1

Getting Started

In this chapter, you discover what Linux is and how it relates to its inspiration, UNIX. You take a guided tour of the facilities provided by a Linux development system, and write and run your first program. Along the way, you'll be looking at

- UNIX, Linux, and GNU
- Programs and programming languages for Linux
- How to locate development resources
- Static and shared libraries
- The UNIX philosophy

An Introduction to UNIX, Linux, and GNU

In recent years Linux has become a phenomenon. Hardly a day goes by without Linux cropping up in the media in some way. We've lost count of the number of applications that have been made available on Linux and the number of organizations that have adopted it, including some government departments and city administrations. Major hardware vendors like IBM and Dell now support Linux, and major software vendors like Oracle support their software running on Linux. Linux truly has become a viable operating system, especially in the server market.

Linux owes its success to systems and applications that preceded it: UNIX and GNU software. This section looks at how Linux came to be and what its roots are.

What Is UNIX?

The UNIX operating system was originally developed at Bell Laboratories, once part of the telecommunications giant AT&T. Designed in the 1970s for Digital Equipment PDP computers, UNIX has become a very popular multiuser, multitasking operating system for a wide variety of hardware platforms, from PC workstations to multiprocessor servers and supercomputers.

A Brief History of UNIX

Strictly, UNIX is a trademark administered by The Open Group, and it refers to a computer operating system that conforms to a particular specification. This specification, known as The Single UNIX Specification, defines the names of, interfaces to, and behaviors of all mandatory UNIX operating system functions. The specification is largely a superset of an earlier series of specifications, the P1003, or POSIX (Portable Operating System Interface) specifications, developed by the IEEE (Institute of Electrical and Electronic Engineers).

Many UNIX-like systems are available commercially, such as IBM's AIX, HP's HP-UX, and Sun's Solaris. Some have been made available for free, such as FreeBSD and Linux. Only a few systems currently conform to The Open Group specification, which allows them to be marketed with the name UNIX.

In the past, compatibility among different UNIX systems has been a real problem, although POSIX was a great help in this respect. These days, by following a few simple rules it is possible to create applications that will run on all UNIX and UNIX-like systems. You can find more details on Linux and UNIX standards in Chapter 18.

UNIX Philosophy

In the following chapters we hope to convey a flavor of Linux (and therefore UNIX) programming. Although programming in C is in many ways the same whatever the platform, UNIX and Linux developers have a special view of program and system development.

The UNIX operating system, and hence Linux, encourages a certain programming style. Following are a few characteristics shared by typical UNIX programs and systems:

- **Simplicity:** Many of the most useful UNIX utilities are very simple and, as a result, small and easy to understand. KISS, "Keep It Small and Simple," is a good technique to learn. Larger, more complex systems are guaranteed to contain larger, more complex bugs, and debugging is a chore that we'd all like to avoid!
- **Focus:** It's often better to make a program perform one task well than to throw in every feature along with the kitchen sink. A program with "feature bloat" can be difficult to use and difficult to maintain. Programs with a single purpose are easier to improve as better algorithms or interfaces are developed. In UNIX, small utilities are often combined to perform more demanding tasks when the need arises, rather than trying to anticipate a user's needs in one large program.
- **Reusable Components:** Make the core of your application available as a library. Well-documented libraries with simple but flexible programming interfaces can help others to develop variations or apply the techniques to new application areas. Examples include the `dbm` database library, which is a suite of reusable functions rather than a single database management program.
- **Filters:** Many UNIX applications can be used as filters. That is, they transform their input and produce output. As you'll see, UNIX provides facilities that allow quite complex applications to be developed from other UNIX programs by combining them in novel ways. Of course, this kind of reuse is enabled by the development methods that we've previously mentioned.
- **Open File Formats:** The more successful and popular UNIX programs use configuration files and data files that are plain ASCII text or XML. If either of these is an option for your program development, it's a good choice. It enables users to use standard tools to change and search for configuration items and to develop new tools for performing new functions on the data files. A good example of this is the `ctags` source code cross-reference system, which records symbol location information as regular expressions suitable for use by searching programs.

