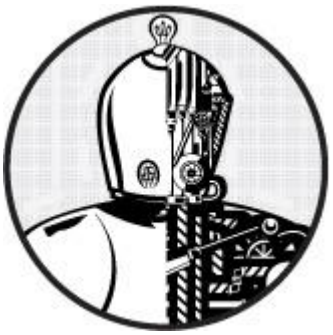
Chapter 15. Development Tools



Linux and Unix are very popular with programmers, not just due to the overwhelming array of tools and environments available but also because the system is exceptionally well documented and transparent. On a Linux machine, you don't have to be a programmer to take advantage of development tools, but when working with the system, you should know something about programming tools because they play a larger role in managing Unix systems than in other operating systems. At the very least, you should be able to identify development utilities and have some idea of how to run them.

This chapter packs a lot of information into a small space, but you don't need to master everything here. You can easily skim the material and come back later. The discussion of shared libraries is likely the most important thing that you need to know. But to understand where shared libraries come from, you first need some background on how to build programs.

15.1 The C Compiler

Knowing how to run the C programming language compiler can give you a great deal of insight into the origin of the programs that you see on your Linux system. The source code for most Linux utilities, and for many applications on Linux systems, is written in C or C++. We'll primarily use examples in C for this chapter, but you'll be able to carry the information over to C++.

C programs follow a traditional development process: You write programs, you compile them, and they run. That is, when you write a C program and want to run it, you must *compile* the source code that you wrote into a binary low-level form that the computer understands. You can compare this to the scripting languages that we'll discuss later, where you don't need to compile anything.

NOTE

By default, most distributions do not include the tools necessary to compile C code because these tools occupy a fairly large amount of space. If you can't find some of the tools described here, you can install the build-essential package for Debian/Ubuntu or the Chapter 15 `yum` groupinstall for Fedora/CentOS. Failing that, try a package search for "C compiler."

The C compiler executable on most Unix systems is the GNU C complier, `gcc`, though the newer `clang` compiler from the LLVM project is gaining popularity. C source code files end with *.c*. Take a look at the single, self-contained C source code file called *hello.c*, which you can find in *The C Programming Language*, 2nd edition, by Brian W. Kernighan and Dennis M. Ritchie (Prentice Hall, 1988):

```
#include <stdio.h>


main() {
    printf("Hello, World.\n");
```

```
}
```

Put this source code in a file called *hello.c* and then run this command:

```
$ cc hello.c
```

The result is an executable named *a.out*, which you can run like any other executable on the system. However, you should probably give the executable another name (such as *hello*). To do this, use the compiler's `-o` option:

```
$ cc -o hello hello.c
```

For small programs, there isn't much more to compiling than that. You might need to add an extra include directory or library (see 15.1.2 Header (Include) Files and Directories and 15.1.3 Linking with Libraries), but let's look at slightly larger programs before getting into those topics.

15.1.1 Multiple Source Files

Most C programs are too large to reasonably fit inside a single source code file. Mammoth files become too disorganized for the programmer, and compilers sometimes even have trouble parsing large files. Therefore, developers group components of the source code together, giving each piece its own file.

When compiling most *.c* files, you don't create an executable right away. Instead, use the compiler's `-c` option on each file to create *object files*. To see how this works, let's say you have two files, *main.c* and *aux.c*. The following two compiler commands do most of the work of building the program:

```
$ cc -c main.c
$ cc -c aux.c
```

The preceding two commands compile the two source files into the two object files *main.o* and *aux.o*.

An object file is a binary file that a processor can almost understand, except that there are still a few loose ends. First, the operating system doesn't know how to run an object file, and second, you likely need to combine several object files and some system libraries to make a complete program.

To build a fully functioning executable from one or more object files, you must run the *linker*, the `ld` command in Unix. Programmers rarely use `ld` on the command line, because the C compiler knows how to run the linker program. So to create an executable called *myprog* from the two object files above, run this command to link them:

```
$ cc -o myprog main.o aux.o
```

Although you can compile multiple source files by hand, as the preceding example shows, it can be hard to keep track of them all during the compiling process when the number of source files multiplies. The `make` system described in 15.2 make is the traditional Unix standard for managing compiles. This system is especially important in managing the files described in the next two sections.

15.1.2 Header (Include) Files and Directories

C *header files* are additional source code files that usually contain type and library function declarations. For example, *stdio.h* is a header file (see the simple program in 15.1 The C Compiler).

Unfortunately, a great number of compiler problems crop up with header files. Most glitches occur when the compiler can't find header files and libraries. There are even some cases where a programmer forgets to include a required header file, causing some of the source code to not compile.

