5

GNU Compiler Collection (GCC)

Objectives

- To be able to use a text editor to create C++ source files.
- To be able to use GCC to compile and execute C++ applications with single source files.
- To be able to use GCC to compile and execute C++ applications with multiple source files.
- To be able to debug C++ applications using the GDB debugger.

Outline

5.1 Introduction

Welcome to the GNU Compiler Collection. In this chapter you will learn how to create, compile and execute C++ programs using the C++ development tool from GNU—GCC. When you complete this chapter, you will be able to use GCC to run applications. This guide is suitable for use as a companion text in a first year university C++ programming course sequence.

This guide does not teach C++; rather, this guide is intended to be used as a companion to our textbook C++ How To Program, Fourth Edition or any other ANSI/ISO C++ textbook. Many of our readers have asked us to provide a supplement that would introduce the fundamental command line concepts using GCC and a basic overview of how to use the GNU debugger. Our readers asked us to use the same “live-code” approach with outputs that we employ in all our How to Program Series textbooks.

Before proceeding with this chapter, you should be familiar with the topics in Chapter 1, “Introduction to Computers and C++ Programming”, Chapter 2, "Control Structures", Chapter 3, "Functions" and Chapter 6, "Classes and Data Abstraction" of C++ How to Program, Fourth Edition. We hope you enjoy learning about the GCC command line compiler with this guide.

5.2 Creating a C++ program

Before creating a C++ program, create a directory to store your files. We created a directory named **/Welcome**, you of course can choose a different name.

You are now ready to create a program. Open a text editor and type in a program, such as the following: [*Note*: We have include line numbers to improve readability of our example, however they are not part of the program and should not be included.]

```
// Welcome.cpp

#include <iostream>

using std::cout;
using std::endl;
```

Fig. 5.1 Code for **Welcome.cpp**

```

int main()
{
   cout << "Welcome to C++!" << endl;

   return 0;   // indicates successful termination

} // end main
```

Fig. 5.1 Code for **Welcome.cpp**

To make your programs easier to read, adhere to the spacing conventions described in C++ How To Program, Fourth Edition.

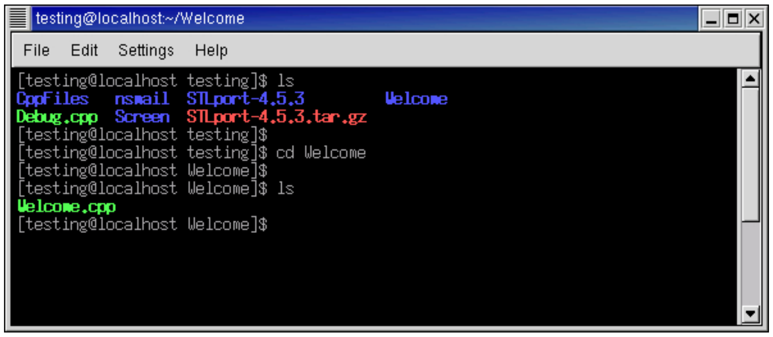
Save the file with a **.cpp** extension, which signifies to the compiler that the file is a C++ file. In our example, we named the file **Welcome.cpp**.

5.3 GNU Compiler Collection (GCC)

GCC is a command line based compiler. It can be used to compile and execute C, C++ and Fortran code. Although many Linux installations include a version of GCC by default, the latest version is available from **gcc.gnu.org/**. There you will find links to download GCC and information on how to compile and install. To access the help menu, enter **g++ --help** into the command prompt. This displays a list of help topics and information about flags that can be raised to the compiler. This help is useful for basic questions and command-line syntax.

5.3.1 Compiling and Executing a Program with GCC

1. In a terminal, use the **cd** command to traverse the directory structure and get to the right folder (**/Welcome**). The steps can be done individually as shown in Figure 5.2 below. The **ls** command displays the contents of a folder.

