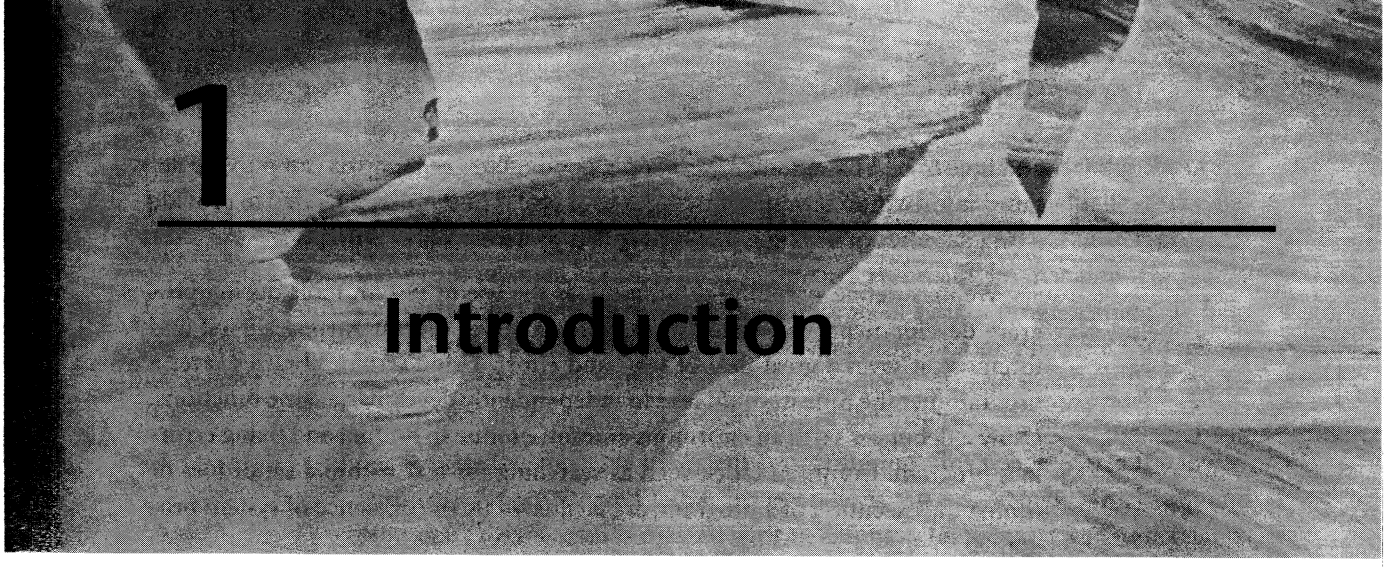

# 1 Introduction

## 1.1 • What is System Programming?

Computer systems are made up of hardware and software. Software generically refers to the programs running on the computer. While both hardware and software can be modified or upgraded, it tends to happen more often with software. In fact, the major reason to have software is to provide the ability to change the instruction stream executed on the computer. This means that it is often *expected* that new programs will be written, or old programs will be modified or evolved, during the life cycle of a computer system.

Given this expectation, it is natural to look for methods to allow the computer system to support program development. A number of tools and resources have evolved over the last 30 years to assist program development. These include standard libraries (also called system libraries), system calls, debuggers, the shell environment, system programs, and scripting languages. Knowledge of these tools greatly enhances the ability of a programmer. While it is important to learn the details of some specific tools, it is more important to understand when

and how to best use a tool. A number of good reference manuals can provide detailed information on a specific language or tool. This book is intended to provide a broader understanding of the concepts in system programming.

We may define system programming as the use of system tools during program development. Proper use of these tools serves several purposes. First and foremost, it saves a great deal of time and effort. Using system libraries saves a programmer the time it would take to independently develop the same functions. Using a debugger saves an enormous amount of time in finding and fixing errors in a program. Common tasks, such as searching for text within a set of files or timing the execution of a program, are facilitated by the existence of system programs.

Second, system tools provide opportunities for program development that are otherwise extremely difficult to come by. System calls provide access to the core functions of the operating system, including memory management, file access, process management, and interprocess communication. Some standard libraries implement complex functions that are beyond the capability of most programmers. For example, the math library includes trigonometric functions and other real-valued operations that require iterative methods to reach a solution.

Third, consistent use of system tools promotes standards, so that code developed for one computer system is more easily ported to another computer system. System libraries provide a layer of abstraction, implementing the same function calls on multiple computing systems. An application can call a system library without worrying about the details of the underlying hardware. In this manner, the application can be ported as long as the destination system possesses the same system libraries. For graphics, this has become increasingly important as the number and variety of hardware display capabilities has expanded.

Knowledge of the basic system file structure assists in program management. A Unix computer system typically includes well over 10,000 files related to system operation (this does not include user data files). Over time, a standard method for organizing these files has evolved. There are common places for libraries, system programs, device files (connections to hardware), applications, and user data.

Finally, there is the shell environment. The shell environment is rich with options, capabilities, and configurability, to the point that it is overwhelming to novice programmers. However, once some proficiency has been gained, the shell is a powerful tool for any serious system programmer. It offers tremendous flexibility in process control, system management, and program development.

This text was written with three goals. First, it supports teaching about the tools and concepts of system programming. Second, it should help the reader elevate his or her programming skill beyond an introductory level. Third, it provides a rigorous regimen of programming exercises and examples that allow the reader to practice and develop the skills and concepts of system programming. To help achieve these goals, example code pieces and programs are provided throughout the text. Each chapter ends with numerous questions and exercises that can be undertaken to strengthen understanding of the material.

Besides the concepts of system programming, this text explores the lower-level data types: bits and bytes, bit operations, arrays, strings, structures, and pointers. This material is covered with an emphasis on memory and understanding how and why these different data types are used. It is common for a student to be less comfortable with these topics than with other basic programming concepts, such as loops and conditionals. The coverage of the lower-level data types is intended to reinforce an earlier exposure to these topics. The goal is to advance the programming skill of the reader to the point where these lower-level data types are well and comfortably used.

### 1.1.1 Required Background

This text assumes that the reader has a basic understanding of programming, such as variables, loops, conditionals, and control flow. For example, the reader may have completed a single semester of study covering an introduction to C programming. If the reader studied a different language, such as Java or C++, then Section 1.5 can be studied to provide background on the equivalent C syntax. The text also assumes that the reader has a working computer system, with a C compiler, text editor, shell, and debugger already installed. The reader is assumed to be familiar with the basic operation of the computer system, such as navigating a directory or folder hierarchy and executing programs. Lastly, some shell knowledge is also assumed, such as that obtained from an introductory programming course.

### 1.1.2 Why Unix?

The majority of the material presented in this text can be studied on any computer system, using any operating system. However, it would be naive not to recognize the two most prevalent operating systems at the time of this writing:

Microsoft Windows™ and Unix.[1] For reasons about to be explained, this text advocates the study of system programming concepts on a Unix system. Note that this discussion centers on which is to be preferred for *study*. There are other ongoing debates as to which operating system has the better business model, better development, and other issues. The interested reader is directed to seek other sources for discussion on these debates.

The Windows operating system is designed to simplify computer usage. A Windows computer is essentially a closed system. The system is designed to be turnkey, in line with the business model of providing the typical user with a system that is as easy to use as possible. This design strategy necessarily obstructs getting "under the hood," to keep the typical user from doing something harmful to the system. In addition, Microsoft publishes limited information on the internal workings and design of Windows. This is also a business policy to protect their design from competitors. Finally, Windows is monolithic, not allowing for various parts of the system to be disconnected or swapped for alternatives. Once again, this is a business decision; Microsoft wants to sell its products and only its products, so it makes its system fully integrated. It is straightforward to understand the business motivations for the closed design, scarcity of published details, and non-modularity of Windows. However, the very properties that aid the typical computer user can be frustrating for the student studying system operation.

Unix, on the other hand, and specifically Linux, has different design principles. It is open source, so that all details of its inner workings can be studied. It is completely modular, so that any system component can be swapped for an alternative. For example, the Linux kernel is developed completely independently of the desktop environment. Within the kernel, a Linux computer operator has many choices as to how to configure the operation of the system. The kernel itself can be swapped or modified. One could argue that these properties prevent the more widespread adoption of Unix by typical computer users, who are not interested in this flexibility and openness. However, it is these design properties that make Unix an attractive choice for a student studying system operation.

Throughout this text, the examples shown are taken from a computer running the Linux operating system. For the most part, these examples can be run just as easily on a Windows or other Unix system. There are some important design issues that differentiate Unix/Linux and Windows, such as the multiuser versus single-user nature of the systems. These differences will be discussed in detail at the appropriate places. Beyond these considerations, deep down under the hood, the system concepts are largely similar.