Fixing Include File Problems

Tracking down the correct include files isn't always easy. Sometimes there are several include files with the

same names in different directories, and it's not clear which is the correct one. When the compiler can't find an include file, the error message looks like this:

```
badinclude.c:1:22: fatal error: notfound.h: No such file or directory
```

This message reports that the compiler can't find the *notfound.h* header file that the *badinclude.c* file references. This specific error is a direct result of this directive on line 1 of *badinclude.c*:

```
#include <notfound.h>
```

The default include directory in Unix is */usr/include*; the compiler always looks there unless you explicitly tell it not to. However, you can make the compiler look in other include directories (most paths that contain header files have *include* somewhere in the name).

NOTE

You'll learn more about how to find missing include files in Chapter 16.

For example, let's say that you find *notfound.h* in */usr/junk/include*. You can make the compiler see this directory with the `-I` option:

```
$ cc -c -I/usr/junk/include badinclude.c
```

Now the compiler should no longer stumble on the line of code in *badinclude.c* that references the header file.

You should also beware of includes that use double quotes (`" "`) instead of angle brackets (`< >`), like this:

```
#include "myheader.h"
```

Double quotes mean that the header file is not in a system include directory but that the compiler should otherwise search its include path. It often means that the include file is in the same directory as the source file. If you encounter a problem with double quotes, you're probably trying to compile incomplete source code.

What Is the C Preprocessor (cpp)?

It turns out that the C compiler does not actually do the work of looking for all of these include files. That task falls to the *C preprocessor*, a program that the compiler runs on your source code before parsing the actual program. The preprocessor rewrites source code into a form that the compiler understands; it's a tool for making source code easier to read (and for providing shortcuts).

Preprocessor commands in the source code are called *directives*, and they start with the `#` character. There are three basic types of directives:

- **Include files**. An `#include` directive instructs the preprocessor to include an entire file. Note that the compiler's `-I` flag is actually an option that causes the preprocessor to search a specified directory for include files, as you saw in the previous section.
- **Macro definitions**. A line such as `#define BLAH something` tells the preprocessor to substitute `something` for all occurrences of `BLAH` in the source code. Convention dictates that macros appear in all uppercase, but it should come as no shock that programmers sometimes use macros whose names look like functions and variables. (Every now and then, this causes a world of headaches. Many programmers make a sport out of abusing the preprocessor.)

NOTE

Instead of defining macros within your source code, you can also define macros by passing parameters to the compiler: `-DBLAH=something` works like the directive above.

- **Conditionals**. You can mark out certain pieces of code with `#ifdef`, `#if`, and `#endif`. The `#ifdef` *`MACRO`* directive checks to see whether the preprocessor macro *`MACRO`* is defined, and `#if` *`condition`* tests to see whether *`condition`* is nonzero. For both directives, if the condition following the "if

statement" is false, the preprocessor does not pass any of the program text between the `#if` and the next `#endif` to the compiler. If you plan to look at any C code, you'd better get used to this.

An example of a conditional directive follows. When the preprocessor sees the following code, it checks to see whether the macro `DEBUG` is defined and, if so, passes the line containing `fprintf()` on to the compiler. Otherwise, the preprocessor skips this line and continues to process the file after the `#endif`:

```
#ifdef DEBUG
  fprintf(stderr, "This is a debugging message.\n");
#endif
```

NOTE

The C preprocessor doesn't know anything about C syntax, variables, functions, and other elements. It understands only its own macros and directives.

On Unix, the C preprocessor's name is `cpp`, but you can also run it with `gcc -E`. However, you'll rarely need to run the preprocessor by itself.

15.1.3 Linking with Libraries

The C compiler doesn't know enough about your system to create a useful program all by itself. You need *libraries* to build complete programs. A C library is a collection of common precompiled functions that you can build into your program. For example, many executables use the math library because it provides trigonometric functions and the like.

Libraries come into play primarily at link time, when the linker program creates an executable from object files. For example, if you have a program that uses the gobject library but you forget to tell the compiler to link against that library, you'll see linker errors like this:

```
badobject.o(.text+0x28): undefined reference to 'g_object_new'
```

The most important parts of these error messages are in bold. When the linker program examined the *badobject.o* object file, it couldn't find the function that appears in bold, and as a consequence, it couldn't create the executable. In this particular case, you might suspect that you forgot the gobject library because the missing function is `g_object_new()`.

NOTE

Undefined references do not always mean that you're missing a library. One of the program's object files could be missing in the link command. It's usually easy to differentiate between library functions and functions in your object files, though.