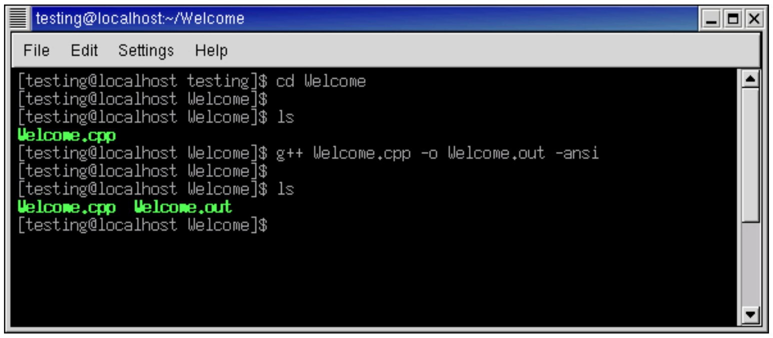


```
[testing@localhost testing]$ ls
CppFiles   nsmail  STLport-4.5.3         Welcome
Debug.cpp  Screen  STLport-4.5.3.tar.gz
[testing@localhost testing]$
[testing@localhost testing]$ cd Welcome
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ls
Welcome.cpp
[testing@localhost Welcome]$
```

Fig. 5.2 Changing to the correct directory.

2. Once in the appropriate folder, use the GCC compiler to compile the program. To do this, enter **g++ *FileName*.cpp -o *OutputFileName*.out**. The **g++** command signifies that the C++ compiler should be used instead of the C compiler. *FileName* is the name of the **.cpp** file that is to be compiled. The **-o** flag spec-

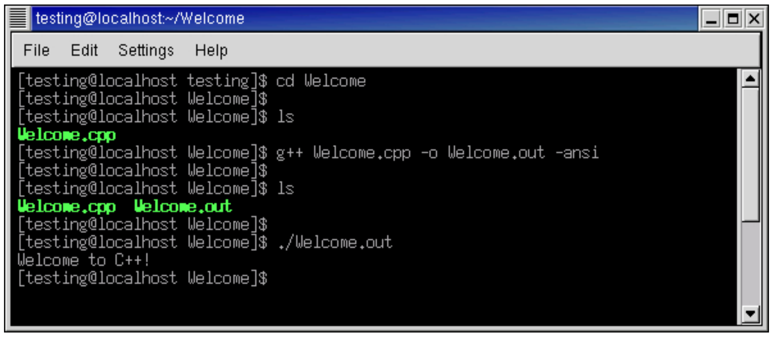
ifies that the output file should not receive the default name and the *OutputFileName* is what the **`.out`** file will be called. If the **`-o`** flag is not raised, the output file is named **`a.out`**. The command used to compile **`Welcome.cpp`** is **`g++ Welcome.cpp -o Welcome.out -ansi`**. Raise the **`-ansi`** flag to conform to ANSI/ISO standards.



```
[testing@localhost testing]$ cd Welcome
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ls
Welcome.cpp
[testing@localhost Welcome]$ g++ Welcome.cpp -o Welcome.out -ansi
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ls
Welcome.cpp  Welcome.out
[testing@localhost Welcome]$
```

Fig. 5.3 Compiling **`Welcome.cpp`**.

3. GCC returns the user to the prompt if there are no syntax errors. It might appear as if nothing has changed, but use the **ls** command to reveal the **`Welcome.out`** file. To execute this program, enter **`./Welcome.out`**. The program executes and displays the **Welcome to C++!** output.



```
[testing@localhost testing]$ cd Welcome
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ls
Welcome.cpp
[testing@localhost Welcome]$ g++ Welcome.cpp -o Welcome.out -ansi
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ls
Welcome.cpp  Welcome.out
[testing@localhost Welcome]$
[testing@localhost Welcome]$ ./Welcome.out
Welcome to C++!
[testing@localhost Welcome]$
```

Fig. 5.4 Executing **`Welcome.exe`**.

