I

Advanced UNIX Programming with Linux

1 Getting Started

THIS CHAPTER SHOWS YOU HOW TO PERFORM THE BASIC steps required to create a C or C++ Linux program. In particular, this chapter shows you how to create and modify C and C++ source code, compile that code, and debug the result. If you're already accustomed to programming under Linux, you can skip ahead to Chapter 2, "Writing Good GNU/Linux Software;" pay careful attention to Section 2.3, "Writing and Using Libraries," for information about static versus dynamic linking that you might not already know.

Throughout this book, we'll assume that you're familiar with the C or C++ programming languages and the most common functions in the standard C library. The source code examples in this book are in C, except when demonstrating a particular feature or complication of C++ programming. We also assume that you know how to perform basic operations in the Linux command shell, such as creating directories and copying files. Because many Linux programmers got started programming in the Windows environment, we'll occasionally point out similarities and contrasts between Windows and Linux.

1.1 Editing with Emacs

An *editor* is the program that you use to edit source code. Lots of different editors are available for Linux, but the most popular and full-featured editor is probably GNU Emacs.

About Emacs

Emacs is much more than an editor. It is an incredibly powerful program, so much so that at CodeSourcery, it is affectionately known as the One True Program, or just the OTP for short. You can read and send email from within Emacs, and you can customize and extend Emacs in ways far too numerous to discuss here. You can even browse the Web from within Emacs!

If you're familiar with another editor, you can certainly use it instead. Nothing in the rest of this book depends on using Emacs. If you don't already have a favorite Linux editor, then you should follow along with the mini-tutorial given here.

If you like Emacs and want to learn about its advanced features, you might consider reading one of the many Emacs books available. One excellent tutorial, *Learning GNU Emacs*, is written by Debra Cameron, Bill Rosenblatt, and Eric S. Raymond (O'Reilly, 1996).

1.1.1 Opening a C or C++ Source File

You can start Emacs by typing `emacs` in your terminal window and pressing the Return key. When Emacs has been started, you can use the menus at the top to create a new source file. Click the Files menu, choose Open Files, and then type the name of the file that you want to open in the "minibuffer" at the bottom of the screen.[1] If you want to create a C source file, use a filename that ends in `.c` or `.h`. If you want to create a C++ source file, use a filename that ends in `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.C`, or `.H`. When the file is open, you can type as you would in any ordinary word-processing program. To save the file, choose the Save Buffer entry on the Files menu. When you're finished using Emacs, you can choose the Exit Emacs option on the Files menu.

If you don't like to point and click, you can use keyboard shortcuts to automatically open files, save files, and exit Emacs. To open a file, type `C-x C-f`. (The `C-x` means to hold down the Control key and then press the `x` key.) To save a file, type `C-x C-s`. To exit Emacs, just type `C-x C-c`. If you want to get a little better acquainted with Emacs, choose the Emacs Tutorial entry on the Help menu. The tutorial provides you with lots of tips on how to use Emacs effectively.

1. If you're not running in an X Window system, you'll have to press F10 to access the menus.

1.1.2 Automatic Formatting

If you're accustomed to programming in an *Integrated Development Environment (IDE)*, you'll also be accustomed to having the editor help you format your code. Emacs can provide the same kind of functionality. If you open a C or C++ source file, Emacs automatically figures out that the file contains source code, not just ordinary text. If you hit the Tab key on a blank line, Emacs moves the cursor to an appropriately indented point. If you hit the Tab key on a line that already contains some text, Emacs indents the text. So, for example, suppose that you have typed in the following:

```
int main ()
{
printf ("Hello, world\n");
}
```

If you press the Tab key on the line with the call to `printf`, Emacs will reformat your code to look like this:

```
int main ()
{
  printf ("Hello, world\n");
}
```

Notice how the line has been appropriately indented.

As you use Emacs more, you'll see how it can help you perform all kinds of complicated formatting tasks. If you're ambitious, you can program Emacs to perform literally any kind of automatic formatting you can imagine. People have used this facility to implement Emacs modes for editing just about every kind of document, to implement games[2], and to implement database front ends.

1.1.3 Syntax Highlighting

In addition to formatting your code, Emacs can make it easier to read C and C++ code by coloring different syntax elements. For example, Emacs can turn keywords one color, built-in types such as `int` another color, and comments another color. Using color makes it a lot easier to spot some common syntax errors.