- **Flexibility:** You can't anticipate exactly how ingeniously users will use your program. Try to be as flexible as possible in your programming. Try to avoid arbitrary limits on field sizes or number of records. If you can, write the program so that it's network-aware and able to run across a network as well as on a local machine. Never assume that you know everything that the user might want to do.

What Is Linux?

As you may already know, Linux is a freely distributed implementation of a UNIX-like kernel, the low-level core of an operating system. Because Linux takes the UNIX system as its inspiration, Linux and UNIX programs are very similar. In fact, almost all programs written for UNIX can be compiled and run on Linux. Also, some commercial applications sold for commercial versions of UNIX can run unchanged in binary form on Linux systems.

Linux was developed by Linus Torvalds at the University of Helsinki, with the help of UNIX programmers from across the Internet. It began as a hobby inspired by Andy Tanenbaum's Minix, a small UNIX-like system, but has grown to become a complete system in its own right. The intention is that the Linux kernel will not incorporate proprietary code but will contain nothing but freely distributable code.

Versions of Linux are now available for a wide variety of computer systems using many different types of CPUs, including PCs based on 32-bit and 64-bit Intel *x*86 and compatible processors; workstations and servers using Sun SPARC, IBM PowerPC, AMD Opteron, and Intel Itanium; and even some handheld PDAs and Sony's Playstations 2 and 3. If it's got a processor, someone somewhere is trying to get Linux running on it!

The GNU Project and the Free Software Foundation

Linux owes its existence to the cooperative efforts of a large number of people. The operating system kernel itself forms only a small part of a usable development system. Commercial UNIX systems traditionally come bundled with applications that provide system services and tools. For Linux systems, these additional programs have been written by many different programmers and have been freely contributed.

The Linux community (together with others) supports the concept of free software, that is, software that is free from restrictions, subject to the GNU General Public License (the name GNU stands for the recursive *GNU's Not Unix*). Although there may be a cost involved in obtaining the software, it can thereafter be used in any way desired and is usually distributed in source form.

The Free Software Foundation was set up by Richard Stallman, the author of GNU Emacs, one of the best-known text editors for UNIX and other systems. Stallman is a pioneer of the free software concept and started the GNU Project, an attempt to create an operating system and development environment that would be compatible with UNIX, but not suffer the restrictions of the proprietary UNIX name and source code. GNU may one day turn out to be very different from UNIX in the way it handles the hardware and manages running programs, but it will still support UNIX-style applications.

The GNU Project has already provided the software community with many applications that closely mimic those found on UNIX systems. All these programs, so-called GNU software, are distributed under the terms of the GNU General Public License (GPL); you can find a copy of the license at `http://www.gnu.org`. This license embodies the concept of *copyleft* (a takeoff on "copyright"). Copyleft is intended to prevent others from placing restrictions on the use of free software.

A few major examples of software from the GNU Project distributed under the GPL follow:

- GCC: The GNU Compiler Collection, containing the GNU C compiler
- G++: A C++ compiler, included as part of GCC
- GDB: A source code–level debugger
- GNU make: A version of UNIX `make`
- Bison: A parser generator compatible with UNIX `yacc`
- bash: A command shell
- GNU Emacs: A text editor and environment

Many other packages have been developed and released using free software principles and the GPL, including spreadsheets, source code control tools, compilers and interpreters, Internet tools, graphical image manipulation tools such as the Gimp, and two complete object-based environments: GNOME and KDE. We discuss GNOME and KDE in Chapters 16 and 17.

There is now so much free software available that with the addition of the Linux kernel it could be said that the goal of a creating GNU, a free UNIX-like system, has been achieved with Linux. To recognize the contribution made by GNU software, many people now refer to Linux systems in general as GNU/Linux.

You can learn more about the free software concept at `http://www.gnu.org`.