5.3.2 Compiling programs with Multiple Source Files

More complex programs often consist of multiple C++ source files. We introduce this concept, called multiple source files, in chapter 6 of C++ How to Program, Fourth Edition. This section explains how to compile a program with multiple source files using the GCC compiler

Compiling a program, which has two or more source files, can be accomplished two ways. The first method requires listing all the files on the command line. The second method uses the wild-card character(*). Using the wild-card character followed by **`.cpp`** will give you access to all the files with the **.cpp** extension in the current directory. For example, typing **ls *.cpp** at the command prompt will list all files with the **.cpp** extension in the current directory. Both methods of compiling multiple files will be demonstrated using **Hello.cpp** (Fig. 5.5) and **Welcome2.cpp** (Fig. 5.6).

```
// Hello.cpp

#include <iostream>

using std::cout;
using std::endl;

// print text to the output stream
void myFunction()
{
   cout << "Hello from Hello.cpp!" << endl;

} // end function myFunction
```

Fig. 5.5 Code for **Hello.cpp**

```
// Welcome2.cpp

#include <iostream>

using std::cout;
using std::endl;

void myFunction();   // function prototype

int main()
{
   cout << "Welcome to C++!" << endl;
   myFunction();   // call function myFunction

   return 0; // indicates successful termination

} // end main
```

Fig. 5.6 Code for **Welcome2.cpp**

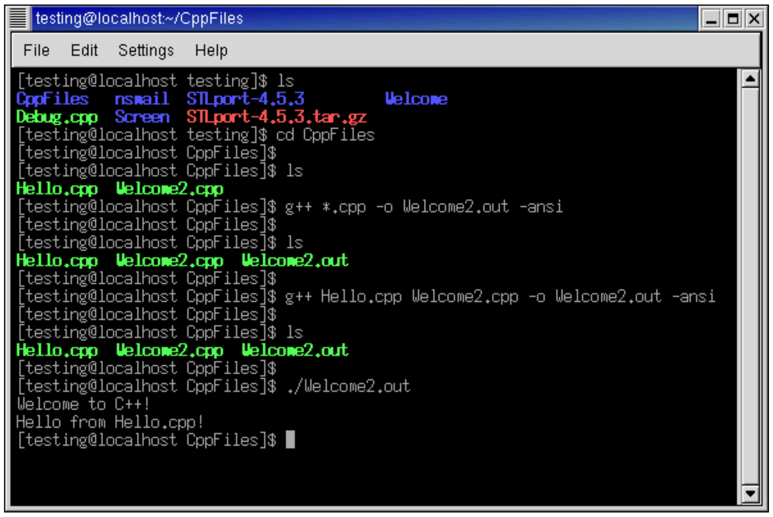
For this example the files are placed in the **/CppFiles** directory. Use the **cd** command to get into the folder containing the source files.

To compile the program by listing the files on the command line, type:

```
g++ Hello.cpp Welcome2.cpp -o Welcome2.out -ansi
```

To compile the program using the wild-card character, type:

```
g++ *.cpp -o Welcome2.out -ansi
```



```
[testing@localhost testing]$ ls
CppFiles   nsmail  STLport-4.5.3         Welcome
Debug.cpp  Screen  STLport-4.5.3.tar.gz
[testing@localhost testing]$ cd CppFiles
[testing@localhost CppFiles]$
[testing@localhost CppFiles]$ ls
Hello.cpp  Welcome2.cpp
[testing@localhost CppFiles]$ g++ *.cpp -o Welcome2.out -ansi
[testing@localhost CppFiles]$
[testing@localhost CppFiles]$ ls
Hello.cpp  Welcome2.cpp  Welcome2.out
[testing@localhost CppFiles]$
[testing@localhost CppFiles]$ g++ Hello.cpp Welcome2.cpp -o Welcome2.out -ansi
[testing@localhost CppFiles]$
[testing@localhost CppFiles]$ ls
Hello.cpp  Welcome2.cpp  Welcome2.out
[testing@localhost CppFiles]$
[testing@localhost CppFiles]$ ./Welcome2.out
Welcome to C++!
Hello from Hello.cpp!
[testing@localhost CppFiles]$ █
```

Fig. 5.7 Compiling multiple C++ source files.