To fix this problem, you must first find the gobject library and then use the compiler's `-l` option to link against the library. As with include files, libraries are scattered throughout the system (*/usr/lib* is the system default location), though most libraries reside in a subdirectory named *lib*. For the preceding example, the basic gobject library file is *libgobject.a*, so the library name is `gobject`. Putting it all together, you would link the program like this:

```
$ cc -o badobject badobject.o -lgobject
```

You must tell the linker about nonstandard library locations; the parameter for this is `-L`. Let's say that the `badobject` program requires *libcrud.a* in */usr/junk/lib*. To compile and create the executable, use a command like this:

```
$ cc -o badobject badobject.o -lgobject -L/usr/junk/lib -lcrud
```

NOTE

If you want to search a library for a particular function, use the `nm` *command. Be prepared for a lot of output. For example, try this:* **`nm libgobject.a`**. *(You might need to use the* `locate` *command to find* libgobject.a*; many distributions now put libraries in architecture-specific subdirectories in* /usr/lib.*)*

15.1.4 Shared Libraries

A library file ending with *.a* (such as *libgobject.a*) is called a *static library*. When you link a program against a static library, the linker copies machine code from the library file into your executable. Therefore, the final executable does not need the original library file to run, and furthermore, the executable's behavior never changes.

However, library sizes are always increasing, as is the number of libraries in use, and this makes static libraries wasteful in terms of disk space and memory. In addition, if a static library is later found to be inadequate or insecure, there's no way to change any executable linked against it, short of recompiling the executable.

Shared libraries counter these problems. When you run a program linked against one, the system loads the library's code into the process memory space only when necessary. Many processes can share the same shared library code in memory. And if you need to slightly modify the library code, you can generally do so without recompiling any programs.

Shared libraries have their own costs: difficult management and a somewhat complicated linking procedure. However, you can bring shared libraries under control if you know four things:

- How to list the shared libraries that an executable needs
- How an executable looks for shared libraries
- How to link a program against a shared library
- The common shared library pitfalls

The following sections tell you how to use and maintain your system's shared libraries. If you're interested in how shared libraries work or if you want to know about linkers in general, you can check out *Linkers and Loaders* by John R. Levine (Morgan Kaufmann, 1999), "The Inside Story on Shared Libraries and Dynamic Loading" by David M. Beazley, Brian D. Ward, and Ian R. Cooke (*Computing in Science & Engineering*, September/October 2001), or online resources such as the Program Library HOWTO (*http://dwheeler.com/program-library/*). The ld.so(8) manual page is also worth a read.

Listing Shared Library Dependencies

Shared library files usually reside in the same places as static libraries. The two standard library directories on a Linux system are */lib* and */usr/lib*. The */lib* directory should not contain static libraries.

A shared library has a suffix that contains *.so* (shared object), as in *libc-2.15.so* and *libc.so.6*. To see what shared libraries a program uses, run `ldd` *`prog`*, where *`prog`* is the executable name. Here's an example for the shell:

```
$ ldd /bin/bash
    linux-gate.so.1 => (0xb7799000)
    libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb7765000)
    libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7760000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b5000)
```

```
        /lib/ld-linux.so.2 (0xb779a000)
```

In the interest of optimal performance and flexibility, executables alone don't usually know the locations of their shared libraries; they know only the names of the libraries, and perhaps a little hint about where to find them. A small program named `ld.so` (the *runtime dynamic linker/loader*) finds and loads shared libraries for a program at runtime. The preceding `ldd` output shows the library names on the left—that's what the executable knows. The right side shows where `ld.so` finds the library.

The final line of output here shows the actual location of `ld.so`: *ld-linux.so.2*.

How ld.so Finds Shared Libraries

One of the common trouble points for shared libraries is that the dynamic linker cannot find a library. The first place the dynamic linker *should* normally look for shared libraries is an executable's preconfigured *runtime library search path* (*rpath*), if it exists. You'll see how to create this path shortly.

Next, the dynamic linker looks in a system cache, */etc/ld.so.cache*, to see if the library is in a standard location. This is a fast cache of the names of library files found in directories listed in the cache configuration file */etc/ld.so.conf*.

NOTE

As is typical of many of the Linux configuration files that you've seen, ld.so.conf *may include a number of files in a directory such as* /etc/ld.so.conf.d.

Each line in */ld.so.conf* is a directory that you want to include in the cache. The list of directories is usually short, containing something like this:

```
/lib/i686-linux-gnu

/usr/lib/i686-linux-gnu
```

The standard library directories */lib* and */usr/lib* are implicit, which means that you don't need to include them in */etc/ld.so.conf*.