1. In this book, Unix refers to any Unix-like system, including Linus, Mac OS X, BSD variants, etc.

### 1.1.3 Why C?

Selection of a particular programming language is an old debate in computing. For application development, the debate still rages. For system programming, however, very few experts argue for a language other than C. The reason is simple: C is closest to the hardware. All programming languages provide various levels of abstraction to assist in program development. For example, the concept of a named variable, as opposed to a numeric memory address, tremendously simplifies program development. Out of all the commonly used programming languages, C provides for the least abstraction and hence is closest to the hardware. Most single C statements translate simply to machine code. The available data types in C tend to reflect what the hardware directly supports. Accessing memory via indirection (pointers) provides the programmer with the ability to access all parts of the system.

Historically, the development of the Linux kernel, as well as the development of the original Unix operating system, was done in C. Most system software is developed in C. Device drivers are almost always written in C. An indirect benefit of being close to the hardware is speed. Code written in C tends to execute faster than code written in other languages. For a programmer who intends to work on system software, or who intends to develop code that closely interacts with hardware (peripherals or the main system), studying concepts using the C language provides opportunities to develop the most practical skills.

This choice does not preclude the study of other languages or advocate learning only C. Other programming concepts outside the scope of this text may be more readily studied and implemented using another programming language. However, it is the opinion of this author that a firm understanding of the programming language closest to the hardware better supports an understanding and proper use of a more abstract programming language.

## 1.2 • The Three Tools

The three main tools of a system programmer are a shell, a text editor, and a debugger. Familiarity with these tools increases programming skill and decreases the time it takes to get programs working properly. The following serves as an introduction to these tools and their interdependency. The real trick is in knowing how to use all three together. Noticeably absent from this list is a compiler. A compiler is a powerful tool that can certainly aid in program development. However, the compiler as a tool is addressed more fully in Section 6.1 during a discussion of program building.

### 1.2.1 Shell

It is assumed that the reader is familiar with basic operations in a shell, such as generating a file listing. There are a number of good books and online references that provide the necessary details to operate a specific shell. Appendix B offers a list of common commands. This section is concerned with *why* a shell should be used, and how a shell assists with program development.

A shell is a program that allows the user to run other programs. A shell is usually executed in a terminal. Historically, a terminal was a simple display and keyboard that was connected to a computer, providing a text-only interface to the computer. Today, most operating systems provide a graphical interface in which multiple virtual terminals can be run simultaenously, each running a separate shell. If a specific terminal is reserved for system administration or error messages on behalf of the entire computer system, then it is sometimes referred to as a console. In practice, the words shell, terminal, and console are often used interchangeably.

Most Unix systems provide several methods to start a shell. Some systems provide a shell after login, without the benefits of a graphical desktop. On other systems a shell must be started manually through a menu or mouse click interface. Once started, a typical shell looks like the ones shown in Figure 1.1. Commands are entered into the shell through a text-only interface. The shell informs the user that it is waiting for its next command via a prompt. In the figure, the prompt is `ahoover@video>`, which is the user name and machine name. Some shell configurations show the current directory or other information in the prompt. Throughout this text, the shell prompt will hereafter be shown as `ahoover@video>` to promote clarity.

A typical command is the name of a program, which starts execution of that program. For example, both the shells in Figure 1.1 show the execution of the `ls` program, which provides a listing of files in the current directory. Many of the programs run in a shell have text-only input and output, similar to `ls`. However, it is also possible to run graphics- (GUI-) based programs from a shell. For example, Figure 1.2 shows the result of typing `xclock` at the prompt; it starts the xclock program.

This method for starting programs may seem strange to those familiar with today's desktop approach to running programs. Clicking on an icon and perusing through a pull-down menu are typical operations used to run a program. Why then start a program, the shell, just to run other programs? The answer is flexibility. The desktop mouse and menu operations provide limited options in how a program is run. Typically, the desktop and menu shortcuts run a program

