

Chapter 1

Introduction

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 for the computer system to support program development. A number of tools 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 the basic system file structure. Knowledge of these tools greatly enhances the ability of a programmer. Being able to use these tools elevates a *programmer* to a *system programmer*, often considered the top level of expertise in the field of 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. 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 so 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 linux 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 preferred 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 the teaching of 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. Towards that end, example code pieces and programs are provided throughout the text. Each chapter ends with numerous problems 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 introductory coverage obtained previously. The goal is to advance the programming skill of the reader to the point where these topics are well and comfortably used.

This text assumes that the reader has a basic understanding of C programming. For example, the reader may have completed a single semester of study covering an introduction to C programming. 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 basic operation of the computer system, such as navigating a directory or folder hierarchy and executing programs.

Curriculum

The material in this text is intended to be used for study in a single semester course in system programming. It is intended to follow an introductory course to programming. A colloquial title for the work might be “Second Semester Programming”. It is intended to precede the study of more advanced topics in programming such as data structures, algorithms, operating systems, and compilers. While it is not necessary to sequence these studies in this manner, the study of system programming will enhance the ability of a student to effectively implement the more advanced topics. There is a strong emphasis in this text on improving the practical programming skill of the reader, which should benefit a student in subsequent courses of study.

Why linux?

The majority of the material presented in this text can be studied on any computer system, using any operating system. However, we would be naive not to recognize the two most prevalent operating systems at the time of this writing: Microsoft WindowsTM and unix/linux. For reasons about to be explained, this text advocates the study of system programming on a unix/linux system, specifically (and hereafter referred to as) a linux system. Note that this discussion centers on which is to be preferred for *study*. There

are other ongoing debates as to which has the better business model, better development, and other issues. The interested reader is directed to seek other sources for discussion on these debates.

There are two kinds of automobile owners: hobbyists and drivers. An automobile hobbyist wants to understand how the vehicle works. He (or she) gets under the hood and figures out how the engine coolant system works, and how the fuel/air mixture system works. He crawls under the vehicle and learns how the braking system works. He studies these things with an eye towards modifications, either for improvement or repair. Thoughts of an automobile hobbyist bring to mind not only professional mechanics, but also “car hackers” that are constantly tinkering with the operation of their vehicles. In short, an automobile hobbyist is an expert.

Most automobile owners are not hobbyists, they are simply drivers. A typical motorist wants to be able to climb into a car, turn the ignition, press the gas and brake pedals, and turn the steering wheel. If something goes wrong, or an improvement is desired (for example the installation of a towing hitch) he calls upon the services of an expert. It is a testament to automotive engineering that vehicles have become so safe and easy to use that virtually anyone can drive. When automobiles were first manufactured, the typical motorist needed far more knowledge of its operation than does the average driver today.

Computer operators can similarly be divided into two types: users and administrators. A user just wants to turn the machine on and run his particular applications. If something goes wrong, or an improvement or upgrade is desired, the user calls upon the services of an administrator. An administrator on the other hand learns about the underlying operation of the system. He learns about system installation and configuration, process management, and troubleshooting. A computer administrator can get “under the hood” of a computer.

The Windows operating system is built for “computer drivers” (users). The typical Windows computer operator just wants to run applications. It is a testament to computer system engineering that computers have become so easy to use. One could argue that a major advantage of Windows is that it supports computer usage by anyone; it makes “computer driving” easy enough so that the typical user can operate independent of any real understanding of how the system works.

This is not to say that there are not Windows administrators. There of course must be people who are expert in how a Windows system operates.

The problem, at least for education purposes, is that a Windows computer is essentially a closed system. The system is designed to be “turn key”, 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 the automobile analogy, this is similar to housing the fuel/air mixture system out of reach of the driver.

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, these very properties that make it more useful to the typical computer user can make it frustrating for a student to study system operation.

Linux, on the other hand, 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 (the core program running on a linux system) 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 linux by typical computer users, who are not interested in this flexibility and openness. However, these design properties are what makes linux an attractive choice for a student to study 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 just as easily be run on a Windows system, or others. There are some important design issues that differentiate linux and Windows, such as the multi-user versus single-user nature of the systems. These differences will be discussed in detail at the appropriate places. Beyond those issues, deep down “under the hood”, the systems are largely similar. After all, an internal combustion engine is an engine, whether manufactured by Ford or Toyota,

and all brake systems ultimately stop the vehicle.

