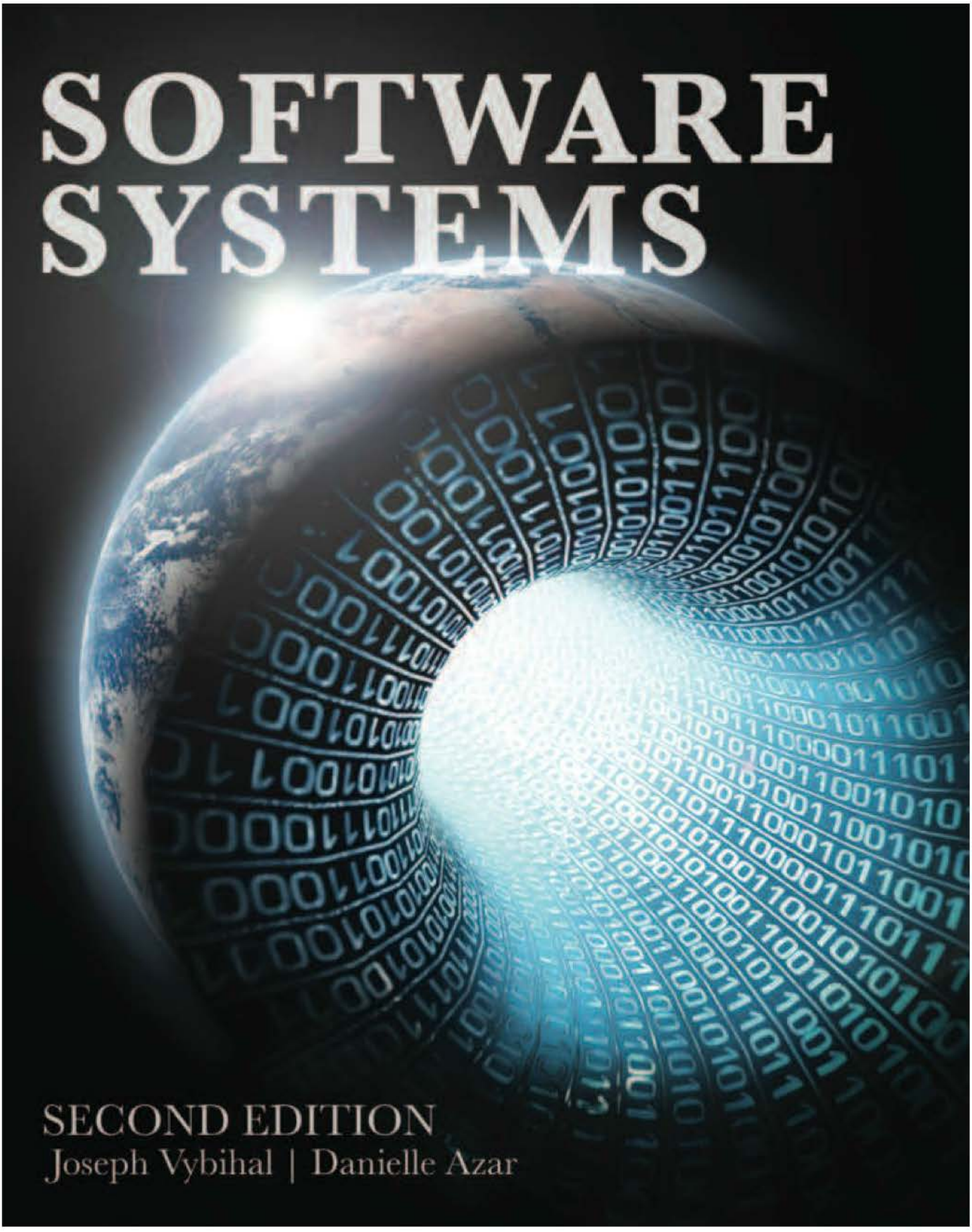
SOFTWARE SYSTEMS



SECOND EDITION

Joseph Vybihal | Danielle Azar

SOFTWARE SYSTEMS

SECOND EDITION

Joseph Vybihal
McGill University

Danielle Azar
Lebanese American University

Kendall Hunt
publishing company

Contents

To my wife Electra and my children Abigail and Bethany,
as well as Remus. —JPV

To my life partner Elie, my parents, and my sister Pascale. —DA

Preface

INTRODUCTION

The Software Systems course at McGill University covers many fundamental topics in computer science. This comprehensive approach allows the student to experience the manner in which multiple software systems can be combined, through programming, into a single super system. The problem with the course was its requirement that students should purchase five textbooks. Textbooks are expensive. This reduced most students to purchasing less than what they needed. The more adventurous student attempted to find material online, share, or purchase older versions of these texts. We have attempted to address this issue through this single textbook.