Linux Distributions

As we have already mentioned, Linux is actually just a kernel. You can obtain the sources for the kernel to compile and install it on a machine and then obtain and install many other freely distributed software programs to make a complete Linux installation. These installations are usually referred to as *Linux systems*, because they consist of much more than just the kernel. Most of the utilities come from the GNU Project of the Free Software Foundation.

As you can probably appreciate, creating a Linux system from just source code is a major undertaking. Fortunately, many people have put together ready-to-install distributions (often called *flavors*), usually downloadable or on CD-ROMs or DVDs, that contain not just the kernel but also many other programming tools and utilities. These often include an implementation of the X Window System, a graphical environment common on many UNIX systems. The distributions usually come with a setup program and additional documentation (normally all on the CD[s]) to help you install your own Linux system. Some well-known distributions, particularly on the Intel *x*86 family of processors, are Red Hat Enterprise Linux and its community-developed cousin Fedora, Novell SUSE Linux and the free openSUSE variant, Ubuntu Linux, Slackware, Gentoo, and Debian GNU/Linux. Check out the DistroWatch site at `http://distrowatch.com` for details on many more Linux distributions.

Programming Linux

Many people think that programming Linux means using C. It's true that UNIX was originally written in C and that the majority of UNIX applications are written in C, but C is not the only option available to

Linux programmers, or UNIX programmers for that matter. In the course of the book, we'll mention a couple of the alternatives.

> **In fact, the first version of UNIX was written in PDP 7 assembler language in 1969. C was conceived by Dennis Ritchie around that time, and in 1973 he and Ken Thompson rewrote essentially the entire UNIX kernel in C, quite a feat in the days when system software was written in assembly language.**

A vast range of programming languages are available for Linux systems, and many of them are free and available on CD-ROM collections or from FTP archive sites on the Internet. Here's a partial list of programming languages available to the Linux programmer:

Ada	C	C++
Eiffel	Forth	Fortran
Icon	Java	JavaScript
Lisp	Modula 2	Modula 3
Oberon	Objective C	Pascal
Perl	PostScript	Prolog
Python	Ruby	Smalltalk
PHP	Tcl/Tk	Bourne Shell

We show how you can use a Linux shell (`bash`) to develop small- to medium-sized applications in Chapter 2. For the rest of the book, we mainly concentrate on C. We direct our attention mostly toward exploring the Linux programming interfaces from the perspective of the C programmer, and we assume knowledge of the C programming language.

Linux Programs

Linux applications are represented by two special types of files: *executables* and *scripts*. Executable files are programs that can be run directly by the computer; they correspond to Windows `.exe` files. Scripts are collections of instructions for another program, an interpreter, to follow. These correspond to Windows `.bat` or `.cmd` files, or interpreted BASIC programs.

Linux doesn't require executables or scripts to have a specific filename or any extension whatsoever. File system attributes, which we discuss in Chapter 2, are used to indicate that a file is a program that may be run. In Linux, you can replace scripts with compiled programs (and vice versa) without affecting other programs or the people who call them. In fact, at the user level, there is essentially no difference between the two.

When you log in to a Linux system, you interact with a shell program (often `bash`) that runs programs in the same way that the Windows command prompt does. It finds the programs you ask for by name by

searching for a file with the same name in a given set of directories. The directories to search are stored in a shell variable, PATH, in much the same way as with Windows. The search path (to which you can add) is configured by your system administrator and will usually contain some standard places where system programs are stored. These include:

- /bin: Binaries, programs used in booting the system
- /usr/bin: User binaries, standard programs available to users
- /usr/local/bin: Local binaries, programs specific to an installation

An administrator's login, such as root, may use a PATH variable that includes directories where system administration programs are kept, such as /sbin and /usr/sbin.

Optional operating system components and third-party applications may be installed in subdirectories of /opt, and installation programs might add to your PATH variable by way of user install scripts.