If you alter */ld.so.conf* or make a change to one of the shared library directories, you must rebuild the */etc/ld.so.cache* file by hand with the following command:

```
# ldconfig -v
```

The `-v` option provides detailed information on libraries that `ldconfig` adds to the cache and any changes that it detects.

There is one more place that `ld.so` looks for shared libraries: the environment variable `LD_LIBRARY_PATH`. We'll talk about this soon.

Don't get into the habit of adding stuff to */etc/ld.so.conf*. You should know what shared libraries are in the system cache, and if you put every bizarre little shared library directory into the cache, you risk conflicts and an extremely disorganized system. When you compile software that needs an obscure library path, give your executable a built-in runtime library search path. Let's see how to do that.

Linking Programs Against Shared Libraries

Let's say you have a shared library named *libweird.so.1* in */opt/obscure/lib* that you need to link `myprog` against. Link the program as follows:

```
$ cc -o myprog myprog.o -Wl,-rpath=/opt/obscure/lib -L/opt/obscure/lib
-lweird
```

The `-Wl,-rpath` option tells the linker to include a following directory into the executable's runtime library

search path. However, even if you use `-Wl,-rpath`, you still need the `-L` flag.

If you have a pre-existing binary, you can also use the `patchelf` program to insert a different runtime library search path, but it's generally better to do this at compile time.

Problems with Shared Libraries

Shared libraries provide remarkable flexibility, not to mention some really incredible hacks, but it's also possible to abuse them to the point where your system becomes an utter and complete mess. Three particularly bad things can happen:

- Missing libraries
- Terrible performance
- Mismatched libraries

The number one cause of all shared library problems is the environment variable named `LD_LIBRARY_PATH`. Setting this variable to a colon-delimited set of directory names makes `ld.so` search the given directories *before* anything else when looking for a shared library. This is a cheap way to make programs work when you move a library around, if you don't have the program's source code and can't use `patchelf`, or if you're just too lazy to recompile the executables. Unfortunately, you get what you pay for.

Never set `LD_LIBRARY_PATH` in shell startup files or when compiling software. When the dynamic runtime linker encounters this variable, it must often search through the entire contents of each specified directory more times than you'd care to know. This causes a big performance hit, but more importantly, you can get conflicts and mismatched libraries because the runtime linker looks in these directories for *every* program.

If you *must* use `LD_LIBRARY_PATH` to run some crummy program for which you don't have the source (or an application that you'd rather not compile, like Mozilla or some other beast), use a wrapper script. Let's say your executable is */opt/crummy/bin/crummy.bin* and needs some shared libraries in */opt/crummy/lib*. Write a wrapper script called `crummy` that looks like this:

```
#!/bin/sh

LD_LIBRARY_PATH=/opt/crummy/lib

export LD_LIBRARY_PATH

exec /opt/crummy/bin/crummy.bin $@
```

Avoiding `LD_LIBRARY_PATH` prevents most shared library problems. But one other significant problem that occasionally comes up with developers is that a library's application programming interface (API) may change slightly from one minor version to another, breaking installed software. The best solutions here are preventive: Either use a consistent methodology to install shared libraries with `-Wl,-rpath` to create a runtime link path or simply use the static versions of obscure libraries.

15.2 make

A program with more than one source code file or that requires strange compiler options is too cumbersome to compile by hand. This problem has been around for years, and the traditional Unix compile management utility that eases these pains is called `make`. You should know a little about `make` if you're running a Unix system, because system utilities sometimes rely on `make` to operate. However, this chapter is only the tip of the iceberg. There are entire books on `make`, such as *Managing Projects with GNU Make* by Robert Mecklenburg (O'Reilly, 2004). In addition, most Linux packages are built using an additional layer around `make` or a similar tool. There are many build systems out there; we'll look at one named autotools in Chapter 16. `make` is a big system, but it's not very difficult to get an idea of how it works. When you see a

file named *Makefile* or *makefile*, you know that you're dealing with `make`. (Try running `make` to see if you can build anything.)

The basic idea behind `make` is the *target*, a goal that you want to achieve. A target can be a file (a *.o* file, an executable, and so on) or a label. In addition, some targets depend on other targets; for instance, you need a complete set of *.o* files before you can link your executable. These requirements are called *dependencies*.

To build a target, `make` follows a *rule*, such as a rule for how to go from a *.c* source file to a *.o* object file. `make` already knows several rules, but you can customize these existing rules and create your own.

15.2.1 A Sample Makefile

The following very simple Makefile builds a program called `myprog` from *aux.c* and *main.c*:

```
# object files
OBJS=aux.o main.o


all: myprog


myprog: $(OBJS)
        $(CC) -o myprog $(OBJS)
```

The `#` in the first line of this Makefile denotes a comment.