The easiest way to turn on colorization is to edit the file `~/.emacs` and insert the following string:

```
(global-font-lock-mode t)
```

Save the file, exit Emacs, and restart. Now open a C or C++ source file and enjoy!

You might have noticed that the string you inserted into your `.emacs` looks like code from the LISP programming language. That's because it *is* LISP code! Much of Emacs is actually written in LISP. You can add functionality to Emacs by writing more LISP code.

2. Try running the command `M-x dunnet` if you want to play an old-fashioned text adventure game.

1.2 Compiling with GCC

A *compiler* turns human-readable source code into machine-readable object code that can actually run. The compilers of choice on Linux systems are all part of the GNU Compiler Collection, usually known as GCC.[3] GCC also include compilers for C, C++, Java, Objective-C, Fortran, and Chill. This book focuses mostly on C and C++ programming.

Suppose that you have a project like the one in Listing 1.2 with one C++ source file (`reciprocal.cpp`) and one C source file (`main.c`) like in Listing 1.1. These two files are supposed to be compiled and then linked together to produce a program called `reciprocal`.[4] This program will compute the reciprocal of an integer.

Listing 1.1 **(*main.c*) C source file—*main.c***

```
#include <stdio.h>
#include "reciprocal.hpp"

int main (int argc, char **argv)
{
  int i;

  i = atoi (argv[1]);
  printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
  return 0;
}
```

Listing 1.2 **(*reciprocal.cpp*) C++ source file—*reciprocal.cpp***

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
  // I should be non-zero.
  assert (i != 0);
  return 1.0/i;
}
```

3. For more information about GCC, visit `http://gcc.gnu.org`.

4. In Windows, executables usually have names that end in `.exe`. Linux programs, on the other hand, usually have no extension. So, the Windows equivalent of this program would probably be called `reciprocal.exe`; the Linux version is just plain `reciprocal`.

There's also one header file called `reciprocal.hpp` (see Listing 1.3).

Listing 1.3 (*reciprocal.hpp*) Header file—*reciprocal.hpp*

```
#ifdef __cplusplus
extern "C" {
#endif

extern  double reciprocal (int i);

#ifdef __cplusplus
}
#endif
```

The first step is to turn the C and C++ source code into object code.

1.2.1 Compiling a Single Source File

The name of the C compiler is `gcc`. To compile a C source file, you use the `-c` option. So, for example, entering this at the command prompt compiles the `main.c` source file:

```
% gcc -c main.c
```

The resulting object file is named `main.o`.

The C++ compiler is called `g++`. Its operation is very similar to `gcc`; compiling `reciprocal.cpp` is accomplished by entering the following:

```
% g++ -c reciprocal.cpp
```

The `-c` option tells `g++` to compile the program to an object file only; without it, `g++` will attempt to link the program to produce an executable. After you've typed this command, you'll have an object file called `reciprocal.o`.

You'll probably need a couple other options to build any reasonably large program. The `-I` option is used to tell GCC where to search for header files. By default, GCC looks in the current directory and in the directories where headers for the standard libraries are installed. If you need to include header files from somewhere else, you'll need the `-I` option. For example, suppose that your project has one directory called `src`, for source files, and another called `include`. You would compile `reciprocal.cpp` like this to indicate that `g++` should use the `../include` directory in addition to find `reciprocal.hpp`:

```
% g++ -c -I ../include reciprocal.cpp
```

Sometimes you'll want to define macros on the command line. For example, in production code, you don't want the overhead of the assertion check present in `reciprocal.cpp`; that's only there to help you debug the program. You turn off the check by defining the macro `NDEBUG`. You could add an explicit `#define` to `reciprocal.cpp`, but that would require changing the source itself. It's easier to simply define `NDEBUG` on the command line, like this:

```
% g++ -c -D NDEBUG reciprocal.cpp
```

If you had wanted to define `NDEBUG` to some particular value, you could have done something like this:

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

If you're really building production code, you probably want to have GCC optimize the code so that it runs as quickly as possible. You can do this by using the `-O2` command-line option. (GCC has several different levels of optimization; the second level is appropriate for most programs.) For example, the following compiles `reciprocal.cpp` with optimization turned on:

```
% g++ -c -O2 reciprocal.cpp
```

Note that compiling with optimization can make your program more difficult to debug with a debugger (see Section 1.4, "Debugging with GDB"). Also, in certain instances, compiling with optimization can uncover bugs in your program that did not manifest themselves previously.

You can pass lots of other options to `gcc` and `g++`. The best way to get a complete list is to view the online documentation. You can do this by typing the following at your command prompt:

```
% info gcc
```

1.2.2 Linking Object Files

Now that you've compiled `main.c` and `utilities.cpp`, you'll want to link them. You should always use `g++` to link a program that contains C++ code, even if it also contains C code. If your program contains only C code, you should use `gcc` instead. Because this program contains both C and C++, you should use `g++`, like this:

```
% g++ -o reciprocal main.o reciprocal.o
```

The `-o` option gives the name of the file to generate as output from the link step. Now you can run `reciprocal` like this:

```
% ./reciprocal 7
The reciprocal of 7 is 0.142857
```

As you can see, `g++` has automatically linked in the standard C runtime library containing the implementation of `printf`. If you had needed to link in another library (such as a graphical user interface toolkit), you would have specified the library with the `-l` option. In Linux, library names almost always start with `lib`. For example, the Pluggable Authentication Module (PAM) library is called `libpam.a`. To link in `libpam.a`, you use a command like this:

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

The compiler automatically adds the `lib` prefix and the `.a` suffix.

As with header files, the linker looks for libraries in some standard places, including the `/lib` and `/usr/lib` directories that contain the standard system libraries. If you want the linker to search other directories as well, you should use the `-L` option, which is the parallel of the `-I` option discussed earlier. You can use this line to instruct the linker to look for libraries in the `/usr/local/lib/pam` directory before looking in the usual places:

```
% g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
```

Although you don't have to use the `-I` option to get the preprocessor to search the current directory, you do have to use the `-L` option to get the linker to search the current directory. In particular, you could use the following to instruct the linker to find the `test` library in the current directory:

```
% gcc -o app app.o -L. -ltest
```

1.3 Automating the Process with GNU Make

If you're accustomed to programming for the Windows operating system, you're probably accustomed to working with an Integrated Development Environment (IDE). You add sources files to your project, and then the IDE builds your project automatically. Although IDEs are available for Linux, this book doesn't discuss them. Instead, this book shows you how to use GNU Make to automatically recompile your code, which is what most Linux programmers actually do.

The basic idea behind `make` is simple. You tell `make` what *targets* you want to build and then give *rules* explaining how to build them. You also specify *dependencies* that indicate when a particular target should be rebuilt.

In our sample `reciprocal` project, there are three obvious targets: `reciprocal.o`, `main.o`, and the `reciprocal` itself. You already have rules in mind for building these targets in the form of the command lines given previously. The dependencies require a little bit of thought. Clearly, `reciprocal` depends on `reciprocal.o` and `main.o` because you can't link the complete program until you have built each of the object files. The object files should be rebuilt whenever the corresponding source files change. There's one more twist in that a change to `reciprocal.hpp` also should cause both of the object files to be rebuilt because both source files include that header file.

In addition to the obvious targets, there should always be a `clean` target. This target removes all the generated object files and programs so that you can start fresh. The rule for this target uses the `rm` command to remove the files.

You can convey all that information to `make` by putting the information in a file named `Makefile`. Here's what `Makefile` contains:

```
reciprocal: main.o reciprocal.o
        g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
        gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
        g++ $(CFLAGS) -c reciprocal.cpp

clean:
        rm -f *.o reciprocal
```

You can see that targets are listed on the left, followed by a colon and then any dependencies. The rule to build that target is on the next line. (Ignore the `$(CFLAGS)` bit for the moment.) The line with the rule on it must start with a Tab character, or `make` will get confused. If you edit your `Makefile` in Emacs, Emacs will help you with the formatting.

If you remove the object files that you've already built, and just type

```
% make
```

on the command-line, you'll see the following:

```
% make
gcc -c main.c
g++ -c reciprocal.cpp
g++ -o reciprocal main.o reciprocal.o
```

You can see that `make` has automatically built the object files and then linked them. If you now change `main.c` in some trivial way and type `make` again, you'll see the following:

```
% make
gcc -c main.c
g++ -o reciprocal main.o reciprocal.o
```

You can see that `make` knew to rebuild `main.o` and to re-link the program, but it didn't bother to recompile `reciprocal.cpp` because none of the dependencies for `reciprocal.o` had changed.

The `$(CFLAGS)` is a `make` variable. You can define this variable either in the `Makefile` itself or on the command line. GNU `make` will substitute the value of the variable when it executes the rule. So, for example, to recompile with optimization enabled, you would do this:

```
% make clean
rm -f *.o reciprocal
% make CFLAGS=-O2
gcc -O2 -c main.c
g++ -O2 -c reciprocal.cpp
g++ -O2 -o reciprocal main.o reciprocal.o
```

Note that the `-O2` flag was inserted in place of `$(CFLAGS)` in the rules.

In this section, you've seen only the most basic capabilities of `make`. You can find out more by typing this:

```
% info make
```

In that manual, you'll find information about how to make maintaining a `Makefile` easier, how to reduce the number of rules that you need to write, and how to automatically compute dependencies. You can also find more information in *GNU, Autoconf, Automake, and Libtool* by Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor (New Riders Publishing, 2000).

1.4 Debugging with GNU Debugger (GDB)

The *debugger* is the program that you use to figure out why your program isn't behaving the way you think it should. You'll be doing this a lot.[5] The GNU Debugger (GDB) is the debugger used by most Linux programmers. You can use GDB to step through your code, set breakpoints, and examine the value of local variables.

1.4.1 Compiling with Debugging Information

To use GDB, you'll have to compile with debugging information enabled. Do this by adding the `-g` switch on the compilation command line. If you're using a `Makefile` as described previously, you can just set `CFLAGS` equal to `-g` when you run `make`, as shown here:

```
% make CFLAGS=-g
gcc -g -c main.c
g++ -g -c reciprocal.cpp
g++ -g -o reciprocal main.o reciprocal.o
```

When you compile with `-g`, the compiler includes extra information in the object files and executables. The debugger uses this information to figure out which addresses correspond to which lines in which source files, how to print out local variables, and so forth.

1.4.2 Running GDB

You can start up `gdb` by typing:

```
% gdb reciprocal
```

When `gdb` starts up, you should see the GDB prompt:

```
(gdb)
```

5. …unless your programs always work the first time.

The first step is to run your program inside the debugger. Just enter the command `run` and any program arguments. Try running the program without any arguments, like this:

```
(gdb) run
Starting program: reciprocal

Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287     strtol.c: No such file or directory.
(gdb)
```

The problem is that there is no error-checking code in `main`. The program expects one argument, but in this case the program was run with no arguments. The `SIGSEGV` message indicates a program crash. GDB knows that the actual crash happened in a function called `__strtol_internal`. That function is in the standard library, and the source isn't installed, which explains the "No such file or directory" message. You can see the stack by using the `where` command:

```
(gdb) where
#0  __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
    at strtol.c:287
#1  0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

You can see from this display that `main` called the `atoi` function with a `NULL` pointer, which is the source of the trouble.

You can go up two levels in the stack until you reach `main` by using the `up` command:

```
(gdb) up 2
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

Note that `gdb` is capable of finding the source for `main.c`, and it shows the line where the erroneous function call occurred. You can view the value of variables using the `print` command:

```
(gdb) print argv[1]
$2 = 0x0
```

That confirms that the problem is indeed a `NULL` pointer passed into `atoi`.

You can set a breakpoint by using the `break` command:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

This command sets a breakpoint on the first line of `main`.[6] Now try rerunning the program with an argument, like this:

```
(gdb) run 7
Starting program: reciprocal 7

Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

You can see that the debugger has stopped at the breakpoint.