> **It's not a good idea to delete directories from PATH unless you are sure that you understand what will result if you do.**

Note that Linux, like UNIX, uses the colon (:) character to separate entries in the PATH variable, rather than the semicolon (;) that MS-DOS and Windows use. (UNIX chose : first, so ask Microsoft why Windows is different, not why UNIX is different!) Here's a sample PATH variable:

```
/usr/local/bin:/bin:/usr/bin:.:/home/neil/bin:/usr/X11R6/bin
```

Here the PATH variable contains entries for the standard program locations, the current directory (.), a user's home directory, and the X Window System.

> **Remember, Linux uses a forward slash (/) to separate directory names in a filename rather than the backslash (\) of Windows. Again, UNIX got there first.**

Text Editors

To write and enter the code examples in the book, you'll need to use an editor. There are many to choose from on a typical Linux system. The vi editor is popular with many users.

Both of the authors like Emacs, so we suggest you take the time to learn some of the features of this powerful editor. Almost all Linux distributions have Emacs as an optional package you can install, or you can get it from the GNU website at http://www.gnu.org or a version for graphical environments at the XEmacs site at http://www.xemacs.org.

To learn more about Emacs, you can use its online tutorial. To do this, start the editor by running the emacs command, and then type Ctrl+H followed by t for the tutorial. Emacs also has its entire manual available. When in Emacs, type Ctrl+H and then i for information. Some versions of Emacs may have menus that you can use to access the manual and tutorial.

The C Compiler

On POSIX-compliant systems, the C compiler is called `c89`. Historically, the C compiler was simply called `cc`. Over the years, different vendors have sold UNIX-like systems with C compilers with different facilities and options, but often still called `cc`.

When the POSIX standard was prepared, it was impossible to define a standard `cc` command with which all these vendors would be compatible. Instead, the committee decided to create a new standard command for the C compiler, `c89`. When this command is present, it will always take the same options, independent of the machine.

On Linux systems that do try to implement the standards, you might find that any or all of the commands `c89`, `cc`, and `gcc` refer to the system C compiler, usually the GNU C compiler, or `gcc`. On UNIX systems, the C compiler is almost always called `cc`.

In this book, we use `gcc` because it's provided with Linux distributions and because it supports the ANSI standard syntax for C. If you ever find yourself using a UNIX system without `gcc`, we recommend that you obtain and install it. You can find it at `http://www.gnu.org`. Wherever we use `gcc` in the book, simply substitute the relevant command on your system.

Try It Out — Your First Linux C Program

In this example you start developing for Linux using C by writing, compiling, and running your first Linux program. It might as well be that most famous of all starting points, Hello World.

1. Here's the source code for the file `hello.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello World\n");
    exit(0);
}
```

2. Now compile, link, and run your program.