Figure 5.7 demonstrates both methods of compiling mutliple source files, each creating an executable file with the name **`Welcome2.out`**. As with compilation of programs with a single source file, if the **`-o`** flag is not raised, the executable file will be name **`a.out`**.

5.4 The STLPort Library

GCC is an open source application. For this reason its standard library is not fully conformant to ANSI/ISO standards. As GCC moves towards a fully-conformant standard library, a free library provided by STLPort is available as a replacement and offers a standard compliant I/O streams library. This library can be downloaded from **`www.STLport.org`**. Enter the **Download** section, and save the file named **`STLport-4.5.3.tar.gz`**. Extract it using the command **`tar -zxvf STLport-4.5.3.tar.gz`**.

1. From the command line, switch to the STLPort directory. By default, the command is **`cd ~/STLport-4.5.3`**. The makefiles are located within the **src** directory; enter this directory using **`cd src`**.
2. The makefiles are accessible from the **src** directory. Enter the command **`make -f gcc-linux.mak`** to begin compiling the libraries.
3. When completed, enter **`make -f gcc-linux.mak install`** to begin the installation of the new library.
4. To export the library into the local path, enter the following command into the command line: **`export LD_LIBRARY_PATH=/usr/local/lib`**
5. The new library is now installed. To test it, compile a program with the command **`g++ -I/usr/local/include/stlport`** ***`File`*** **`.cpp -L/usr/local/`**

lib -lstlport_gcc -pthread -o *ExecutableFile*. *File* is the C++ source file and *ExecutableFile* is the compiled program.

5.5 Using the GDB Debugger

GNU provides the GDB debugger to help programmers find run-time logic errors in programs that compile and link successfully but do not produce expected results. The debugger lets the programmer view the executing program and its data as the program runs either one step at a time or at full speed. The program stops on a selected line of code or upon a fatal run-time error. When the programmer does not understand how incorrect results are produced by a program, running the program one statement at a time and monitoring the intermediate results can help the programmer isolate the cause of error. The programmer can correct the code.

GDB can be used in two ways. One way is using the console window and the other is using GDB under *emacs* (i.e., a text editor that is available in most Linux machines), which gives the programmer the opportunity to view and edit the source code while debugging the program. Because emacs provides the programmer a much easier and cleaner user interface than the console window, we use it to demonstrate how to debug applications with the GDB debugger.

To use the debugger under emacs, compile the program with the **-g** flag. The **-g** flag generates debugging symbols that the debugger can use to examine and manipulate data while the program is executing. After compiling the program, open the source code and set one or more breakpoints. A breakpoint is a marker set at a specified line of code that causes the debugger to suspend execution of the program upon reaching that line of code. Breakpoints help the programmer verify that a program is executing correctly. When a breakpoint is set, emacs displays a confirmation (i.e., **break Debug.cpp:28**) on the bottom of the window that the breakpoint has been set. To add breakpoints, use the **break** command or the shortcut *Control–x* followed by *Space*. The **continue** command continues the execution of the program after a breakpoint. Shortcuts for the most used commands are displayed under the GUD menu.

Often, certain variables are monitored by the programmer during the debugging process- a process known as setting a ***watch***. The **watch** command allows the programmer to monitor variables as their values change. The debugger buffer displays the old and the new values, whenever the variable value is changed. Variable values can also be modified during the debugging process.

The **step** command executes program statements, including code in functions that are called, allowing the programmer to confirm the proper execution of the functions, line-by-line. The **next** command executes the next executable line of code and advances to the following executable line in the program. If the line of code contains a function call, the function is executed in its entirety as one step. This allows the user to execute the program one line at a time and examine the execution of the program without seeing the details of every function that is called.