Why C?

Selection of a particular programming language is an old debate in computing. For application development, the debate still rages. However, for system programming, 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 only learning 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.

1.1 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. Later chapters of this text will explore additional aspects of these tools.

1.1.1 Shell

A shell, sometimes referred to as a terminal or a console, is a program that allows the user to run other programs. Most linux 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 Figure 1.1, 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 graphical (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 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

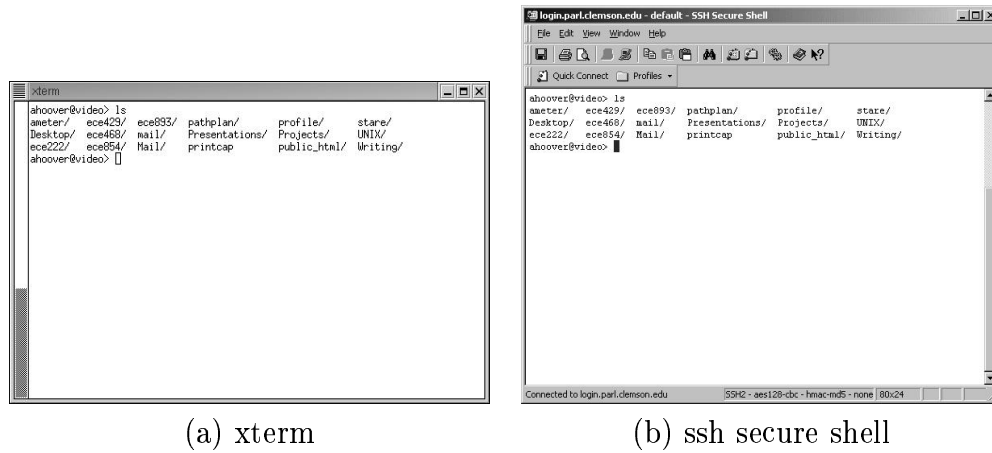


Figure 1.1: Two examples of a shell, running in different terminal emulators.

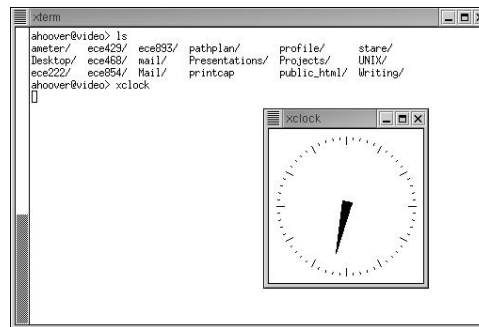


Figure 1.2: Starting a GUI-based program by typing its name in the shell.

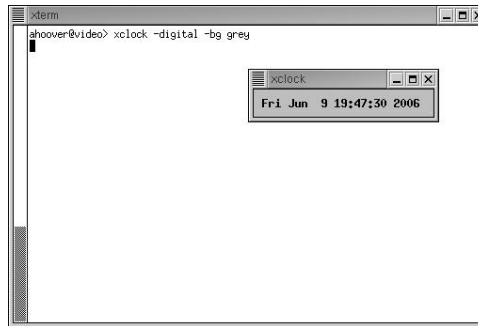


Figure 1.3: Command line arguments change the way a program is run.

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 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`, `tcsch`, `ksh`, and `bash`. On Windows systems 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

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

Table 1.1: Some common shell internal commands.

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>
```

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 only intended to be 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 linux 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

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

Table 1.2: Some common system programs.

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 linux 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.

1.1.2 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, 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 keyboard symbol `%` moves

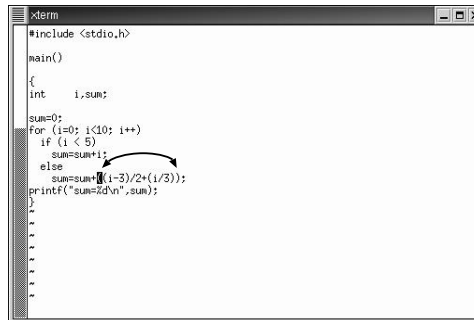


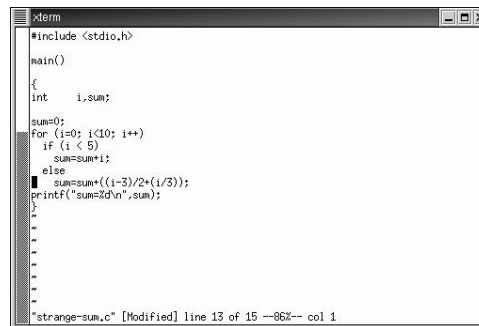
Figure 1.4: Using a text editor to identify matching parentheses.

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). This can be quite useful in tracking down logic errors on expressions, flow errors on loops, array usage, and other bugs.

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`. These operations 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 what 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 and code blocks.

Not all text editors share all features. The text editor `vi` is the oldest, perhaps one of the most powerful, but also one of the most arcane. It uses strange keystroke combinations to access features. Once learned, they can