```
$ gcc -o hello hello.c
$ ./hello
Hello World
$
```

How It Works

You invoked the GNU C compiler (on Linux this will most likely be available as `cc` too) that translated the C source code into an executable file called `hello`. You ran the program and it printed a greeting. This is just about the simplest example there is, but if you can get this far with your system, you should be able to compile and run the remainder of the examples in the book. If this did not work for you, make sure that the C compiler is installed on your system. For example, many Linux distributions have an install option called Software Development (or something similar) that you should select to make sure the necessary packages are installed.

Because this is the first program you've run, it's a good time to point out some basics. The `hello` program will probably be in your home directory. If `PATH` doesn't include a reference to your home directory, the shell won't be able to find `hello`. Furthermore, if one of the directories in `PATH` contains another program called `hello`, that program will be executed instead. This would also happen if such a directory is mentioned in `PATH` before your home directory. To get around this potential problem, you can prefix program names with `./` (for example, `./hello`). This specifically instructs the shell to execute the program in the current directory with the given name. (The dot is an alias for the current directory.)

If you forget the `-o name` option that tells the compiler where to place the executable, the compiler will place the program in a file called `a.out` (meaning assembler output). Just remember to look for an `a.out` if you think you've compiled a program and you can't find it! In the early days of UNIX, people wanting to play games on the system often ran them as `a.out` to avoid being caught by system administrators, and some UNIX installations routinely delete all files called `a.out` every evening.

Development System Roadmap

For a Linux developer, it can be important to know a little about where tools and development resources are located. The following sections provide a brief look at some important directories and files.

Applications

Applications are usually kept in directories reserved for them. Applications supplied by the system for general use, including program development, are found in `/usr/bin`. Applications added by system administrators for a specific host computer or local network are often found in `/usr/local/bin` or `/opt`.

Administrators favor `/opt` and `/usr/local`, because they keep vendor-supplied files and later additions separate from the applications supplied by the system. Keeping files organized in this way may help when the time comes to upgrade the operating system, because only `/opt` and `/usr/local` need be preserved. We recommend that you compile your applications to run and access required files from the `/usr/local` hierarchy for system-wide applications. For development and personal applications it's best just to use a folder in your `home` directory.

Additional features and programming systems may have their own directory structures and program directories. Chief among these is the X Window System, which is commonly installed in the `/usr/X11` or `/usr/bin/X11` directory. Linux distributions typically use the X.Org Foundation version of the X Window System, based on Revision 7 (X11R7). Other UNIX-like systems may choose different versions of the X Window System installed in different locations, such as `/usr/openwin` for Sun's Open Windows provided with Solaris.

The GNU compiler system's driver program, `gcc` (which you used in the preceding programming example), is typically located in `/usr/bin` or `/usr/local/bin`, but it will run various compiler-support applications from another location. This location is specified when you compile the compiler itself and varies with the host computer type. For Linux systems, this location might be a version-specific subdirectory of `/usr/lib/gcc/`. On one of the author's machines at the time of writing it is `/usr/lib/gcc/i586-suse-linux/4.1.3`. The separate passes of the GNU C/C++ compiler, and GNU-specific header files, are stored here.

Header Files

For programming in C and other languages, you need header files to provide definitions of constants and declarations for system and library function calls. For C, these are almost always located in `/usr/include` and subdirectories thereof. You can normally find header files that depend on the particular incarnation of Linux that you are running in `/usr/include/sys` and `/usr/include/linux`.

Other programming systems will also have header files that are stored in directories that get searched automatically by the appropriate compiler. Examples include `/usr/include/X11` for the X Window System and `/usr/include/c++` for GNU C++.

You can use header files in subdirectories or nonstandard places by specifying the `-I` flag (for `include`) to the C compiler. For example,

```
$ gcc -I/usr/openwin/include fred.c
```

will direct the compiler to look in the directory `/usr/openwin/include`, as well as the standard places, for header files included in the `fred.c` program. Refer to the manual page for the C compiler (`man gcc`) for more details.

It's often convenient to use the `grep` command to search header files for particular definitions and function prototypes. Suppose you need to know the name of the `#define`s used for returning the exit status from a program. Simply change to the `/usr/include` directory and `grep` for a probable part of the name like this:

```
$ grep EXIT_ *.h
...
stdlib.h:#define        EXIT_FAILURE    1       /* Failing exit status.  */
stdlib.h:#define        EXIT_SUCCESS    0       /* Successful exit status.  */
...
$
```

Here `grep` searches all the files in the directory with a name ending in `.h` for the string `EXIT_`. In this example, it has found (among others) the definition you need in the file `stdlib.h`.

Library Files

Libraries are collections of precompiled functions that have been written to be reusable. Typically, they consist of sets of related functions to perform a common task. Examples include libraries of screen-handling functions (the `curses` and `ncurses` libraries) and database access routines (the `dbm` library). We show you some libraries in later chapters.

Standard system libraries are usually stored in `/lib` and `/usr/lib`. The C compiler (or more exactly, the linker) needs to be told which libraries to search, because by default it searches only the standard C library. This is a remnant of the days when computers were slow and CPU cycles were expensive. It's not enough to put a library in the standard directory and hope that the compiler will find it; libraries need to follow a very specific naming convention and need to be mentioned on the command line.

A library filename always starts with `lib`. Then follows the part indicating what library this is (like `c` for the C library, or `m` for the mathematical library). The last part of the name starts with a dot (`.`), and specifies the type of the library:

- `.a` for traditional, static libraries
- `.so` for shared libraries (see the following)

The libraries usually exist in both static and shared formats, as a quick `ls /usr/lib` will show. You can instruct the compiler to search a library either by giving it the full path name or by using the `-l` flag. For example,