5.5.1 Debugging an Application

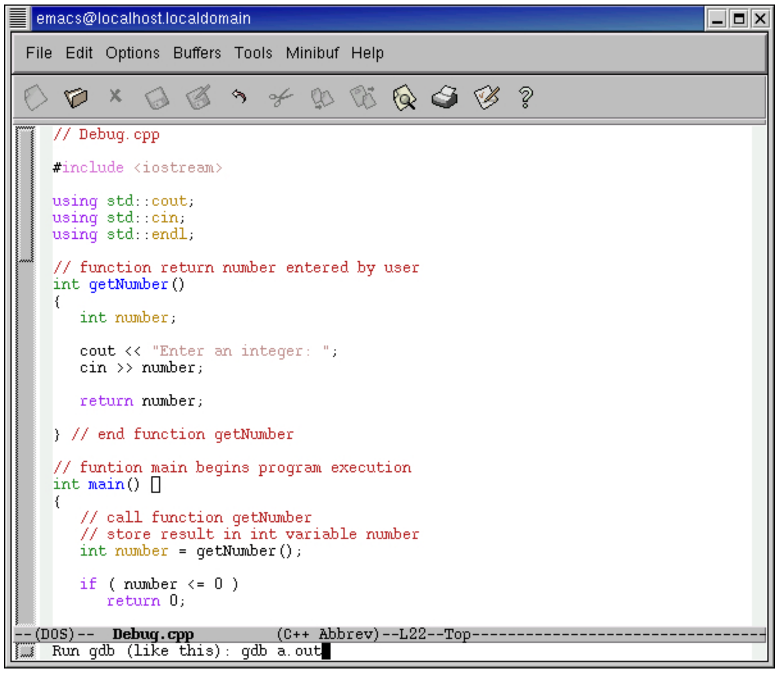
This section guides the user through the debugging process using the GDB debugger. The sample program, **Debug.cpp** (Fig. 5.8), is provided to guide you through the process of debugging a simple C++ application. This application obtains a number from the user and counts all the numbers from one up to the number entered.

```cpp
// Debug.cpp

#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// function return number entered by user
int getNumber()
{
   int number;

   cout << "Enter an integer: ";
   cin >> number;

   return number;

} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;

   else
      for ( int i = 1; i <= number; i++ )
         cout << i << endl;   // print from 1 to the number

   return 0;   // indicate that program ended successfully

} // end main
```

Fig. 5.8 Code for **`Debug.cpp`**

1. The first step is to compile the program with the **`-g`** flag. This allows the debugger to show the code for the program as opposed to the memory locations. This is done by typing **`g++ Debug.cpp -g`** in the terminal window. Then, to open the source file using emacs, type **`emacs Debug.cpp`**. Once the source file is opened, Select **Tools** > **Debugger (GUD) ...** to run GDB under emacs. Type **`a.out`** in the minibuffer window, located on the bottom of the window, and press *Enter*. There are now two buffer windows open. One is the source code buffer and the other is

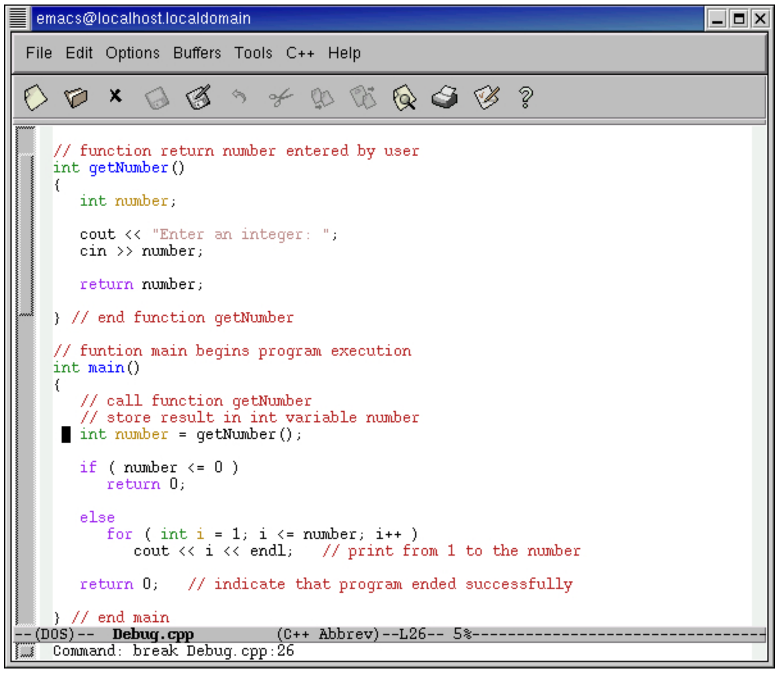
the debugger buffer. To switch between them select **Buffers** from the menu and then select the buffer desired.