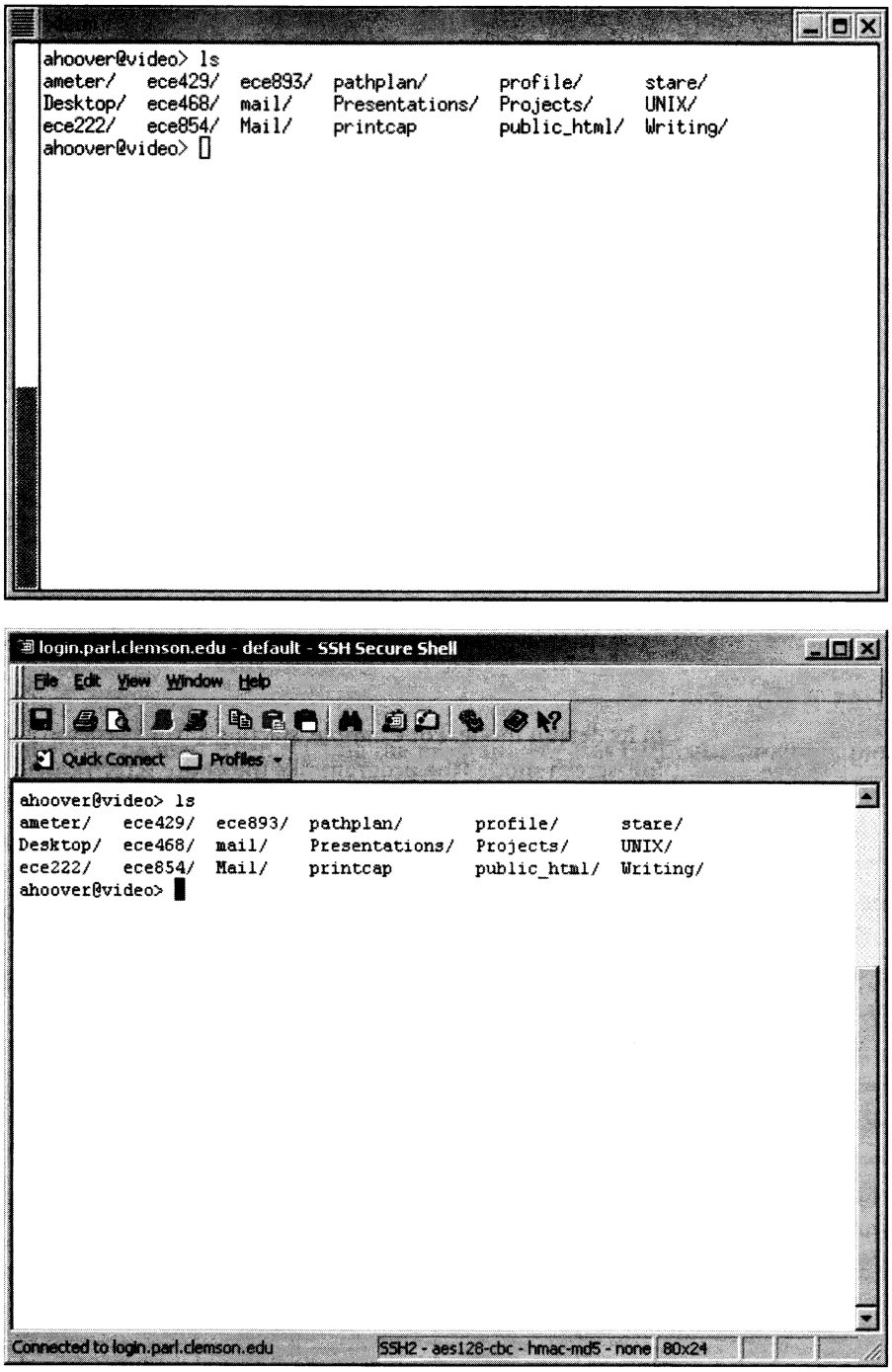


**Figure 1.1** Two examples of a shell, running in different terminal emulators.

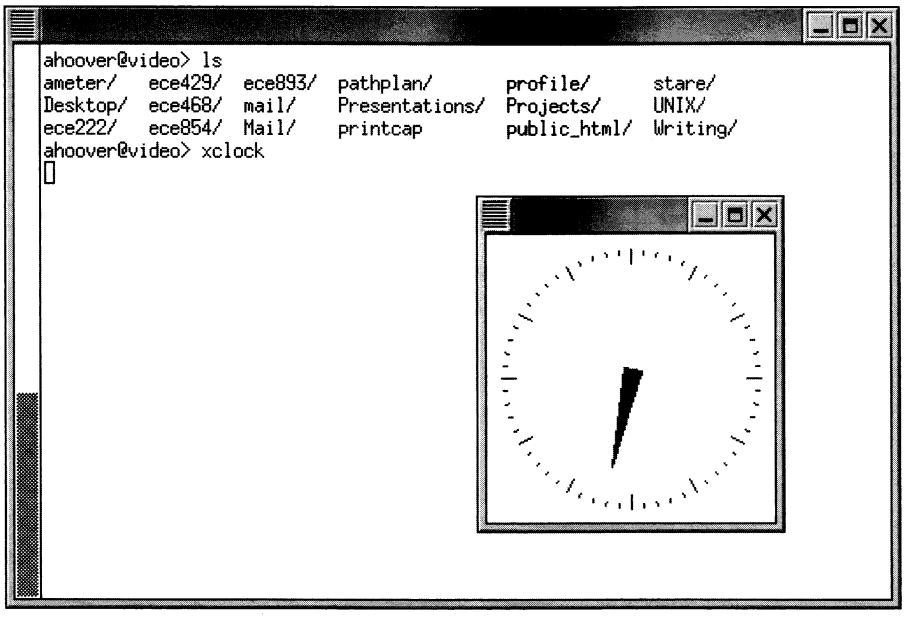


**Figure 1.2** Starting a GUI-based program by typing its name in the shell.

in its default mode. For example, starting a word processor opens the program in full-screen mode (the program fills the entire screen), with an open blank page, and all options set to their defaults (bold is off; text is left- justified; font is Times Roman; etc.). Suppose one desired to start the word processor with some of those options changed? A shell provides for this through command line arguments. A command line argument is anything typed at the shell prompt after the name of the program to execute. It provides information about how the user wishes to run the program. For example, typing `xclock -help` at the prompt yields the following:

```
ahoover@video> xclock -help
Usage: xclock [-analog] [-bw <pixels>] [-digital] [-brief]
       [-utime] [-fg <color>] [-bg <color>] [-hd <color>]
       [-hl <color>] [-bd <color>]
       [-fn <font_name>] [-help] [-padding <pixels>]
       [-rv] [-update <seconds>] [-display displayname]
       [-geometry geom]
ahoover@video>
```

In response to the `-help` command line argument, the xclock program displays its usage and then quits. The usage explains all the command line arguments

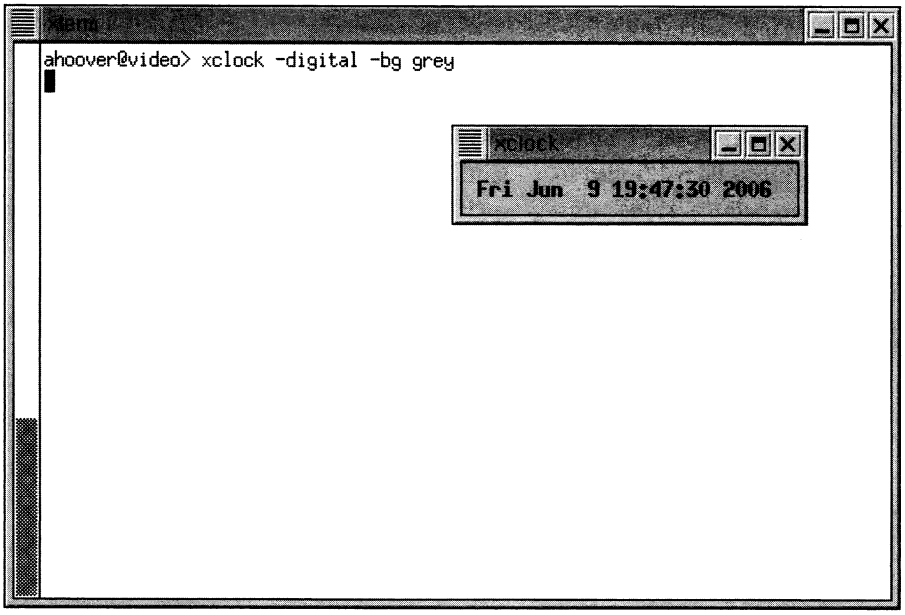


**Figure 1.3** Command line arguments change the way a program is run.

that can be used when starting the program. For this particular program, most of these arguments change the way the clock is displayed. For example, `xclock -digital -bg grey` causes the program to run as displayed in Figure 1.3.

The control and flexibility offered by command line arguments is often useful during program development and system administration. While many programs can be reconfigured while running, selecting options through menu interfaces can take time. Configuring the program at startup through command line arguments can save a great deal of time, especially if a program is run multiple times, such as during development.

A variety of shells have been developed over the years. Some examples include `sh`, `csh`, `tcsh`, `ksh`, and `bash`. On a Windows system, there is a very simple shell called `console`, sometimes called `DOS console` or `command prompt`. The shells differ in their intrinsic capabilities. Besides having the ability to run programs, and provide command line arguments, a shell has a list of internal commands that it can perform. For example, most shells have the ability to set up aliases for commonly typed commands. In the `tcsh` shell, typing

```
ahoover@video> alias xc xclock -digital -bg grey
ahoover@video>
```

**Table 1.1** Some common shell internal commands.

| Command | Description |
|---|---|
| alias | Create an alias |
| cd | Change directory |
| pwd | Print current working directory |
| set | Give a shell variable a value |
| which | Identify full path of program |

**Table 1.2** Some common system programs.

| Command | Description |
|---|---|
| grep | Search files for specific text |
| ls | List files and their attributes |
| man | Display manual (help) for command/program |
| more | Display a text file using pausable scrolling |
| time | Measure the running time of a program |
| sort | Sort lines in a text file |

causes the shorter command `xc` to become an alias for the longer command `xclock -digital -bg grey`. This can be quite useful when one is running the same command over and over, for example during program debugging. Table 1.1 lists some common internal commands for shells. All these commands are common to all the most popular shells (notably excluding the Windows console, which is intended to be only a limited shell). Unfortunately, different shells implement some of these internal commands using different syntaxes. For example, to create the same alias using the `bash` shell as given in the above example for the `tcsh` shell, one would type

```
ahoover@video> alias xc="xclock -digital -bg grey"
ahoover@video>
```

Most advanced programmers select a single shell with which to become proficient. Luckily, most of the shells are similar enough that proficiency with a particular shell allows a programmer to work adequately in any shell.

In addition to the internal shell commands, there are a number of programs pre-compiled and ready to run on most Unix systems. Table 1.2 lists some of the commonly used programs. These programs are called system programs because

they generally provide capabilities to manipulate, explore, and develop programs for the computer system. For example, `ls` is a system program that provides a listing of files in the current directory. Perhaps the most important system program to begin using is `man`. It accesses a manual of help files stored on the local computer system. Usually, there are individual "man pages" for all programs, and often for support files for the more complex programs. There are also man pages for all the functions within the various libraries on the system. These man pages usually come installed by default on a Unix system, but they are also posted many times over on the Internet, and can be found using a web search engine. It is also possible (and recommended) to find tutorials and other help via the Web on using a particular shell.

The purpose of both shell internal commands and system programs is to assist the system programmer. It is not terribly important to remember whether a particular operation is a shell command or a system program. Sometimes the operations are listed all together. The important thing is to become comfortable with the common operations that save time and effort. These programs are revisited in Section 5.3.1 during a discussion of pipeline chaining, and Appendices B and C provide longer lists.

### 1.2.2 Text Editor

The second tool considered here is a text editor. The basic operations of a text editor allow the user to write and edit code, save it to a file, and load it from a file. These operations are not much different from those supported by a word processor. In fact, it is possible to use a word processor to write code (although it is not recommended). However, there are additional features that a text editor can provide, beyond what a typical word processor provides, that are designed to support programming. For example, Figure 1.4 illustrates the finding of matching parentheses. Using the text editor `vi`, if the cursor is on an opening or closing parenthesis, pressing the percent symbol (`%`) moves the cursor to the matching parenthesis. Pressing it a second time moves the cursor back to where it started. The same keystroke matches opening and closing braces (the symbols surrounding blocks of code) and square brackets (the symbols used for array indices). For the text editor `gedit`, bracket matching is enabled through a menu option. For the text editor `emacs`, the bracket matching feature can briefly move the cursor to the opening bracket every time a closing bracket is typed, animating the grouping. Some text editors also provide color coding to highlight matching brackets and their enclosed blocks of code. Whichever text editor is used, the bracket matching feature can be quite useful in tracking down logic errors on expressions, flow errors on loops, array usage, and other bugs.