```
$ gcc -o fred fred.c /usr/lib/libm.a
```

tells the compiler to compile file `fred.c`, call the resulting program file `fred`, and search the mathematical library in addition to the standard C library to resolve references to functions. A similar result is achieved with the following command:

```
$ gcc -o fred fred.c -lm
```

The `-lm` (no space between the `l` and the `m`) is shorthand (shorthand is much valued in UNIX circles) for the library called `libm.a` in one of the standard library directories (in this case `/usr/lib`). An additional advantage of the `-lm` notation is that the compiler will automatically choose the shared library when it exists.

Although libraries are usually found in standard places in the same way as header files, you can add to the search directories by using the `-L` (uppercase letter) flag to the compiler. For example,

```
$ gcc -o x11fred -L/usr/openwin/lib x11fred.c -lX11
```

will compile and link a program called `x11fred` using the version of the library `libX11` found in the `/usr/openwin/lib` directory.

Static Libraries

The simplest form of library is just a collection of object files kept together in a ready-to-use form. When a program needs to use a function stored in the library, it includes a header file that declares the function. The compiler and linker take care of combining the program code and the library into a single executable program. You must use the `–l` option to indicate which libraries other than the standard C runtime library are required.

Static libraries, also known as *archives*, conventionally have names that end with `.a`. Examples are `/usr/lib/libc.a` and `/usr/lib/libX11.a` for the standard C library and the X11 library, respectively.

You can create and maintain your own static libraries very easily by using the `ar` (for archive) program and compiling functions separately with `gcc -c`. Try to keep functions in separate source files as much as possible. If functions need access to common data, you can place them in the same source file and use static variables declared in that file.

Try It Out — Static Libraries

In this example, you create your own small library containing two functions and then use one of them in an example program. The functions are called `fred` and `bill` and just print greetings.

1. First, create separate source files (imaginatively called `fred.c` and `bill.c`) for each function. Here's the first:

```
#include <stdio.h>


void fred(int arg)
{
    printf("fred: we passed %d\n", arg);
}
```

And here's the second:

```
#include <stdio.h>


void bill(char *arg)
{
    printf("bill: we passed %s\n", arg);
}
```

2. You can compile these functions individually to produce object files ready for inclusion into a library. Do this by invoking the C compiler with the `-c` option, which prevents the compiler from trying to create a complete program. Trying to create a complete program would fail because you haven't defined a function called `main`.

```
$ gcc -c bill.c fred.c
$ ls *.o
bill.o  fred.o
```

3. Now write a program that calls the function `bill`. First, it's a good idea to create a header file for your library. This will declare the functions in your library and should be included by all applications that want to use your library. It's a good idea to include the header file in the files `fred.c` and `bill.c` too. This will help the compiler pick up any errors.

```
/*
    This is lib.h. It declares the functions fred and bill for users
```

```
*/


void bill(char *);
void fred(int);
```

4. The calling program (`program.c`) can be very simple. It includes the library header file and calls one of the functions from the library.

```
#include <stdlib.h>
#include "lib.h"


int main()
{
    bill("Hello World");
    exit(0);
}
```

5. You can now compile the program and test it. For now, specify the object files explicitly to the compiler, asking it to compile your file and link it with the previously compiled object module `bill.o`.

```
$ gcc -c program.c
$ gcc -o program program.o bill.o
$ ./program
bill: we passed Hello World
$
```

6. Now you'll create and use a library. Use the `ar` program to create the archive and add your object files to it. The program is called `ar` because it creates archives, or collections, of individual files placed together in one large file. Note that you can also use `ar` to create archives of files of any type. (Like many UNIX utilities, `ar` is a generic tool.)