emacs@localhost.localdomain
File Edit Options Buffers Tools Minibuf Help

```
// Debug.cpp

#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// function return number entered by user
int getNumber()
{
   int number;

   cout << "Enter an integer: ";
   cin >> number;

   return number;

} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;
--(DOS)--  Debug.cpp        (C++ Abbrev)--L22--Top----------------------------------
Run gdb (like this): gdb a.out
```

Fig. 5.9   Using the GDB debugger in emacs.

2. The next step is to add a breakpoint. In the source code buffer window, click on the line where the breakpoint is to be located and press *Control-x* followed by *Space*, or enter the command **`break`** *line number* in the debugger buffer. Add a breakpoint to line 26, which is the first executable line inside the main function.

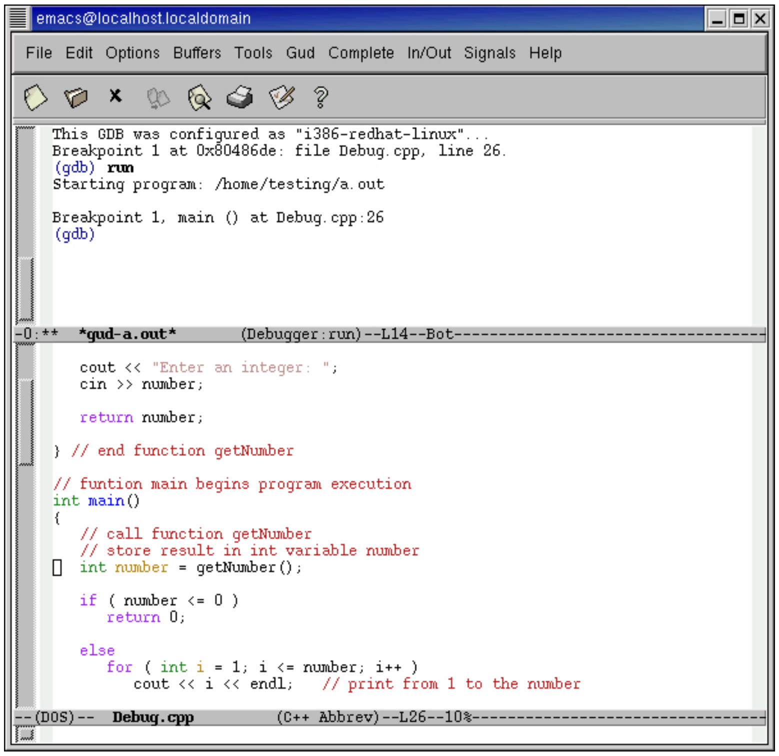

emacs@localhost.localdomain
File Edit Options Buffers Tools C++ Help

```
// function return number entered by user
int getNumber()
{
   int number;

   cout << "Enter an integer: ";
   cin >> number;

   return number;

} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;

   else
      for ( int i = 1; i <= number; i++ )
         cout << i << endl;   // print from 1 to the number

   return 0;   // indicate that program ended successfully

} // end main
--(DOS)--  Debug.cpp        (C++ Abbrev)--L26-- 5%-----------------------------------
Command: break Debug.cpp:26
```

Fig. 5.10 Setting a breakpoint.