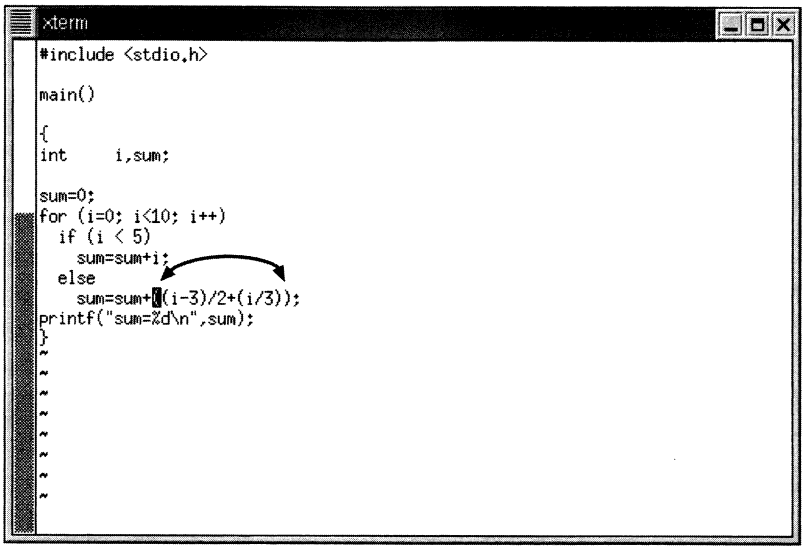


**Figure 1.4** Using a text editor to identify matching parentheses.

Perhaps the most important features a text editor can provide to a programmer are the ability to display the line number of the program for the given cursor location, and the ability to move the cursor to a given line number. Figure 1.5 illustrates an example. Using the text editor vi, the keyboard sequence `CTRL-G` displays the line number. The keyboard sequence `:N[CR]` relocates the cursor to line number N. Using the text editor gedit, this feature is provided through menu options. Using the text editor emacs, the keyboard sequence `[ESC]GN` relocates the cursor to line number N (assuming a commonly configured installation). Whichever text editor is used, it is important to learn these operations. They allow a programmer to use line numbers to communicate with a debugger. The programmer can tell the debugger to pause program execution at a given line number. The debugger can tell the programmer at which line number a given error occurred. In this manner, the programmer can work with the debugger to focus on the relevant line of code.

As a programmer becomes familiar with a given text editor, other useful features will be learned. The ability to search and/or replace a given string is often helpful in debugging variable usage. The ability to cut and paste a word, line, or block of code is often useful during code writing. The ability to arrange indentation helps support good coding practices. Some text editors support color coding of keywords as well as of code blocks.

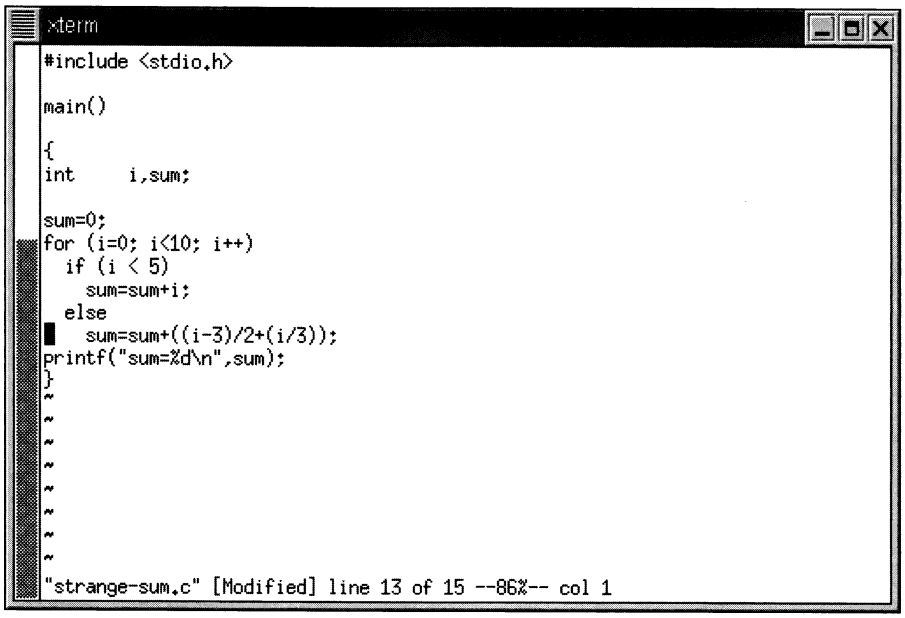


**Figure 1.5** Using a text editor to identify or find a program line number.

Not all text editors share all features. The text editor vi relies upon keystroke combinations to access features. The text editor emacs uses a combination of keystroke and menu operations. Newer text editors tend to rely more upon menu operations since they can be easier to find. It is beyond the scope of this book to delve into all the features for various text editors. One can easily find an online manual as well as online feature guides for all the popular text editors. The important thing to realize is that whichever text editor is used, a programmer should dedicate some time to becoming comfortable with those features that support programming.

### 1.2.3 Debugger

The debugger is perhaps the most important tool for a system programmer. It allows a programmer to observe the execution of a program, pausing it while it runs, in order to examine the values of variables. It also allows a programmer to determine if and when specific lines of code are executed. It allows a programmer to step through a program, executing it one line at a time, in order to observe program flow through branches. This section describes how a debugger works; the process of debugging is addressed in the next section.

The debugger is itself a program, which is executed like any other program. As with shells and text editors, there are many debuggers. In this book, examples are explained using the GNU debugger, which is usually executed as `gdb`. Although it is possible for a debugger to interoperate with more than one compiler, most debuggers are correlated with specific compilers. In this book, the concepts and examples are explained using the GNU C compiler, which is usually executed as `gcc`.

To explain how a debugger works, we will use the code example given in Figure 1.5. Suppose this code is stored in a file called `sum.c`. In order to compile the file, one could execute the following operation:

```
ahoover@video> gcc sum.c
ahoover@video>
```

This produces a file called a.out, which is an executable program. Typing `a.out` runs the program:[2]

```
ahoover@video> a.out
sum=29
ahoover@video>
```

The program is executed, running until it ends, at which time the shell prompts for another command. In order to use the debugger to run the program, one must follow a sequence of operations:

```
ahoover@video> gcc -g sum.c
ahoover@video> gdb a.out
(gdb) run
Starting program: /home/ahoover/a.out
sum=29
Program exited with code 07.
(gdb) quit
ahoover@video>
```

We will discuss each of these steps in detail.

First, when compiling, we make use of the command line argument `-g`. This tells the compiler that the executable file is intended for debugging. (Other compilers use similar flags or options.) While creating the executable file, the compiler will store additional information about the program, called a *symbol table*. The

2. This assumes the current directory is included in the PATH environment variable; otherwise `./a.out` must be typed.

symbol table includes a list of the names of variables used by the program. For our example, this list includes `i` and `sum`. The program is also compiled without *optimization*. Normally, a compiler will rearrange code to make it execute faster. However, if the program is intended for debugging, then any rearrangement of the code will make it difficult to relate which line of C code is currently being executed. When compiling for debugging, it is normally desirable to turn off all optimizations so that program execution follows the original C code exactly.

One can see the effects of compiling for debugging by looking at the size of the executable:

```
ahoover@video> gcc sum.c
ahoover@video> ls -l a.out
-rwxr-xr-x   1 ahoover  fusion       4759 Jun 19 18:53 a.out*
ahoover@video> gcc -g sum.c
ahoover@video> ls -l a.out
-rwxr-xr-x   1 ahoover  fusion       5843 Jun 19 18:54 a.out*
ahoover@video>
```

The executable has increased in size by 1,084 bytes. This increase in size is caused by the inclusion of the symbol table, and because the compiler was not allowed to optimize the code, so that its final output is not as efficient as it could be. Note that if you forget to compile for debugging, then the debugger will not be able to operate on your executable. Without the symbol table, or in the presence of optimizations, the debugger will be lost.

After compiling, we run the debugger `gdb` on the executable `a.out` that we just created. This does not immediately execute our program. It runs the debugger and loads our program into the debugger environment. This is emphasized by the fact that the prompt has changed. Instead of `ahoover@video>`, which is the shell prompt, we now see `(gdb)`, which is the debugger prompt. One can think of a debugger as a wrapper around a program. Figure 1.6 shows a diagram. When the debugger is started, it uses the symbol table and original C code file to keep track of what the program is doing during execution.

Once the debugger is started, it has its own set of commands. One of these commands is `run`, which begins execution of the program. In our example, this results in the program running as it would from the shell, eventually producing the output `sum=29` and then exiting. With the program finished, we are back at the debugger prompt `(gdb)`. To exit the debugger, we issue the command `quit` that takes us back to the shell.

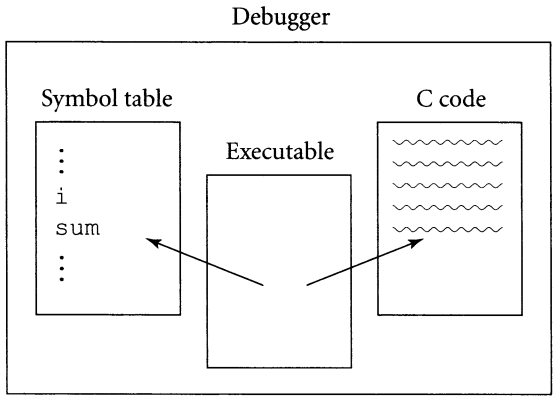


**Figure 1.6** A debugger relates an executable to the original variable names and source code file so that a programmer can track execution.