```
$ ar crv libfoo.a bill.o fred.o
a - bill.o
a - fred.o
```

7. The library is created and the two object files added. To use the library successfully, some systems, notably those derived from Berkeley UNIX, require that a table of contents be created for the library. Do this with the `ranlib` command. In Linux, this step isn't necessary (but it is harmless) when you're using the GNU software development tools.

```
$ ranlib libfoo.a
```

Your library is now ready to use. You can add to the list of files to be used by the compiler to create your program like this:

```
$ gcc -o program program.o libfoo.a
$ ./program
bill: we passed Hello World
$
```

You could also use the `-l` option to access the library, but because it is not in any of the standard places, you have to tell the compiler where to find it by using the `-L` option like this:

```
$ gcc -o program program.o -L. -lfoo
```

The `-L.` option tells the compiler to look in the current directory (`.`) for libraries. The `-lfoo` option tells the compiler to use a library called `libfoo.a` (or a shared library, `libfoo.so`, if one is present). To see which functions are included in an object file, library, or executable program, you can use the `nm` command. If you take a look at `program` and `lib.a`, you see that the library contains both `fred` and `bill`, but that `program` contains only `bill`. When the program is created, it includes only functions from the library that it actually needs. Including the header file, which contains declarations for all of the functions in the library, doesn't cause the entire library to be included in the final program.

If you're familiar with Windows software development, there are a number of direct analogies here, illustrated in the following table.

Item	UNIX	Windows
object module	`func.o`	`FUNC.OBJ`
static library	`lib.a`	`LIB.LIB`
program	`program`	`PROGRAM.EXE`

Shared Libraries

One disadvantage of static libraries is that when you run many applications at the same time and they all use functions from the same library, you may end up with many copies of the same functions in memory and indeed many copies in the program files themselves. This can consume a large amount of valuable memory and disk space.

Many UNIX systems and Linux-support shared libraries can overcome this disadvantage. A complete discussion of shared libraries and their implementation on different systems is beyond the scope of this book, so we'll restrict ourselves to the visible implementation under Linux.

Shared libraries are stored in the same places as static libraries, but shared libraries have a different filename suffix. On a typical Linux system, the shared version of the standard math library is `/lib/libm.so`.

When a program uses a shared library, it is linked in such a way that it doesn't contain function code itself, but references to shared code that will be made available at run time. When the resulting program is loaded into memory to be executed, the function references are resolved and calls are made to the shared library, which will be loaded into memory if needed.

In this way, the system can arrange for a single copy of a shared library to be used by many applications at once and stored just once on the disk. An additional benefit is that the shared library can be updated independently of the applications that rely on it. Symbolic links from the `/lib/libm.so` file to the actual library revision (`/lib/libm.so.N` where `N` represents a major version number — 6 at the time of writing) are used. When Linux starts an application, it can take into account the version of a library required by the application to prevent major new versions of a library from breaking older applications.

> **The following example outputs are taken from a SUSE 10.3 distribution. Your output may differ slightly if you are not using this distribution.**

For Linux systems, the program (the dynamic loader) that takes care of loading shared libraries and resolving client program function references is called `ld.so` and may be made available as `ld-linux.so.2` or `ld-lsb.so.2` or `ld-lsb.so.3`. The additional locations searched for shared libraries are configured in the file `/etc/ld.so.conf`, which needs to be processed by `ldconfig` if changed (for example, if X11 shared libraries are added when the X Window System is installed).

You can see which shared libraries are required by a program by running the utility `ldd`. For example, if you try running it on your example application, you get the following:

```
$ ldd program
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7db4000)
        /lib/ld-linux.so.2 (0xb7efc000)
```

In this case, you see that the standard C library (`libc`) is shared (`.so`). The program requires major Version 6. Other UNIX systems will make similar arrangements for access to shared libraries. Refer to your system documentation for details.

In many ways, shared libraries are similar to dynamic-link libraries used under Windows. The `.so` libraries correspond to `.DLL` files and are required at run time, and the `.a` libraries are similar to `.LIB` files included in the program executable.

Getting Help

The vast majority of Linux systems are reasonably well documented with respect to the system programming interfaces and standard utilities. This is true because, since the earliest UNIX systems, programmers have been encouraged to supply a manual page with their applications. These manual pages, which are sometimes provided in a printed form, are invariably available electronically.

The `man` command provides access to the online manual pages. The pages vary considerably in quality and detail. Some may simply refer the reader to other, more thorough documentation, whereas others give a complete list of all options and commands that a utility supports. In either case, the manual page is a good place to start.

The GNU software suite and some other free software use an online documentation system called `info`. You can browse full documentation online using a special program, `info`, or via the `info` command of

the `emacs` editor. The benefit of the `info` system is that you can navigate the documentation using links and cross-references to jump directly to relevant sections. For the documentation author, the `info` system has the benefit that its files can be automatically generated from the same source as the printed, typeset documentation.

Try It Out Manual Pages and info

Let's look for documentation of the GNU C compiler (`gcc`).

1. First take a look at the manual page.

```
$ man gcc

GCC(1)                          GNU                          GCC(1)



NAME
       gcc - GNU project C and C++ compiler

SYNOPSIS
       gcc [-c|-S|-E] [-std=standard]
           [-g] [-pg] [-Olevel]
           [-Wwarn...] [-pedantic]
           [-Idir...] [-Ldir...]
           [-Dmacro[=defn]...] [-Umacro]
           [-foption...] [-mmachine-option...]
           [-o outfile] infile...

       Only the most useful options are listed here; see below
       for the remainder.  g++ accepts mostly the same options as
       gcc.

DESCRIPTION
       When you invoke GCC, it normally does preprocessing, com
       pilation, assembly and linking.  The ``overall options''
       allow you to stop this process at an intermediate stage.
       For example, the -c option says not to run the linker.
       Then the output consists of object files output by the
       assembler.

       Other options are passed on to one stage of processing.
       Some options control the preprocessor and others the com
       piler itself. Yet other options control the assembler and
       linker; most of these are not documented here, since we
       rarely need to use any of them.
...
```

If you want, you can read about the options that the compiler supports. The manual page in this case is quite long, but it forms only a small part of the total documentation for GNU C (and C++).

When reading manual pages, you can use the spacebar to read the next page, Enter (or Return if your keyboard has that key instead) to read the next line, and *q* to quit altogether.

2. To get more information on GNU C, you can try `info`.

```
$ info gcc

File: gcc.info,  Node: Top,  Next: G++ and GCC,  Up: (DIR)
Introduction
************

   This manual documents how to use the GNU compilers, as well as their
features and incompatibilities, and how to report bugs.  It corresponds
to GCC version 4.1.3.  The internals of the GNU compilers, including how
to port them to new targets and some information about how to write
front ends for new languages, are documented in a separate manual.
*Note Introduction: (gccint)Top.

* Menu:

* G++ and GCC::     You can compile C or C++ Applications.
* Standards::       Language standards supported by GCC.
* Invoking GCC::    Command options supported by `gcc'.
* C Implementation:: How GCC implements the ISO C specification.
* C Extensions::    GNU extensions to the C language family.
* C++ Extensions::  GNU extensions to the C++ language.
* Objective-C::     GNU Objective-C runtime features.
* Compatibility::   Binary Compatibility
--zz-Info: (gcc.info.gz)Top, 39 lines --Top-------------------------------
Welcome to Info version 4.8. Type ? for help, m for menu item.
```

You're presented with a long menu of options that you can select to move around a complete text version of the documentation. Menu items and a hierarchy of pages allow you to navigate a very large document. On paper, the GNU C documentation runs to many hundreds of pages.

The `info` system also contains its own help page in `info` form pages, of course. If you type Ctrl+H, you'll be presented with some help that includes a tutorial on using `info`. The `info` program is available with many Linux distributions and can be installed on other UNIX systems.

Summary

In this introductory chapter, we've looked at Linux programming and the things Linux holds in common with proprietary UNIX systems. We've noted the wide variety of programming systems available to UNIX developers. We've also presented a simple program and library to demonstrate the basic C tools, comparing them with their Windows equivalents.