The next line is just a macro definition; it sets the `OBJS` variable to two object filenames. This will be important later. For now, take note of how you define the macro and also how you reference it later (`$(OBJS)`).

The next item in the Makefile contains its first target, `all`. The first target is always the default, the target that `make` wants to build when you run `make` by itself on the command line.

The rule for building a target comes after the colon. For `all`, this Makefile says that you need to satisfy something called `myprog`. This is the first dependency in the file; `all` depends on `myprog`. Note that `myprog` can be an actual file or the target of another rule. In this case, it's both (the rule for `all` and the target of `OBJS`).

To build `myprog`, this Makefile uses the macro `$(OBJS)` in the dependencies. The macro expands to *aux.o* and *main.o*, so `myprog` depends on these two files (they must be actual files, because there aren't any targets with those names anywhere in the Makefile).

This Makefile assumes that you have two C source files named *aux.c* and *main.c* in the same directory. Running `make` on the Makefile yields the following output, showing the commands that `make` is running:

```
$ make
cc       -c -o aux.o aux.c
cc       -c -o main.o main.c
cc -o myprog aux.o main.o
```

A diagram of the dependencies is shown in Figure 15-1.

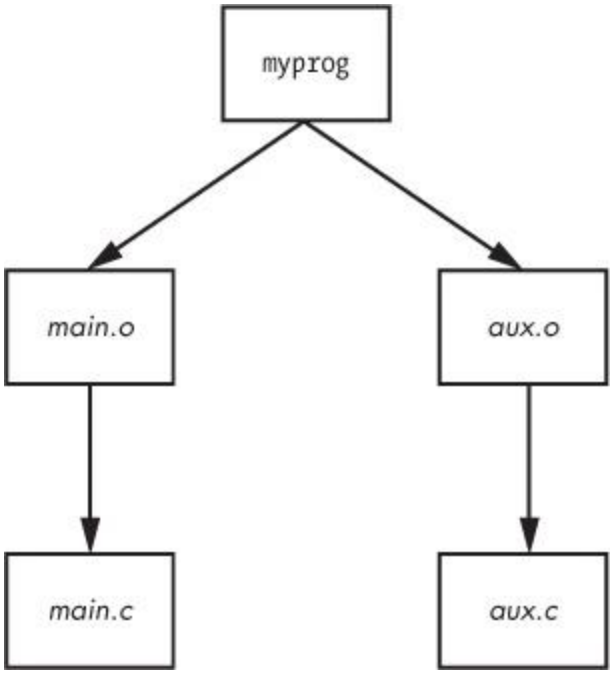


Figure 15-1. Makefile dependencies

15.2.2 Built-in Rules

So how does `make` know how to go from *aux.c* to *aux.o*? After all, *aux.c* is not in the Makefile. The answer is that `make` follows its built-in rules. It knows to look for a *.c* file when you want a *.o* file, and furthermore, it knows how to run `cc -c` on that *.c* file to get to its goal of creating a *.o* file.

15.2.3 Final Program Build

The final step in getting to `myprog` is a little tricky, but the idea is clear enough. After you have the two object files in `$(OBJS)`, you can run the C compiler according to the following line (where `$(CC)` expands to the compiler name):

```
$(CC) -o myprog $(OBJS)
```

The whitespace before `$(CC)` is a tab. You *must* insert a tab before any real command, on its own line.

Watch out for this:

```
Makefile:7: *** missing separator. Stop.
```

An error like this means that the Makefile is broken. The tab is the separator, and if there is no separator or there's some other interference, you'll see this error.

15.2.4 Staying Up-to-Date

One last `make` fundamental is that targets should be up-to-date with their dependencies. If you type `make` twice in a row for the preceding example, the first command builds `myprog`, but the second yields this output:

```
make: Nothing to be done for 'all'.
```

This second time through, `make` looked at its rules and noticed that `myprog` already exists, so it didn't build `myprog` again because none of the dependencies had changed since it was last built. To experiment with this, do the following:

1. Run **`touch aux.c`**.
2. Run **`make`** again. This time, `make` determines that *aux.c* is newer than the *aux.o* already in the directory, so it compiles *aux.o* again.

3. `myprog` depends on *aux.o*, and now *aux.o* is newer than the preexisting `myprog`, so `make` must create `myprog` again.

This type of chain reaction is very typical.

15.2.5 Command-Line Arguments and Options

You can get a great deal of mileage out of `make` if you know how its command-line arguments and options work.

One of the most useful options is to specify a single target on the command line. For the preceding Makefile, you can run `make aux.o` if you want only the *aux.o* file.