We assume that readers of this text have already completed at least one college-level programming course. With this in mind, some of the details of how programming works are assumed to be understood. Why an if-statement works the way it does, for example, is something the reader should already be familiar with. This assumption would also go for understanding the run-time of functions and methods. The text will define these terms and provide examples for these concepts, but instruction will focus on more pertinent issues.

Our goal is to introduce readers to programs that intercommunicate with each other. These programs form a system, which in turn constitutes a new organism, a super program. These super programs are application programs in their own right; examples would be Internet stores and cloud computing. By the end of this text, readers will have amassed basic skills ranging from systems programming to cloud computing. This text should be viewed as an introduction to these areas.

KEY FEATURES

The text first introduces readers to operating systems and shell scripting. This introduction enables the student to manipulate the operating system environment both by hand (through command-line commands) and through programs (using shell scripting). Since networks and log-in sessions are an important part of modern operating systems, readers will acquire knowledge of these domains as well, and also of how they can be influenced by programming. Our operating system of choice is Unix (and its derivatives).

The text continues with advanced programming techniques using the C Programming language. Readers learn how to mix operating system commands and C Programming to create their first Software System. Fundamental programming tools and techniques are introduced next, using the GNU Tool Set together with simple software engineering strategies.

Then we look at how the Internet functions and communicates over a network. We see how to program in that environment. Readers then learn how to combine all these elements into a super program, like a web store, a web game, or a cloud application. The primary tools for this programming will be the Internet technologies like HTML, CGI, CSS, XML, PHP, and JavaScript. Server-side Internet technologies would include Unix, C, Python, and Perl, to name a few.

The Instant chapters are used for further explorations in languages and tools that facilitate software systems programming. If you are an experienced programmer, you can go directly to the Instant chapters. Each one stands on its own.

HOW TO USE THIS BOOK: CHAPTER ORGANIZATION

This text assumes readers have already completed a college-level course in programming, ideally Object Oriented programming; but, any programming language would do. Given this qualification, readers can take two paths: "Other than the college-level programming course, I do not know much more about programming" (figure P1), or "I am an experienced programmer who wants to learn more about web development" (figure P2).

FIGURE P1: Standard Path

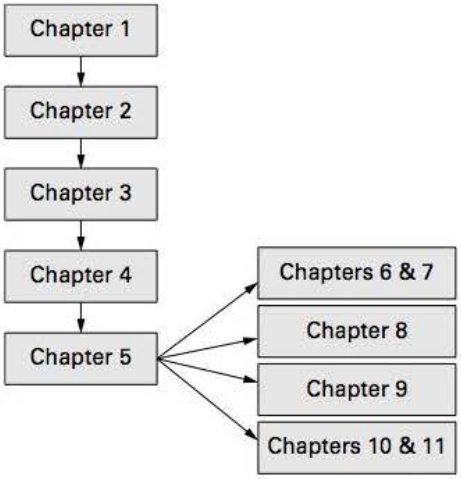
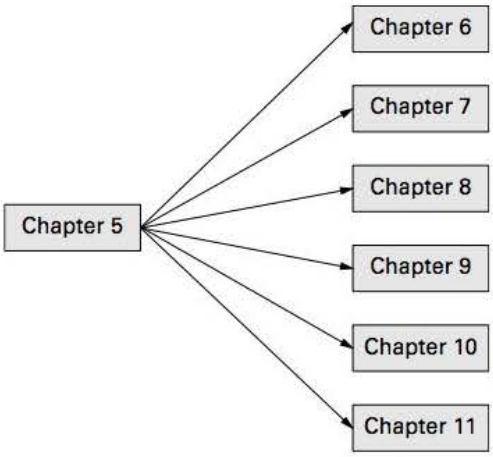


FIGURE P2: Experienced Path



For readers who have only a single college-level programming course, then figure P1 defines the Standard Path. This figure shows the way an expected reader of this textbook will study. As you can see in the diagram, the student learns linearly until chapter 5. At this point, the student has enough background to make choices. Four choices are given to readers. They can study all four topics, or they can pick and choose to meet their specific needs. Chapters 1 through 5 address the fundamentals of software systems and programming techniques: What are software systems, operating systems, C, systems programming, and Internet programming? With this background, readers can then direct their own education selecting, in any order, from: server-side programming languages (Python and Perl), data formatting (XML), dynamic HTML (DHTML), and client-side programming languages (Java Script and Java Applets).

For experienced programmers, figure P2 assumes that these individuals already understand software systems, operating systems, C programming, and simple software engineering techniques. Figure P2 assumes that these experienced programmers are novices to the Internet, and so, recommends they read chapter 5 to introduce the basic techniques for building web pages. Then readers can expand their skills by selecting any or all of the Instant chapters in any order that they like.