Sometimes it is useful to execute a program all the way to completion within a debugger. However, more often it is useful to execute a program "halfway," or through only part of its complete code. This is accomplished by setting a *breakpoint*. It tells the debugger to execute the program until that point is reached, at which time execution is to be paused. The programmer is then able to give commands to the debugger while the program is paused. For example:

```
ahoover@video> gdb a.out
(gdb) break 13
Breakpoint 1 at 0x804837b: file sum.c, line 13.
(gdb) run
Starting program: /home/ahoover/a.out
Breakpoint 1, main () at sum.c:13
13          sum=sum+((i-3)/2+(i/3));
(gdb)
```

At this point, the program has reached line 13 in the file sum.c for the first time. This should happen when the variable `i` reaches a value of 5 in the loop. Execution of the program is paused while we decide what to do. For example, we can display the value of `i`:

```
(gdb) display i
1: i = 5
(gdb)
```

As expected, the value of `i` is 5. We can also ask the debugger to tell us where the program is paused in relation to the original source code:

```
(gdb) where
#0  main () at sum.c:13
#1  0x4004e507 in __libc_start_main (main=0x8048460 <main>,
    argc=1, ubp_av=0xbffffa34, init=0x80482e4 <_init>,
    fini=0x8048530 <_fini>, rtld_fini=0x4000dc14 <_dl_fini>,
    stack_end=0xbffffa2c)
    at ../sysdeps/generic/libc-start.c:129
(gdb)
```

As expected, the first line tells us that we are at line 13 in the file sum.c, which is where we set the breakpoint. For now, we can ignore the other strange-looking line. Chapter 6 discusses functions and variable scope.

When a program is paused, there are three different ways to start it executing again: `step`, `next`, and `continue`. The `step` command executes the next line of code and then pauses again. For example:

```
(gdb) step
9       for (i=0; i<10; i++)
(gdb)
```

We had paused the program prior to the execution of line 13 using a breakpoint. After the `step` command has finished, we have executed line 13 and moved to the next line, which in this case is back to the top of the for loop at line 9. The debugger has again paused the program, prior to executing this line, and is awaiting our command.

The `next` command does the same thing, but if the next line of code is a function call, then the debugger will execute all the lines of code in that function call and then pause after the function returns. In other words, it treats the entire execution of the function call as one line of code. The `step` command will go into the function call and pause inside it at the first line of its code. Successive `step` commands can then be used to go through the entire function.

Issuing `step` or `next` commands repeatedly allows a programmer to run a program one line at a time, pausing after each line. This is called *stepping through a program*. For example, picking up where we left off above:

```
(gdb) next
10        if (i < 5)
(gdb) next
Breakpoint 1, main () at sum.c:13
13          sum=sum+((i-3)/2+(i/3));
(gdb) next
9       for (i=0; i<10; i++)
```

```
(gdb) next
10          if (i < 5)
(gdb) next
Breakpoint 1, main () at sum.c:13
13            sum=sum+((i-3)/2+(i/3));
(gdb) next
```

Each time the command `next` is issued, one more line of code is executed. Note that if a breakpoint is reached, the debugger also informs us of that, although it would have paused anyway because the next line of code was finished executing. Depending on the situation, using `next` to step through a program is often preferred over using `step`. The `step` command may cause the debugger to go into system library function calls, such as `printf()` function calls, which is rarely useful. (We can—hopefully!—expect the system library code to be more bug-free than code we are currently writing.) It is also useful to know that pressing [ENTER] alone will cause the gdb debugger to issue the previous command, so that one does not need to type "next" over and over. Most debuggers have a similar shortcut or keystroke to simplify stepping through a program.

The third method of continuing program execution is enacted by the `continue` command. It restarts execution and allows it to continue until a breakpoint is reached, until the program exits normally, or until the program reaches a line of code doing something illegal. Illegal operations include things like trying to divide by zero, or trying to access a bad memory location.

There are two different ways to observe the value of a variable. The `print` command is a onetime request to see the value. The debugger displays the value once only and will not display it again until requested. The `display` command is a request for ongoing observation. The debugger will display the value of the variable each time the program is paused. For example:

```
ahoover@video> gdb a.out
(gdb) break 13
Breakpoint 1 at 0x8048490: file sum.c, line 13.
(gdb) run
Starting program: /home/ahoover/a.out
Breakpoint 1, main () at sum.c:13
13            sum=sum+((i-3)/2+(i/3));
(gdb) display i
1: i = 5
(gdb) continue
Continuing.
Breakpoint 1, main () at sum.c:13
13            sum=sum+((i-3)/2+(i/3));
```

```
1: i = 6
(gdb) continue
Continuing.
Breakpoint 1, main () at sum.c:13
13          sum=sum+((i-3)/2+(i/3));
1: i = 7
(gdb)
```

Notice that each time the program is paused, the value of `i` is displayed. The `continue` command is used to resume execution of the program each time, causing it to run until it again reaches the breakpoint. Each time this happens, the loop counter `i` has increased by 1.

Multiple breakpoints can be set. For example:

```
(gdb) break 13
Breakpoint 1 at 0x8048490: file sum.c, line 13.
(gdb) break 8
Breakpoint 2 at 0x8048467: file sum.c, line 8.
(gdb) run
Starting program: /home/ahoover/a.out
Breakpoint 2, main () at sum.c:8
8       sum=0;
(gdb) display i
1: i = 134518128
(gdb) continue
Continuing.
Breakpoint 1, main () at sum.c:13
13          sum=sum+((i-3)/2+(i/3));
1: i = 5
(gdb)
```

The order of the breakpoints does not matter. When any breakpoint is reached, the debugger pauses. In this example, we displayed the value of `i` at line 8, before it had been given any value in the program. The strange value 134518128 is essentially a random value that happens to be stored in `i` at the beginning of execution of the program; later when the program is inside the loop, we see the more normal looking value 5.

Breakpoints can be removed during debugging using the `clear` command. This can be useful during extended debugging sessions. Sometimes it is simpler for a programmer to quit and restart the debugging process from scratch, using new breakpoints. Sometimes when a program has done something unexpected, a programmer will want to start over in an effort to identify where the program misbehaved.

The gdb debugger (and many other debuggers as well) includes a large set of commands not discussed here. These commands include capabilities to pause execution based upon variables being read or written (called *watchpoints*), to pause execution based upon signals (called *catchpoints*), and others. While these commands are useful, they are not necessary for common debugging. The commands and concepts introduced in this section generally suffice for the vast majority of debugging problems. The reader is encouraged to get started with this set of concepts and to explore additional debugging capabilities as the need arises.

### 1.2.4 Integrated Development Environment (IDE)

As systems have evolved, so have system tools. The interdependence of the three tools outlined in the previous sections has been recognized for many years. This led to the establishment of an integrated development environment (IDE). An IDE combines the three tools, along with a compiler, into a single program or program interface. Rather than separately running a text editor, compiler, and debugger, they all can be run together from within a single IDE. This allows the tools to be even more tightly integrated. Usually an IDE supports graphics-based operations that tie the individual tools together in a manner that can further speed program development and management.

At the time of this writing, popular examples of IDEs include Microsoft's Visual Studio, Eclipse, Sun Microsystem's NetBeans, and the GNAT Programming Studio. Some IDEs are intended to support a single programming language, such as NetBeans (for Java). Other IDEs support multiplate languages, such as Visual Studio and Eclipse; the former is proprietary, while the latter is open source. The advantage of multiple language support is to be able to assist a team of programmers in large-scale software development, or in multiplatform development. The centralized control of program development within one environment is often one of the biggest advantages to using an IDE.

An IDE is a powerful tool and belongs in the repertoire of any serious system programmer. However, it is important to understand what comprises an IDE, and how it works, by understanding the individual tools within it. Students of system programming should be encouraged to use the basic tools to gain at least some proficiency. That basic proficiency should help in future transitions to other IDEs or systems.

## 1.3 • How to Debug

The previous section introduced the shell environment and the three most important tools for system programming: the shell, the text editor, and the de-

bugger. In this section, we discuss methods to use the debugger. This includes deciding when and how to use the debugger to track down various problems.