You can also define a macro on the command line. For example, to use the `clang` compiler, try

```
$ make CC=clang
```

Here, `make` uses your definition of `CC` instead of its default compiler, `cc`. Command-line macros come in handy when testing preprocessor definitions and libraries, especially with the `CFLAGS` and `LDFLAGS` macros that we'll discuss shortly.

In fact, you don't even need a Makefile to run `make`. If built-in `make` rules match a target, you can just ask `make` to try to create the target. For example, if you have the source to a very simple program called *blah.c*, try `make blah`. The `make` run proceeds like this:

```
$ make blah
cc   blah.o -o blah
```

This use of `make` works only for the most elementary C programs; if your program needs a library or special include directory, you should probably write a Makefile. Running `make` without a Makefile is actually most useful when you're dealing with something like Fortran, Lex, or Yacc and don't know how the compiler or utility works. Why not let `make` try to figure it out for you? Even if `make` fails to create the target, it will probably still give you a pretty good hint as to how to use the tool.

Two `make` options stand out from the rest:

- **`-n`** Prints the commands necessary for a build but prevents `make` from actually running any commands
- **`-f file`** Tells `make` to read from *file* instead of *Makefile* or *makefile*

15.2.6 Standard Macros and Variables

`make` has many special macros and variables. It's difficult to tell the difference between a macro and a variable, so we'll use the term *macro* to mean something that usually doesn't change after `make` starts building targets.

As you saw earlier, you can set macros at the start of your Makefile. These are the most common macros:

- **`CFLAGS`** C compiler options. When creating object code from a *.c* file, `make` passes this as an argument to the compiler.
- **`LDFLAGS`** Like `CFLAGS`, but they're for the linker when creating an executable from object code.
- **`LDLIBS`** If you use `LDFLAGS` but do not want to combine the library name options with the search path, put the library name options in this file.
- **`CC`** The C compiler. The default is `cc`.
- **`CPPFLAGS`** C *preprocessor* options. When `make` runs the C preprocessor in some way, it passes this macro's expansion on as an argument.
- **`CXXFLAGS`** GNU `make` uses this for C++ compiler flags.

A `make` *variable* changes as you build targets. Because you never set `make` variables by hand, the following list includes the `$`.

- **`$@`** When inside a rule, this expands to the current target.
- **`$*`** Expands to the *basename* of the current target. For example, if you're building *blah.o*, this expands to *blah*.

The most comprehensive list of `make` variables on Linux is the `make` info manual.

NOTE

Keep in mind that GNU `make` has many extensions, built-in rules, and features that other variants do not have. This is fine as long as you're running Linux, but if you step off onto a Solaris or BSD machine and expect the same stuff to work, you might be in for a surprise. However, that's the problem that multi-platform build systems such as GNU autotools solve.

15.2.7 Conventional Targets

Most Makefiles contain several standard targets that perform auxiliary tasks related to compiles.

- **`clean`** The `clean` target is ubiquitous; a `make clean` usually instructs `make` to remove all of the object files and executables so that you can make a fresh start or pack up the software. Here's an example rule for the `myprog` Makefile:
  - ```
    clean:
            rm -f $(OBJS) myprog
    ```
- **`distclean`** A Makefile created by way of the GNU autotools system always has a `distclean` target to remove everything that wasn't part of the original distribution, including the Makefile. You'll see more of this in Chapter 16. On very rare occasions, you might find that a developer opts not to remove the executable with this target, preferring something like `realclean` instead.
- **`install`** Copies files and compiled programs to what the Makefile thinks is the proper place on the system. This can be dangerous, so always run a `make -n install` first to see what will happen without actually running any commands.
- **`test` or `check`** Some developers provide `test` or `check` targets to make sure that everything works after performing a build.
- **`depend`** Creates dependencies by calling the compiler with `-M` to examine the source code. This is an unusual-looking target because it often changes the Makefile itself. This is no longer common practice, but if you come across some instructions telling you to use this rule, make sure to do so.
- **`all`** Often the first target in the Makefile. You'll often see references to this target instead of an actual executable.

15.2.8 Organizing a Makefile

Even though there are many different Makefile styles, most programmers adhere to some general rules of thumb. For one, in the first part of the Makefile (inside the macro definitions), you should see libraries and includes grouped according to package:

```
MYPACKAGE_INCLUDES=-I/usr/local/include/mypackage

MYPACKAGE_LIB=-L/usr/local/lib/mypackage -lmypackage

PNG_INCLUDES=-I/usr/local/include

PNG_LIB=-L/usr/local/lib -lpng
```
```

Each type of compiler and linker flag often gets a macro like this:

```
CFLAGS=$(CFLAGS) $(MYPACKAGE_INCLUDES) $(PNG_INCLUDES)

LDFLAGS=$(LDFLAGS) $(MYPACKAGE_LIB) $(PNG_LIB)
```

