAmount` to `Account` member function `debit`. Function `main` then displays the new balance. Note that the result is `$8` (Fig. I.19). This shows that the preceding step changed the value of `withdrawalAmount` from the value `13` that you input to `42`.

```
(gdb) continue
Continuing.
account1 balance: $8
[Inferior 1 (process 13377) exited normally]
(gdb)
```

Fig. I.19 | 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. I.20). This command causes the debugger to terminate.

```
(gdb) quit
$
```

Fig. I.20 | 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.

I.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 figI_03`.
2. *Setting a breakpoint.* Type `break 21` to set a breakpoint at line 21.
3. *Running the program.* Run the program by typing `run`, then enter 13 at the prompt. After the program displays its two output messages, the debugger indicates that the breakpoint has been reached and displays the code at line 21. The debugger then pauses and wait 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 `debit` member function of class `Account` (Fig. I.2). The debugger indicates that the step has been completed and displays the next executable statement (Fig. I.21)—in this case, line 31 of class `Account` (Fig. I.2).

```
(gdb) step
Account::debit (this=0xbffff308, amount=13) at Account.cpp:31
31        if ( amount <= balance ) // debit amount does not exceed balance
(gdb)
```

Fig. I.21 | 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 24 in `main` (Fig. I.22). 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.
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 `13` at the prompt. As in *Step 3*, the program runs until the breakpoint at line 21 is reached, then the debugger pauses and waits for the next command (Fig. I.23).

```
(gdb) finish
Run till exit from #0  Account::debit (this=0xbffff308, amount=13)
    at Account.cpp:31
main () at figI_03.cpp:24
24          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

Fig. I.22 | Using the `finish` command to complete execution of a function and return to the calling function.

```
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at figI_03.cpp:21
21          account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.23 | 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. I.24). In *Step 4*, the `step` command entered the called function. In this example, the `next` command executes `Account` member function `debit`, then the debugger pauses at line 24.

```
(gdb) next
24          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

Fig. I.24 | 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. I.25). While the program is running, this command causes the program to immediately terminate rather than execute the remaining statements in `main`.

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.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 13410] will be killed.

Quit anyway? (y or n) y
pauldeitel@ubuntu:~/Documents/examples/appI$
```

Fig. I.25 | Exiting the debugger using the `quit` command.

I.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 `Account` object's data member `balance` is modified during execution.

1. ***Starting the debugger.*** Start the debugger by typing `gdb figI_03`.
2. ***Setting a breakpoint and running the program.*** Type `break 10` to set a breakpoint at line 10. Then, run the program with the command `run`. The debugger and program will pause at the breakpoint at line 10 (Fig. I.26).

```
(gdb) break 10
Breakpoint 1 at 0x8048806: file figI_03.cpp, line 10.
(gdb) run
Starting program: /home/pauldeitel/Documents/examples/appI/figI_03

Breakpoint 1, main () at figI_03.cpp:10
10        Account account1( 50 ); // create Account object
(gdb)
```

Fig. I.26 | Running the program until the first breakpoint.

3. ***Watching a class's data member.*** Set a watch on `account1`'s `balance` data member by typing `watch account1.balance` (Fig. I.27). 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 account1.balance
Hardware watchpoint 2: account1.balance
(gdb)
```

Fig. I.27 | Setting a watchpoint on a data member.

4. ***Executing the constructor.*** Use the `next` command to execute the constructor and initialize the `account1` object's `balance` data member. The debugger indicates that the `balance` data member's value changed, shows the old and new values and enters break mode at line 18 (Fig. I.28).

```
(gdb) next
Hardware watchpoint 2: account1.balance

Old value = 134515353
New value = 0
Account::Account (this=0xbffff308, initialBalance=50) at Account.cpp:14
14        if ( initialBalance > 0 )
(gdb)
```

Fig. I.28 | Stepping into the constructor.

5. ***Exiting the constructor.*** Type `finish` to complete the constructor's execution and return to `main`.
6. ***Withdrawing money from the account.*** Type `continue` to continue execution and enter a withdrawal value at the prompt. The program executes normally. Line 21 of Fig. I.3 calls `Account` member function `debit` to reduce the `Account` object's `balance` by a specified `amount`. Line 32 of Fig. I.2 inside function `debit` changes the value of `balance`. The debugger notifies you of this change and enters break mode (Fig. I.29).

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x080489f5 in Account::debit (this=0xbffff308, amount=13) at Account.cpp:32
32           balance = balance - amount;
(gdb)
```

Fig. I.29 | Entering break mode when a variable is changed.

7. ***Continuing execution.*** Type `continue`—the program will finish executing function `main` because the program does not attempt any additional changes to `balance`. The debugger removes the watch on `account1`'s `balance` data member because the `account1` object 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. I.30).
8. ***Restarting the debugger and resetting the watch on the variable.*** Type `run` to restart the debugger. Once again, set a watch on `account1` data member `balance` by typing `watch account1.balance`. This watchpoint is labeled as `watchpoint 3`. Type `continue` to continue execution (Fig. I.31).

```
(gdb) continue
Continuing.
account1 balance: $37

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0xb7d314d3 in __libc_start_main () from /lib/i386-linux-gnu/libc.so.6
(gdb)
```

Fig. I.30 | Continuing to the end of the program.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/pauldeitel/Documents/examples/appI/figI_03

Breakpoint 1, main () at figI_03.cpp:10
10          Account account1( 50 ); // create Account object
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 134515353
New value = 0
Account::Account (this=0xbffff308, initialBalance=50) at Account.cpp:14
14          if ( initialBalance > 0 )
(gdb)
```

Fig. I.31 | Resetting the watch on a data member.

9. ***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 `balance` by typing `delete 3` (Fig. I.32). Type `continue`—the program will finish executing without reentering break mode.

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

Fig. I.32 | 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.

I.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.