In this discussion, we must make a distinction between fixing program logic and fixing program errors. Debugging is primarily inteded to help with the latter. Translating logical ideas into program code requires an understanding of how and when to use various programming constructs, such as a loop, a conditional, and an array. A debugger will not help a programmer determine whether a problem requires a pair of nested loops or whether a single loop will do the job. This is a logic concept, and should be approached through pseudocode writing, flowcharting, or other program development techniques. On the other hand, when a programmer is confident (or at least comfortable) with the logic being written into a program, then a debugger is an invaluable tool. It can assist the programmer in finding errors in the implementation or due to unanticipated details. For example, a debugger can help locate the use of an incorrect data type (e.g., using an `int` in place of a `float`), incorrect bounds on a loop, incorrect array indices, equation and logic errors, and typographical errors (some of the more devilish errors turn out to be nothing more than simple typos, such as a missing semicolon).

There are a handful of situations that are common to debugging problems. The following sections will describe each of these situations and go through an example debugging session. We will approach the debugging problem from the perspective of a programmer: we witness a symptom or some observed bad behavior on the part of a program. We then present a technique to locate the problem in the program code and, ultimately, to identify the cause of the program error.

## 1.3.1 Program Crashes

When a program stops executing in an unexpected manner, it is said to have *crashed*. Something went wrong, and the system was unable to continue running the program. For example, suppose the following code is contained in a file called `crash1.c`:

```
#include <stdio.h>

main()
{
int     x,y;

y=54389;
```

```
for (x=10; x>=0; x--)
  y=y/x;
printf("%d\n",y);
}
```

At the shell, we compile[3] and execute the program, only to find that it crashes:

```
ahoover@video> gcc -o crash1 crash1.c
ahoover@video> crash1
Floating exception (core dumped)
ahoover@video>
```

The error message "Floating exception" gives only a limited idea of what went wrong, and almost no idea of where it went wrong. The system has created a *core dump* file to help the programmer. It contains a snapshot of the contents of memory and other information about the system right at the moment that the program crashed. However, core dump files are usually large, containing far more than is needed for common debugging. Typically, core dump files are used only in advanced system programming problems.

A naive programmer might open up the C code file and begin studying the code, looking for possible sources of error. In a program as small as our example, this might even work. However, using a debugger is far simpler and will save a great deal of time. The idea is to run the program in the debugger until it crashes, and at that point look at what happened:

```
ahoover@video> gcc -g -o crash1 crash1.c
ahoover@video> gdb crash1
(gdb) run
Starting program: /home/ahoover/crash1
Program received signal SIGFPE, Arithmetic exception.
0x0804848b in main () at crash1.c:10
10          y=y/x;
(gdb)
```

The debugger tells us that the program crashed at line 10, and shows us the line of code at line 10. Looking at that line, it is easy to see that not many things could have gone wrong. Something must be wrong with either the value of y or x. The most likely scenario is that the value of x is zero, and that the program is therefore attempting to divide by zero. We can test this by asking the debugger to display the value of x:

3. The option to the compiler `-o crash1` tells it to name our executable `crash1` instead of the default `a.out`. It is a good habit to give executables meaningful names instead of calling all of them `a.out`.

```
(gdb) display x
1: x = 0
(gdb)
```

As we suspected, x has a value of zero. Now we can review the code to determine whether this was intended or whether we have an implementation error. For example, we might not have intended the loop to run until x>=0, and instead intended it to run until x>0.

Dividing by zero is not the only thing that can cause a program to crash. Perhaps the most common error resulting in a crash occurs when using arrays or pointers. For example, suppose the following code is stored in a file called crash2.c:

```
#include <stdio.h>

main()
{
int     x,y,z[3];

y=54389;
for (x=10; x>=1; x--)
  z[y]=y/x;
printf("%d\n",z[0]);
}
```

When we compile and execute this code, the program crashes:

```
ahoover@video> gcc -o crash2 crash2.c
ahoover@video> crash2
Segmentation fault (core dumped)
ahoover@video>
```

Using the debugger, we run the program until it crashes to find out where the problem occurred:

```
ahoover@video> gcc -g -o crash2 crash2.c
ahoover@video> gdb crash2
(gdb) run
Starting program: /home/ahoover/crash2
Program received signal SIGSEGV, Segmentation fault.
0x080484a2 in main () at crash2.c:10
10        z[y]=y/x;
(gdb)
```

A "segmentation fault" is usually a bad memory access; in other words, the program has tried to access a memory location that does not belong to the program. For example, an array has a specified size. Trying to access a cell index outside the specified size is a bad memory access. Looking at the line of code where the program crashed, we can see an access to the array `z[]` at cell index `y`. We can ask the debugger for the value of `y` and compare it against the allowed range (`z[]` was defined as a three-element array, so the allowed range is 0 . . . 2):

```
(gdb) display y
1: y = 54389
(gdb)
```

As we suspected, the value for `y` is outside the allowed range for indices for the array `z[]`. Once again, we have quickly identified the point where the program has misbehaved and can now go about the process of determining if the program logic or implementation is at fault.

Using a debugger to discover where a program is crashing is probably the most popular use for a debugger. During program development, if a crash is observed, the first action should almost always be to run the program in a debugger to locate the problem.

## 1.3.2 Program Stuck in Infinite Loop

When a program runs for a long time without displaying anything new, or prompting the user for new input, then it is probably stuck in an infinite loop. This means that the code executing in the loop is never going to cause the conditional controlling the loop to fail, so that the loop runs over and over. Of course, a "long time" is a relative expression. Some programs may need 10 seconds, or a minute or longer, in order to complete a complex calculation. However, if you can go for a cup of coffee, check the baseball scores, come back and still see the program not responding, then it is probably stuck in an infinite loop. For example, suppose the following code is stored in the file `infloop.c`:

```
#include <stdio.h>

main()
{
int     x,y;

for (x=0; x<10; x++)
  {
  y=y+x;
```

```
  if (y > 10)
    x--;
  }
}
```

When we compile and execute this code, the program seems to "run forever":

```
ahoover@video> gcc -o infloop infloop.c
ahoover@video> infloop
_
```

The "_" symbol indicates the cursor. The program is running but never ends, so we never see the shell prompt again. Eventually we press CTRL-C to force the program to stop executing.

We can perform the same operation using the debugger, but pressing CTRL-C in the debugger does not cause the program to quit. Instead, it tells the debugger to pause program execution at whatever line is currently being executed. We can then look at the surrounding code to determine which loop is executing infinitely:

```
ahoover@video> gcc -g -o infloop infloop.c
ahoover@video> gdb infloop
(gdb) run
Starting program: /home/ahoover/infloop
_        [...user presses CTRL-C...]
Program received signal SIGINT, Interrupt.
0x08048444 in main () at infloop.c:8
8       for (x=0; x<10; x++)
(gdb)
```

In this simple example, there is only one loop, so it comes as no surprise that the program is currently executing a line of code somewhere in this loop. In order to determine why the program will not finish the loop, we can watch the loop counter through an iteration:

```
(gdb) display x
1: x = 0
(gdb) next
10        y=y+x;
1: x = 0
(gdb) next
11        if (y > 10)
1: x = 0
(gdb) next
12          x--;
```

```
1: x = 0
(gdb) next
8       for (x=0; x<10; x++)
1: x = -1
(gdb) next
10        y=y+x;
1: x = 0
(gdb)
```

After having watched a complete iteration of the loop, we find that the counter variable x has the same value (zero) at the beginning of every iteration. Since it never reaches 10, the loop never ends. Now we can go about the process of examining the code involving x within the loop to determine the problem.

This technique for debugging is particularly useful when there are many separate loops within a program. It is the fastest way to determine which loop is faulty, and a good way to determine why the loop is not terminating properly.

### 1.3.3 Program Working Partially

Sometimes a program is working correctly, up to a point, when it suddenly starts misbehaving. For example, a program may be processing a series of input commands from the user. For the first few commands, the program seems to work fine, but at some point it starts producing erroneous output. How can a debugger help find the problem? It can help by focusing time and effort on the code in question, skipping over all the code that is seemingly working correctly. For example, consider the following code:

```
#include <stdio.h>

main()
{
int     choice;
float   ppg,rpg;

ppg=rpg=0.0;
choice=0;
do
  {
  printf("(1) Enter points per game\n");
  printf("(2) Enter rebounds per game\n");
  printf("(3) Quit\n");
  scanf("%d",&choice);
  if (choice == 1  ||  choice == 2)
```

```
    {
    printf("Amount: ");
    if (choice = 1)
      scanf("%f",&ppg);
    else if (choice == 2)
      scanf("%f",&rpg);
    printf("Points=%f  Rebounds=%f\n",ppg,rpg);
    }
  }
while (choice != 3);
}
```

The program is supposed to keep track of two statistics, the points per game and the rebounds per game. The program is supposed to allow the user to update these statistics, running until the user decides to quit. When this code is compiled and executed, however, the following happens:

```
ahoover@video> gcc -o wrong wrong.c
ahoover@video> wrong
(1) Enter points per game
(2) Enter rebounds per game
(3) Quit
1
Amount: 14
Points=14.000000  Rebounds=0.000000
(1) Enter points per game
(2) Enter rebounds per game
(3) Quit
2
Amount: 5.3
Points=5.300000  Rebounds=0.000000
(1) Enter points per game
(2) Enter rebounds per game
(3) Quit
```

The first option seemed to work fine, placing the entered value 14 into the ppg variable. But the second option put the entered value 5.3 into the wrong variable. What went wrong?