Object files are usually grouped according to executables. For example, say you have a package that creates executables called `boring` and `trite`. Each has its own *.c* source file and requires the code in *util.c*. You might see something like this:

```
UTIL_OBJS=util.o


BORING_OBJS=$(UTIL_OBJS) boring.o

TRITE_OBJS=$(UTIL_OBJS) trite.o


PROGS=boring trite
```

The rest of the Makefile might look like this:

```
all: $(PROGS)


boring: $(BORING_OBJS)

        $(CC) -o $@ $(BORING_OBJS) $(LDFLAGS)

trite: $(TRITE_OBJS)

       $(CC) -o $@ $(TRITE_OBJS) $(LDFLAGS)
```

You could combine the two executable targets into one rule, but it's usually not a good idea to do so because you would not easily be able to move a rule to another Makefile, delete an executable, or group executables differently. Furthermore, the dependencies would be incorrect: If you had just one rule for `boring` and `trite`, `trite` would depend on *boring.c*, `boring` would depend on *trite.c*, and `make` would always try to rebuild both programs whenever you changed one of the two source files.

NOTE

If you need to define a special rule for an object file, put the rule for the object file just above the rule that builds the executable. If several executables use the same object file, put the object rule above all of the executable rules.

15.3 Debuggers

The standard debugger on Linux systems is `gdb`; user-friendly frontends such as the Eclipse IDE and Emacs systems are also available. To enable full debugging in your programs, run the compiler with `-g` to write a symbol table and other debugging information into the executable. To start `gdb` on an executable named *`program`*, run

```
$ gdb program
```

You should get a `(gdb)` prompt. To run *`program`* with the command-line argument *`options`*, enter this at the `(gdb)` prompt:

```
(gdb) run options
```

If the program works, it should start, run, and exit as normal. However, if there's a problem, `gdb` stops, prints the failed source code, and throws you back to the `(gdb)` prompt. Because the source code fragment often hints at the problem, you'll probably want to print the value of a particular variable that the trouble may be related to. (The `print` command also works for arrays and C structures.)

```
(gdb) print variable
```

To make `gdb` stop the program at any point in the original source code, use the breakpoint feature. In the following command, *`file`* is a source code file, and *`line_num`* is the line number in that file where `gdb` should stop:

```
(gdb) break file:line_num
```

To tell `gdb` to continue executing the program, use

```
(gdb) continue
```

To clear a breakpoint, enter

```
(gdb) clear file:line_num
```

This section has provided only the briefest introduction to `gdb`, which includes an extensive manual that you can read online, in print, or buy as *Debugging with GDB*, 10th edition, by Richard M. Stallman et al. (GNU Press, 2011). *The Art of Debugging* by Norman Matloff and Peter Jay Salzman (No Starch Press, 2008) is another guide to debugging.

NOTE

If you're interested in rooting out memory problems and running profiling tests, try Valgrind (http://valgrind.org/*).*

15.4 Lex and Yacc

You might encounter Lex and Yacc when compiling programs that read configuration files or commands. These tools are building blocks for programming languages.

- Lex is a *tokenizer* that transforms text into numbered tags with labels. The GNU/Linux version is named `flex`. You may need a `-ll` or `-lfl` linker flag in conjunction with Lex.
- Yacc is a *parser* that attempts to read tokens according to a *grammar*. The GNU parser is `bison`; to get Yacc compatibility, run `bison -y`. You may need the `-ly` linker flag.

15.5 Scripting Languages

A long time ago, the average Unix systems manager didn't have to worry much about scripting languages other than the Bourne shell and awk. Shell scripts (discussed in Chapter 11) continue to be an important part of Unix, but awk has faded somewhat from the scripting arena. However, many powerful successors have arrived, and many systems programs have actually switched from C to scripting languages (such as the sensible version of the `whois` program). Let's look at some scripting basics.