```
xterm
#include <stdio.h>

main()
{
  int i, sum;

  sum=0;
  for (i=0; i<10; i++)
    if (i < 5)
      sum=sum+i;
    else
      sum=sum+((i-3)/2+((i/3));
  printf("sum=%d\n", sum);
}

"strange-sum.c" [Modified] line 13 of 15 --86%-- col 1
```

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

be among the fastest methods for implementing these features. However, the initiate should almost never start with vi. Better text editors have been invented over the years. The text editor emacs was perhaps the first major editor to displace vi among newer system programmers. Somewhat less powerful but easier to use text editors include pico and gedit. Whichever text editor a programmer chooses, he should dedicate himself to becoming comfortable with its features that support programming.

1.1.3 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 text, 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 text 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 would 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:

```
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 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. Compiling for debugging turns 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:

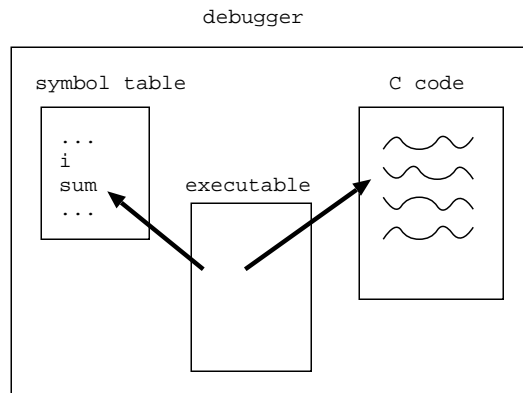


Figure 1.6: A debugger relates an executable to the original variable names and source code file, so that a programmer can track execution.

```

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 possible. 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` which takes us back to the shell.

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 “half way”, or only through 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 print out the value of `i`:

```
(gdb) print i
$1 = 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>, rtdl_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. In a later chapter this text will discuss functions and variable scope and revisit the debugger in that context.

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 one-time 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 one.

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/ahooover/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. However, it is more common for a programmer to quit and restart the debugging process from scratch, using new breakpoints. Usually when a program has done something unexpected, a programmer will want to start over in order to try 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.

1.1.4 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 can all 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, 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 multiple languages, such

as Visual Studio and GNAT; 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 multi-platform 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.

1.2 How To Debug

The last section introduced the shell environment and the three most important tools for system programming: the shell, the text editor, and the debugger. 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 intended 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 if a problem requires a pair of nested loops, or if 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.

1.2.1 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 ¹ 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>
```

¹The 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`.

The error message “Floating exception” gives only a limited idea of what went wrong, and gives 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 only used 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`:

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

As we suspected, `x` has a value of zero. Now we can review the code to determine if this was intended, or if 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.

1.2.2 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.

1.2.3 More debugging problems

There are other instances in which a debugger is useful for finding problems. These include:

1. Variable has wrong value.
2. Identify faulty program block.
3. Watch loop iterations.

4. Watch variable value.

Examples of these are forthcoming, and may be covered in class as time permits.

1.3 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 candidate books include:

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

1.3.1 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.

1.3.2 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 one to) or decrement (subtract one 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 only be used on integer variables. All the other arithmetic operators can be applied to all variables.

1.3.3 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 it will get 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 the 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 loops.

1.3.4 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 (`{` and `}`). In the absence of brackets, a conditional or loop statement only applies 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 a later chapter.

1.3.5 Flow control

Normally, loop iterations must be run to completion. When the bottom of a loop is reached, control is returned 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 iteration 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 program-wide: `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, it should be avoided by all but the most experienced programmers. Unlike the other flow control statements, it can have complex consequences that can outweigh its benefits.