HOW TO USE THIS BOOK: KEY TOPICS BY CHAPTER

Readers who would like to do a topical study instead can use the table below. It is organized by topic, with a short description and prerequisite knowledge, and it then directs readers to the chapter in the text. Readers can scan the list of topics and identify the one(s) they are interested in, making sure they have the prerequisites. Readers who do not have the prerequisites should consider reading those sections before proceeding.

Topic	Description	Prereq	Chapter
Systems	Systems as a concept	None	1
Systems Programming	About programming a system	C	4
Unix Architecture	How does Unix work?	None	2
Unix Shell	Using the Unix shell	Unix	2
Shell Programming	Unix script programming	Unix Shell	2 (5)
Sessions	What are sessions?	Unix	2 (5)
Internet	How does the Internet work?	None	1 & 5

(*Continued*)

Topic	Description	Prereq	Chapter
C Programming	Introduction to C programming	None	3
C Libraries	Various standard libraries	C	3
C system programs	C, Libraries and Unix	C Libs	4
GNU	gcc, gdb, gprof, make	Shell & C	4
Software engineering	Simple team development	None	4
Web development	Introduction to web programming	None	5
HTML	Introduction to HTML	None	5
CSS	Introduction to CSS	HTML	5
CGI	Introduction to CGI	HTML	5
Server side	Server-side programming	C & CGI	5 & Instant
Client side	Client-side programming	Into to Web	5 & Instant

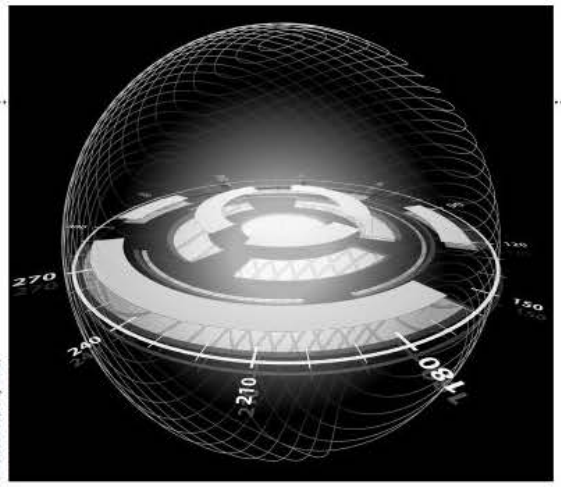
Acknowledgments

A book does not get completed w ithout the help fr om many individuals, people besides the authors. I'd like to thank the students who have proofread the first edition of this book, allowing us to construct a much better second edition of the text . I'd like to thank my wife, Electra, for being supportive. I'd also like to thank McGill for giving me the time to develop this book. Of course, I'd like to thank Danielle Azar, who co-authored this book. It is always a pleasure. —JPV

My thanks go to my life partner, Elie, who has always supported me and helped me in all the ways he could. I also thank my parents, Mimi and Antoine, and my sister Pascale, for their support, sense of humor, and patience, which were greatly needed. I thank the Lebanese American University for giving me the opportunity to work on the book. I also thank the co-author Joseph Vybihal, who has been a great colleague and a wonderful friend. —DA

CHAPTER ONE

Understanding Software Systems



Long has passed the time when a single programmer all on his or her own could write an entire software application without some help from tools, libraries, or the operating system. Today's software integrates and interfaces multiple technologies to run properly. A modern programmer needs to understand more than the programming language used to write the application. They need to master database technologies, Internet and networking technologies, the operating system's features, and off-the-shelf libraries, to name a few. This is what this book is about, learning how to build a modern application program.

This chapter introduces the reader to the concept *systems* as it pertains to computer science. A system is something that does not function all on its own. A system is composed of many smaller parts, called sub-systems, that rely on each other. They work as a team to accomplish some goal. A soccer team would be a good analogy. Each player is a sub-system, fully functional and well designed but incomplete without the rest of the men or women that constitute the soccer team.

In this chapter, we will explore three systems that are critical within the context and scope of this book. The first and fundamental system is the *software system*, without which we would not even have programs. Then we will look at two external systems the software system needs to interact with: the Internet and the run-time environment, two very important systems that every programmer is required to understand well.

SOFTWARE AS A SYSTEM

Software development, in modern times, has become synonymous with the idea of integrating many technologies into a single application. It has become a story about interconnecting

systems. Software does not stand alone anymor e. It is a conflagration of many technologies into a single applica tion. The programmer, when cr eating an applica tion, builds upon systems developed by others. This concept is often missed b y first-time programmers because this relationship is often encapsula ted within programming language statements or software development environments. To fully appreciate this idea, let us look at the following story that realistically describes the evolution of programming.