Using the debugger, we can work to find the problem. We recompile the program for debugging and load it into the debugger:

```
ahoover@video> gcc -g -o wrong wrong.c
ahoover@video> gdb wrong
(gdb)
```

Now we have to decide where to pause execution of the program. It seems as if the display of the menu is working correctly and that the problem lies somewhere in the code where the values are input. A logical place to pause the debugger is therefore after the menu has been displayed but before any input has been received from the user. Using a text editor, we can see that the code `if (choice == 1 || choice == 2)` is at line 16. Therefore, we set a breakpoint at line 16 and run the program up to that point:

```
(gdb) break 16
Breakpoint 1 at 0x80484ec: file wrong.c, line 16.
(gdb) run
Starting program: /home/ahoover/wrong
(1) Enter points per game
(2) Enter rebounds per game
(3) Quit
2

Breakpoint 1, main () at wrong.c:16
16        if (choice == 1  ||  choice == 2)
(gdb)
```

At this point, the program is paused at line 16. Now we can step through the program, one line of code at a time, to see what happens:

```
(gdb) next
18          printf("Amount: ");
(gdb) next
19          if (choice = 1)
(gdb) next
20            scanf("%f",&ppg);
(gdb)
```

Up until that last step, things seemed to be working correctly. However, we entered 2 at the menu, and yet the program is proceeding to the code that asks the user for the ppg value. What went wrong? The only variable involved so far is `choice`, which decided what part of the code to execute next. We can display its value:

```
(gdb) display choice
1: choice = 1
(gdb)
```

For some reason, the program thinks we entered `1` when we know we entered 2. How could this have happened? Now that we are focused on the problem of finding an error involving the variable `choice`, we look backward over the last

few lines of code and notice the problem. At line 19, the code reads `if (choice = 1)` instead of `if (choice == 1)`. The wrong code actually changes the value of `choice` to 1 every time it executes.

This is a simple example because the program is short. However, the important idea is to use the debugger to pause execution near where a problem is occurring, and then to step through the code in question. As a program increases in length, this technique becomes increasingly useful.

### 1.3.4 Loop Behaving Incorrectly

The logic inside a loop can be complicated to the point that it is impossible to mentally outline every possible case through every iteration. If such a loop is behaving incorrectly, a debugger can be used to observe the loop, pausing to examine the variables controlling the logic at each iteration. This can be useful not only for finding errors but also in correcting any problems with the logic inside the loop. For example, consider the following code:

```
#include <stdio.h>

main()
{
char    word[80];
int     i,j;

printf("Enter any word: ");
scanf("%s",word);
i=0;
while (word[i] != '\0')
  {
  if (word[i] == word[i+1])
    {
    j=1;
    while (word[i] == word[i+j])
      j++;
    printf("%d consecutive %c\n",j,word[i]);
    }
  i++;
  }
}
```

This program is supposed to search the word given by the user for consecutive occurrences of any letter, and report them. Compiling and executing, it works correctly on the first test:

```
ahoover@video> gcc -o badloops badloops.c
ahoover@video> badloops
Enter any word: apple
2 consecutive p
ahoover@video>
```

However, on a test having three of the same letter in a row, the program outputs an additional erroneous line:

```
ahoover@video> badloops
Enter any word: appple
3 consecutive p
2 consecutive p
ahoover@video>
```

What caused the output of the extra line "2 consecutive p"? The debugger can be used to watch the iterations of the outer loop to see what happened. The first line in the outer loop is `if (word[i] == word[i+1])`, at line number 14. This line tests if two consecutive letters match, while the following code determines the total span of consecutive letters. Recompiling the code for debugging, the idea is to pause the program at the beginning of each iteration of this loop to see how it behaves on the problem test case.

```
ahoover@video> gcc -g -o badloops badloops.c
ahoover@video> gdb badloops
(gdb) break 14
Breakpoint 1 at 0x80484d4: file badloops.c, line 14.
(gdb) run
Starting program: /parl/ahoover/ece222/book/1/badloops
Enter any word: appple

Breakpoint 1, main () at badloops.c:14
14          if (word[i] == word[i+1])
(gdb)
```

The variable `i` controls the behavior of the loop. Therefore, it is prudent to watch the value of `i` through every iteration, before continuing execution of the program.

```
(gdb) display i
1: i = 0
(gdb) continue
Continuing.
```

```
Breakpoint 1, main () at badloops.c:14
14          if (word[i] == word[i+1])
1: i = 1
(gdb)
```

After the completion of the first iteration, we see that the program worked correctly. No output was displayed, because the first two letters of "appple" do not match. The next iteration should produce output.

```
(gdb) continue
Continuing.
3 consecutive p

Breakpoint 1, main () at badloops.c:14
14          if (word[i] == word[i+1])
1: i = 2
(gdb)
```

The expected output "3 consecutive p" was observed. However, at this point it is possible to see why there will be additional erroneous output. The value of `i` is 2, which means that the next iteration will begin by comparing the second "p" in "appple" to the third "p." The value of `i` should have jumped ahead to test the "l" against the "e." Looking at the code at the bottom of the loop, we now realize that `i++` does not move ahead by enough characters in the case where multiple consecutive characters all match. The logic for this loop must be partially rewritten.

# 1.4 • Program Development

Program development concerns the writing of a program to solve a given problem. Knowledge of the syntax of a programming language is not enough. One must know how to use the language to approach programming problems. An analogy can be made to human conversation. Knowing the vocabulary and rules of grammar for a spoken language is not enough; one must be skilled in oratory in order to speak effectively. There is an art to using a programming language to develop and write programs to accomplish tasks, just as there is an art to using a human language to speak effectively.

One of the most important skills for program development is to be able to break a programming problem into a set of subproblems. Code can be written to solve each subproblem independently. That code can be tested before proceeding to the next subproblem. This is the ancient philosophy of divide and conquer. The

subproblems may themselves be broken up repeatedly into subproblems until a reasonable amount of programming work can be undertaken in isolation. Almost all programming work benefits from following some variation of this approach.

In order to demonstrate, consider the following problem. Write a program that takes an integer as input and then determines whether or not the given number is a sum of two unique squares. For example, given the number 13, the program would find that 13 = 9 + 4 is a sum of unique squares. Similarly, given 17, the program would find 16 + 1; given 90, the program would find 81 + 9.

How should this problem be approached? One possibility is to loop through all integers from 1 to X, where X is the largest integer whose square is less than the given number. Call this first integer *i*. For each of these numbers, a second loop could test all integers from 1 to *i*. Call this second integer *j*. Testing all possible pairs of *i* and *j* should find any sum of unique squares equal to the given number.

With a basic approach in mind, attention can be turned to writing code. Considering the above outline, the work can be broken into two parts. The first part is to prompt the user for a number, and loop through all possible integers whose square is less than the given number. For example:

```
#include <stdio.h>

main()
{
int     i,number;

printf("Enter a number: ");
scanf("%d",&number);
i=1;
while (i*i < number)
  i=i+1;
printf("%d is the largest square within %d\n",i*i,number);
}
```

This code does not solve the whole problem; it tries only to identify the boundary on the range of the first loop. The output statement at the end will be used to debug this portion of the program. If this code is stored in a file named `squares1.c`, then compiling it and executing it produces results like the following:

```
ahoover@video> gcc -o squares1 squares1.c
ahoover@video> squares1
Enter a number: 5
9 is the largest square within 5
```

```
ahoover@video> squares1
Enter a number: 14
16 is the largest square within 14
ahoover@video>
```

After a few tests, it is possible to see that the code works correctly but that after the loop is finished the value of *i* is 1 too high. This can be corrected by subtracting 1 from *i* before printing it out. For example:

```
#include <stdio.h>

main()
{
int     i,number;

printf("Enter a number: ");
scanf("%d",&number);
i=1;
while (i*i < number)
  i=i+1;
i=i-1;
printf("%d is the largest square within %d\n",i*i,number);
}
```

If this code is stored in a file named `squares2.c`, then compiling it and executing it produces results like the following:

```
ahoover@video> gcc -o squares2 squares2.c
ahoover@video> squares2
Enter a number: 5
4 is the largest square within 5
ahoover@video> squares2
Enter a number: 14
9 is the largest square within 14
ahoover@video>
```

This shows that the first loop seems to be working correctly. However, an additional test shows another error:

```
ahoover@video> squares2
Enter a number: 4
1 is the largest square within 4
ahoover@video>
```

This error occurs when the given number is itself a perfect square. This can be fixed by changing the exit condition for the loop from < to <= as follows:

```
.
.
.
while (i*i <= number)
  i=i+1;
.
.
.
```

Further testing of this version of the program reveals that it is now working correctly.