3. Type **run** in the debugger buffer and press *Enter*. The program starts execution and suspends on line 26. The emacs window splits up in two and you will be able to see the debugger buffer and the source code buffer at the same time. Once the execution breaks, the prompt goes back to the debugger buffer and you can enter other commands. To set a watch, enter **watch** *variable name* and press *Enter*. Notice that watches can only be set to variables that have been initialized already. For

example, a watch can only be set to variable **`i`**, when the debugger reaches line 32. Notice that all **`watch`**es are deleted when the program execution terminates.

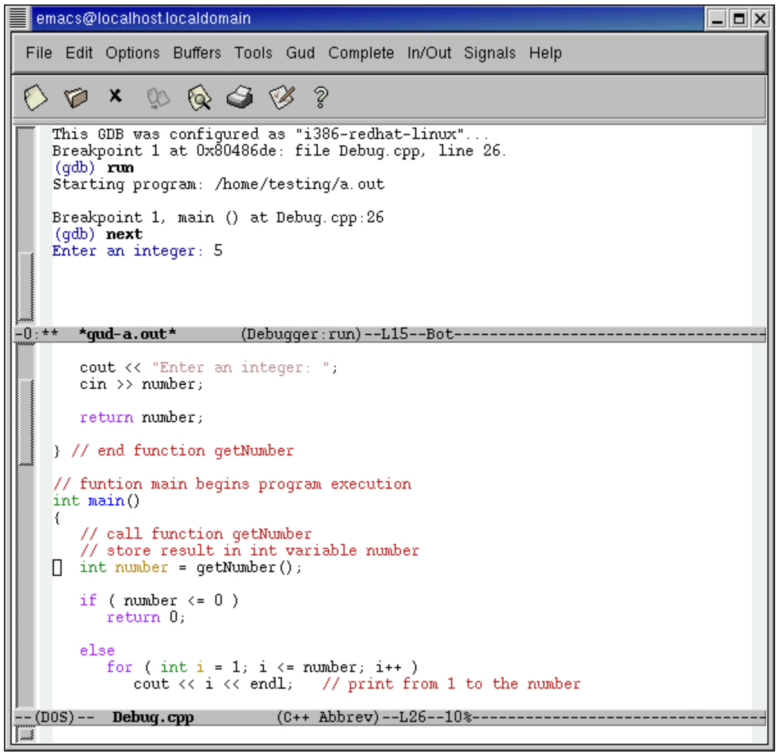

emacs@localhost.localdomain
File Edit Options Buffers Tools Gud Complete In/Out Signals Help

```
This GDB was configured as "i386-redhat-linux"...
Breakpoint 1 at 0x80486de: file Debug.cpp, line 26.
(gdb) run
Starting program: /home/testing/a.out

Breakpoint 1, main () at Debug.cpp:26
(gdb)
-0:**  *gud-a.out*      (Debugger:run)--L14--Bot-----------------------------------

   cout << "Enter an integer: ";
   cin >> number;

   return number;

} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;

   else
      for ( int i = 1; i <= number; i++ )
         cout << i << endl;   // print from 1 to the number
--(DOS)--  Debug.cpp       (C++ Abbrev)--L26--10%--------------------------------
```

Fig. 5.11 Begin debugging **`Debug.cpp`**.

4. Enter **`next`** and enter **`5`** when the program prompts the user to enter a number. Notice that function **`getNumber`** has been called on line 26 and will be entirely

executed without any breaks. Entering **`step`** would have caused the program to step into the function call of **`getNumber`**.


emacs@localhost.localdomain
File Edit Options Buffers Tools Gud Complete In/Out Signals Help

```
This GDB was configured as "i386-redhat-linux"...
Breakpoint 1 at 0x80486de: file Debug.cpp, line 26.
(gdb) run
Starting program: /home/testing/a.out

Breakpoint 1, main () at Debug.cpp:26
(gdb) next
Enter an integer: 5
```

```
-0:**  *gud-a.out*        (Debugger:run)--L15--Bot------------------------------------
```