The first thing you need to know about any scripting language is that the first line of a script looks like the shebang of a Bourne shell script. For example, a Python script starts out like this:

```
#!/usr/bin/python
```

Or this:

```
#!/usr/bin/env python
```

In Unix, *any* executable text file that starts with `#!` is a script. The pathname following this prefix is the scripting language interpreter executable. When Unix tries to run an executable file that starts with a `#!` shebang, it runs the program following the `#!` with the rest of the file as the standard input. Therefore, even this is a script:

```
#!/usr/bin/tail -2

This program won't print this line,

but it will print this line...

and this line, too.
```

The first line of a shell script often contains one of the most common basic script problems: an invalid path to the scripting language interpreter. For example, say you named the previous script `myscript`. What if `tail` were actually in */bin* instead of */usr/bin* on your system? In that case, running `myscript` would produce this error:

```
bash: ./myscript: /usr/bin/tail: bad interpreter: No such file or
directory
```

Don't expect more than one argument in the script's first line to work. That is, the `-2` in the preceding example might work, but if you add another argument, the system could decide to treat the `-2` *and* the new argument as one big argument, spaces and all. This can vary from system to system; don't test your patience on something as insignificant as this.

Now, let's look at a few of the languages out there.

15.5.1 Python

Python is a scripting language with a strong following and an array of powerful features, such as text processing, database access, networking, and multithreading. It has a powerful interactive mode and a very organized object model.

Python's executable is `python`, and it's usually in */usr/bin*. However, Python isn't used just from the command line for scripts. One place you'll find it is as a tool to build websites. *Python Essential Reference*, 4th edition, by David M. Beazley (Addison-Wesley, 2009) is a great reference with a small tutorial at the beginning to get you started.

15.5.2 Perl

One of the older third-party Unix scripting languages is Perl. It's the original "Swiss army chainsaw" of programming tools. Although Perl has lost a fair amount of ground to Python in recent years, it excels in particular at text processing, conversion, and file manipulation, and you may find many tools built with it. *Learning Perl*, 6th edition, by Randal L. Schwartz, brian d foy, and Tom Phoenix(O'Reilly, 2011) is a tutorial-style introduction; a larger reference is *Modern Perl* by Chromatic (Onyx Neon Press, 2014).

15.5.3 Other Scripting Languages

You might also encounter these scripting languages:

- **PHP**. This is a hypertext-processing language often found in dynamic web scripts. Some people use PHP for standalone scripts. The PHP website is at *http://www.php.net/*.
- **Ruby**. Object-oriented fanatics and many web developers enjoy programming in this language (*http://www.ruby-lang.org/*).

- **JavaScript**. This language is used inside web browsers primarily to manipulate dynamic content. Most experienced programmers shun it as a standalone scripting language due to its many flaws, but it's nearly impossible to avoid when doing web programming. You might find an implementation called Node.js with an executable name of `node` on your system.
- **Emacs Lisp**. A variety of the Lisp programming language used by the Emacs text editor.
- **Matlab, Octave**. Matlab is a commercial matrix and mathematical programming language and library. There is a very similar free software project called Octave.
- **R**. A popular free statistical analysis language. See *http://www.r-project.org/* and *The Art of R Programming* by Norman Matloff (No Starch Press, 2011) for more information.
- **Mathematica**. Another commercial mathematical programming language with libraries. **m4** This is a macro-processing language, usually found only in the GNU autotools.
- **Tcl**. Tcl (tool command language) is a simple scripting language usually associated with the Tk graphical user interface toolkit and Expect, an automation utility. Although Tcl does not enjoy the widespread use that it once did, don't discount its power. Many veteran developers prefer Tk, especially for its embedded capabilities. See *http://www.tcl.tk/* for more on Tk.

15.6 Java

Java is a compiled language like C, with a simpler syntax and powerful support for object-oriented programming. It has a few niches in Unix systems. For one, it's often used as a web application environment, and it's popular for specialized applications. For example, Android applications are usually written in Java. Even though it's not often seen on a typical Linux desktop, you should know how Java works, at least for standalone applications.

There are two kinds of Java compilers: native compilers for producing machine code for your system (like a C compiler) and bytecode compilers for use by a bytecode interpreter (sometimes called a *virtual machine*, which is different from the virtual machine offered by a hypervisor, as described in Chapter 17). You'll practically always encounter bytecode on Linux.

Java bytecode files end in *.class*. The Java runtime environment (JRE) contains all of the programs you need to run Java bytecode. To run a bytecode file, use

```
$ java file.class
```

You might also encounter bytecode files that end in *.jar*, which are collections of archived *.class* files. To run a *.jar* file, use this syntax:

```
$ java -jar file.jar
```

Sometimes you need to set the `JAVA_HOME` environment variable to your Java installation prefix. If you're really unlucky, you might need to use `CLASSPATH` to include any directories containing classes that your program expects. This is a colon-delimited set of directories like the regular `PATH` variable for executables.

If you need to compile a *.java* file into bytecode, you need the Java Development Kit (JDK). You can run the `javac` compiler from JDK to create some *.class* files:

```
$ javac file.java
```

JDK also comes with `jar`, a program that can create and pick apart *.jar* files. It works like `tar`.