After code has been written and debugged for the first part of the problem, it is easier to work on implementing code for the second part of the problem. For example:

```
#include <stdio.h>

main()
{
int     i,j,number;

printf("Enter a number: ");
scanf("%d",&number);
i=1;
while (i*i <= number)
  {
  j=1;
  while (j < i)
    {
    if (i*i + j*j == number)
      printf("Found: %d + %d\n",i*i,j*j);
    j++;
    }
  i=i+1;
  }
}
```

Code has been added for looping *j* through all values for the second integer. If this program is stored in a file named `squares3.c`, then compiling it and executing it produces results like the following:

```
ahoover@video> gcc -o squares3 squares3.c
ahoover@video> squares3
Enter a number: 90
Found: 81 + 9
ahoover@video> squares3
Enter a number: 14
ahoover@video> squares3
Enter a number: 101
Found: 100 + 1
ahoover@video>
```

The program works as expected.

Even for a problem as simple as this example, it is possible to see how a divide-and-conquer approach can benefit during program development. Two errors were uncovered while implementing code for the first part of the program design. Finding these kinds of errors is more difficult when the individual parts of a program are not tested prior to further development or code writing. As mentioned earlier, this approach can benefit almost any programming work and should be practiced whenever possible.

## 1.5 • Review of C

The following serves as a quick review of the basic data types, operations, and statements in the C programming language. The reader is directed to any one of a number of excellent books covering the syntax of C for a deeper coverage. The goal of this review is to remind the reader of a few key concepts. This section can also assist the reader who has not yet studied C but has studied an introduction to programming in a related language, such as C++ or Java. It is possible to learn the syntax of C while studying the concepts of system programming in this text, provided that the reader is willing to undertake the extra burden. Such a reader is strongly encouraged to acquire an additional textbook that covers the C programming language to use in conjunction with this text. Several excellent candidates include:

1. *The C Programming Language*, 2nd ed., B. Kernighan and D. Ritchie, Prentice Hall, 1988, ISBN 0131103628.
2. *Programming in C*, 3rd ed., S. Kochan, Sams, 2004, ISBN 0672326663.
3. *C Primer Plus*, 5th ed., S. Prata, Sams, 2004, ISBN 0672326965.

### 1.5.1 Basic Data Types

There are four basic data types in C: `int`, `float`, `double`, and `char`. The `int` data type is intended to store whole numbers. The `float` data type is intended to store real numbers. The `double` data type is also intended to store real numbers but has twice the precision so that it can store a larger range of numbers. The `char` data type is intended to store character symbols and controls used to display text. The following code demonstrates the differences between the types:

```
#include <stdio.h>

int main()
{
int     x,y;
char    a;
float   f,e;
double  d;

x=4;
y=7;
a='H';
f=-3.4;
d=54.123456789;
e=54.123456789;

printf("%d  %c  %f  %lf\n",x,a,e,d);
printf("%d  %c  %.9f  %.9lf\n",x,a,e,d);
}
```

Executing this code produces the following result:

```
4  H  54.123455  54.123457
4  H  54.123455048  54.123456789
```

In the first line of output, the float variable has seemingly been rounded downward in the last displayed digit, while the double variable has correctly been rounded upward in the last displayed digit. In fact, the float has simply run out of precision. This can be seen in the second line of output, where both variables are forced to print to nine decimal places. The double variable has the correct value, but the float has erroneous values in the latter digits.

The `printf()` and `scanf()` functions are the primary output and input functions in C. They are included in the C standard library, which is usually linked to an executable by default (in other words, a programmer can use these functions

without worrying about where they come from). The syntax for them involves pairing up each variable in the list of arguments with a formatting symbol within the quoted string. For the details of this formatting, the reader is encouraged to consult a C programming book or the man page for either function.

## 1.5.2 Basic Arithmetic

The basic arithmetic operations supported in C include addition, subtraction, multiplication, division, and modulus (remainder). Within loops, it is common to increment (add 1 to) or decrement (subtract 1 from) a variable. The operators ++ and -- are provided for this reason. The following code demonstrates the basic arithmetic operations:

```
#include <stdio.h>

int main()
{
int     x,y;
int     r1,r2,r3,r4,r5;

x=4;
y=7;
r1=x+y;
r2=x-y;
r3=x/y;
r4=x*y;
printf("%d  %d  %d  %d\n",r1,r2,r3,r4);

r3++;
r4--;
r5=r4%r1;
printf("%d  %d  %d\n",r3,r4,r5);
}
```

The output of executing this code is as follows:

```
11  -3  0  28
1  27  5
```

The modulus operator can be used only on integer variables. All the other arithmetic operators can be applied to all variables.

### 1.5.3 Loops

There are three basic types of loops in C: `for`, `while`, and `do-while`. The `for` loop is intended to be executed a fixed number of iterations, known before the loop is entered. Hence, it is given both a starting condition and an ending condition. The `while` loop is intended to be executed an unknown number of iterations. Hence, it is only given an ending condition. The `do-while` loop is also intended to be executed an unknown number of iterations but will be executed at least once. The while loop may be executed zero times if it fails the condition on the first attempt. The do-while loop does not test the condition until it has finished the loop, so it will execute the loop at least once. The following code demonstrates all three types of loops:

```
#include <stdio.h>

int main()
{
int     i,x;

x=0;
for (i=0; i<4; i++)
  {
  x=x+i;
  printf("%d\n",x);
  }
while (i<7)
  {
  x=x+i;
  i++;
  printf("%d\n",x);
  }
do
  {
  x=x+i;
  i++;
  printf("%d\n",x);
  }
while (i<9);
}
```

The following is the output of executing this code:

```
0
1
3
6
10
15
21
28
36
```

The reader is encouraged to identify which lines of output came from which loop.

## 1.5.4 Conditionals and Blocks

The basic conditional in C is the `if-else` statement. It supports tests for equality (==), inequality (!=), and relative size (>, <, >=, and <=). Multiple conditions can be tested within a single statement using the logical AND (&&) and logical OR (||) operators to group the individual conditions. Statements (individual lines of code) are grouped using brackets ({}). In the absence of brackets, a conditional or loop statement applies only to the single following statement. The following code demonstrates conditionals and blocks:

```
#include <stdio.h>

int main()
{
int     i,x;

x=0;
for (i=0; i<5; i++)
  {
  if (i%2 == 0  ||  i == 1)
    x=x+i;
  else
    x=x-i;
  printf("%d\n",x);
  }
}
```

The following is the output of executing this code:

```
0
1
3
0
4
```

The indentation in the code is for convenience only; it does not affect the grouping of statements. The topic of formatting code for easier management and understanding is addressed in Section 6.2.

## 1.5.5 Flow Control

Normally, loop iterations must be run to completion. When the bottom of a loop is reached, control is returned either to the top of the loop or to the statement immediately following the loop, depending on the result of evaluating the loop conditional. There are two flow control statements that change the way an interation through a loop is executed: `continue` and `break`. The `continue` statement returns control to the beginning of the loop, testing the loop conditional to start the next iteration. In effect, it skips the rest of the current iteration and starts the next one. The `break` statement terminates the loop and immediately proceeds to the next line of code following the loop. The following code demonstrates flow control:

```
#include <stdio.h>

int main()
{
int     i,x;

x=0;
for (i=0; i<5; i++)
  {
  if (i%2 == 0)
    continue;
  x=x-i;
  if (i%4 == 0)
    break;
  printf("%d\n",x);
  }
}
```

Executing this code produces the following output:

```
-1
-4
```

Out of the five iterations in the loop, only two reach the printf() statement. The fourth iteration ends in the break statement, which terminates the loop, so that the fifth iteration never runs. Beginning students of C may be discouraged from using these flow control statements. The alternative is to use conditionals to control flow inside loops. The advantage to using control flow statements is that it simplifies (reduces) the number of program blocks, which usually simplifies the indentation that goes along with multiple program blocks.

There are also two control flow statements that are programwide: `exit` and `goto`. The `exit` statement immediately terminates the program. It is useful for handling unwanted situations, such as when a user inputs data outside an allowed range. The `goto` statement jumps program execution to the named line of code. In general, its use is discouraged. Unlike the other flow control statements, it can have complex consequences that can outweigh its benefits.