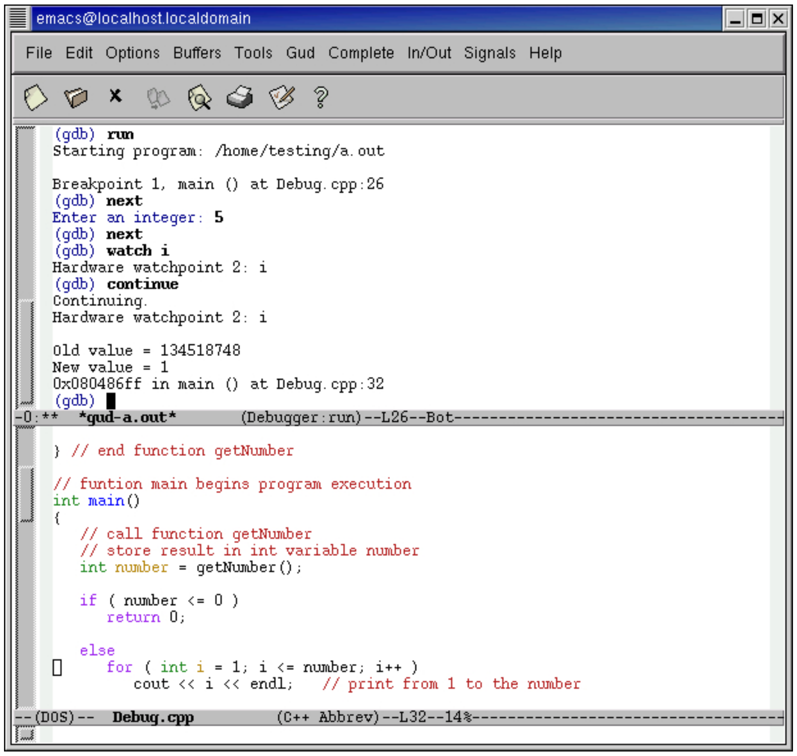
```
   cout << "Enter an integer: ";
   cin >> number;

   return number;

} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;

   else
      for ( int i = 1; i <= number; i++ )
         cout << i << endl;   // print from 1 to the number
```

```
--(DOS)--  Debug.cpp        (C++ Abbrev)--L26--10%--------------------------------
```

Fig. 5.12 Using the **`next`** command.

5. Execution suspends on line 28. Enter **`next`** and the debugger reaches line 32. Add a **`watch`** for the variable **`i`**. This watch can only be added when within the scope of **`i`**, meaning inside the **`for`** loop. Every time the value of **`i`** changes, the debug-

ger displays the old and the new values of the variable. Notice that the output is also displayed in the debugger buffer.


emacs@localhost.localdomain
File Edit Options Buffers Tools Gud Complete In/Out Signals Help

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

Breakpoint 1, main () at Debug.cpp:26
(gdb) next
Enter an integer: 5
(gdb) next
(gdb) watch i
Hardware watchpoint 2: i
(gdb) continue
Continuing.
Hardware watchpoint 2: i

Old value = 134518748
New value = 1
0x080486ff in main () at Debug.cpp:32
(gdb) 
-0:**  *gud-a.out*       (Debugger:run)--L26--Bot-----------------------------------
} // end function getNumber

// funtion main begins program execution
int main()
{
   // call function getNumber
   // store result in int variable number
   int number = getNumber();

   if ( number <= 0 )
      return 0;

   else
      for ( int i = 1; i <= number; i++ )
         cout << i << endl;   // print from 1 to the number
--(DOS)--  Debug.cpp       (C++ Abbrev)--L32--14%---------------------------------
```

Fig. 5.13 Setting a watch for variable i.

6. Enter **`continue`** until the debugger iterates through the **`for`** loop and program execution terminates.
7. Perform the debugging session once again and experiment with the different options that are available.