You can step over the call to `atoi` using the `next` command:

```
(gdb) next
9         printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
```

If you want to see what's going on inside `reciprocal`, use the `step` command like this:

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6         assert (i != 0);
```

You're now in the body of the `reciprocal` function.

You might find it more convenient to run `gdb` from within Emacs rather than using `gdb` directly from the command line. Use the command `M-x gdb` to start up `gdb` in an Emacs window. If you are stopped at a breakpoint, Emacs automatically pulls up the appropriate source file. It's easier to figure out what's going on when you're looking at the whole file rather than just one line of text.

1.5 Finding More Information

Nearly every Linux distribution comes with a great deal of useful documentation. You could learn most of what we'll talk about in this book by reading documentation in your Linux distribution (although it would probably take you much longer). The documentation isn't always well-organized, though, so the tricky part is finding what you need. Documentation is also sometimes out-of-date, so take everything that you read with a grain of salt. If the system doesn't behave the way a *man page* (manual pages) says it should, for instance, it may be that the man page is outdated.

To help you navigate, here are the most useful sources of information about advanced Linux programming.

6. Some people have commented that saying `break main` is a little bit funny because usually you want to do this only when `main` is already broken.

1.5.1 Man Pages

Linux distributions include man pages for most standard commands, system calls, and standard library functions. The man pages are divided into numbered sections; for programmers, the most important are these:

(1) User commands

(2) System calls

(3) Standard library functions

(8) System/administrative commands

The numbers denote man page sections. Linux's man pages come installed on your system; use the `man` command to access them. To look up a man page, simply invoke `man` *`name`*, where *`name`* is a command or function name. In a few cases, the same name occurs in more than one section; you can specify the section explicitly by placing the section number before the name. For example, if you type the following, you'll get the man page for the `sleep` command (in section 1 of the Linux man pages):

```
% man sleep
```

To see the man page for the `sleep` library function, use this command:

```
% man 3 sleep
```

Each man page includes a one-line summary of the command or function. The `whatis` *`name`* command displays all man pages (in all sections) for a command or function matching *`name`*. If you're not sure which command or function you want, you can perform a keyword search on the summary lines, using `man -k` *`keyword`*.

Man pages include a lot of very useful information and should be the first place you turn for help. The man page for a command describes command-line options and arguments, input and output, error codes, configuration, and the like. The man page for a system call or library function describes parameters and return values, lists error codes and side effects, and specifies which include file to use if you call the function.

1.5.2 Info

The Info documentation system contains more detailed documentation for many core components of the GNU/Linux system, plus several other programs. Info pages are hypertext documents, similar to Web pages. To launch the text-based Info browser, just type `info` in a shell window. You'll be presented with a menu of Info documents installed on your system. (Press Control+H to display the keys for navigating an Info document.)

Among the most useful Info documents are these:

- `gcc`—The gcc compiler
- `libc`—The GNU C library, including many system calls
- `gdb`—The GNU debugger

- `emacs`—The Emacs text editor
- `info`—The Info system itself

Almost all the standard Linux programming tools (including `ld`, the linker; `as`, the assembler; and `gprof`, the profiler) come with useful Info pages. You can jump directly to a particular Info document by specifying the page name on the command line:

```
% info libc
```

If you do most of your programming in Emacs, you can access the built-in Info browser by typing `M-x info` or `C-h i`.

1.5.3 Header Files

You can learn a lot about the system functions that are available and how to use them by looking at the system header files. These reside in `/usr/include` and `/usr/include/sys`. If you are getting compile errors from using a system call, for instance, take a look in the corresponding header file to verify that the function's signature is the same as what's listed in the man page.

On Linux systems, a lot of the nitty-gritty details of how the system calls work are reflected in header files in the directories `/usr/include/bits`, `/usr/include/asm`, and `/usr/include/linux`. For instance, the numerical values of signals (described in Section 3.3, "Signals," in Chapter 3, "Processes") are defined in `/usr/include/bits/signum.h`. These header files make good reading for inquiring minds. Don't include them directly in your programs, though; always use the header files in `/usr/include` or as mentioned in the man page for the function you're using.

1.5.4 Source Code

This is Open Source, right? The final arbiter of how the system works is the system source code itself, and luckily for Linux programmers, that source code is freely available. Chances are, your Linux distribution includes full source code for the entire system and all programs included with it; if not, you're entitled under the terms of the GNU General Public License to request it from the distributor. (The source code might not be installed on your disk, though. See your distribution's documentation for instructions on installing it.)

The source code for the Linux kernel itself is usually stored under `/usr/src/linux`. If this book leaves you thirsting for details of how processes, shared memory, and system devices work, you can always learn straight from the source code. Most of the system functions described in this book are implemented in the GNU C library; check your distribution's documentation for the location of the C library source code.