In the beginning, programmers created programs from "scratch" (all on their own using only a single programming language). These first programmers had only primitive tools. Their tools were even more primitive than what I will describe here, but suffice for generality, they had a simple text editor, an assembler pr ogramming language, and a m achine (the precursor of today's modern computer). They had nothing else. In these simpler times, the "true" programmers created everything they needed on their own, without help. If they wanted to read a single character from the keyboard, they had no tools to do this. They had to build it on their own, using assembler. This process required them to h ave intimate knowledge of the m achine so that they could locate and extract the information they needed. They would have had to write something like the following:

```
GetChar:
    lui $a3, 0xffff       # Load the base address of memory map
CkReady:
    lw $t1, 0($a3)        # Read from receiver control register
    andi $t1, $t1, 0x1    # Extract ready bit
    beqz $t1, CkReady     # If 1, then load char, else keep looping
    lw $v0, 4($a3)        # Load character from the keyboard register
    jr $ra                # Return to place in program before function
call
```

As you can see from the above assembler function, to read a **single** character from a keyboard requires you to kno w some ph ysical information about th at keyboard. In the abo ve example, that would include the addr ess locations in memor y where the keyboar d is ph ysically connected (without loss of gener ality, this could also be a logical connection). I n this case, two hardware registers need to be accessed: the Con trol and the Da ta register at addresses 0xffff0000 and 0xffff0004, respectively. You would need to kno w additionally that bit 1 of the Control register is set to tr ue when da ta has been put in to the Da ta register. This is checked in a busy loop un til that event occurs. Then the program moves the input ch aracter into the program.

If the programmer was smart, the programmer would reuse the function GetChar. Now that we know how to read a single ch aracter, a new, mor e complex function could be cr eated. Let us create the function GETS, read it as GET-S, for get-string. This new function would be a Wrapper function. It would wrap GetChar within another function called GETS that would use GetChar repeatedly, reading multiple characters until some special ch aracter, like the en ter key,

would be pressed to indicate the end of the series of characters. The series of characters would be returned as a contiguous block of characters representing a sentence or a word. We call this a string. Often, it is represented as words between double quotes, like: "I am a string". This would also have to be done in assembler, but we could hide the hardware-related knowledge of how to access a single character within the GetChar function. The writer of GETS would not need to know the actual physical method for getting a single character from hardware.

The above is an important advancement. It is called Encapsulation. Encapsulation uses information hiding, information that is hidden behind some kind of structure. In the above paragraph, the GetChar function was used as that hiding structure. It hid the information about accessing characters from hardware. The builder of GetChar could now build GETS, or any other function that needs to access characters, without actually needing to know how to physically get a character from the hardware.

Information hiding, encapsulation, functions, and wrapper functions are important concepts in computer science. They are ways of controlling *complexity*. Writing software can be complex business if you need to do everything from scratch. But, I am getting ahead of myself. We will come to these concepts again, later.

The above story is true for everything the primitive programmer had to do. If the programmer wanted to write to the screen, print on a printer, access a mouse, or anything else, that would have to be built from the ground up, from scratch. This would be a lot of work, but it would be *all* the work the programmer had to do, and he or she would be proud of this accomplishment. Programmers could even show off their work. It would have taken a lot of time and effort, but, if it was useful, then it would have been worth it. This is how our primitive programmer thought, and he would have been right because it cost a lot of time, and pride would have been a justifiable feeling. But, programmers are not called to make only one program in their lifetime.

Since primitive programmers wrote many programs, after a while, they realized that the functions they wrote to solve a problem in one program seemed to reappear in other programs. For example, GetChar is a function that programmers would need in almost all the programs they might write.

Smart programmers created special text files that they called Common.txt or Repository.txt. In these files, they would copy and paste all the useful functions they came across. The idea was that they would merge this file with their next programming project. This idea became so popular that programming languages evolved to contain special *including* commands that would automatically merge a text file within a program. In the C programming language, we see this in the #include command.

These Repository.txt files became so popular that programmers began to share them with each other. We then had RepositoryJack.txt and RepositoryMary.txt. This practice then evolved to become specialized text files, like RepositoryIO.txt and RepositoryMath.txt. Programming

communities were created to gather this information and disseminate it to other programmers. Electronically, this was done through Electronic Bulletin Boards, Diskette sharing clubs, and sold by special repository companies (the Internet did not exist yet).

Programming advanced to include higher-level languages like COBOL, FORTRAN, Basic, C, and Pascal. These new programming languages used encapsulation and wrapper functions, so much that programmers did not need to program in assembler anymore. In the C programming language, we have the wrapper functions getc() and gets() that call the assembler functions GetChar and GETS.

Information hiding advanced so that programmers not only were not required to know how to physically extract information from hardware, but in addition, they did not even need to know assembler. These new programming languages used higher-English words like if-statements and for-loops. You could create complex data structures like strings and arrays. But an even more important advancement came about: the development of the Library.

Programming languages like FORTRAN, C, and Pascal had comprehensive libraries that standardized the Common.txt and Repository.txt files. When programmers purchased a programming language, not only did they receive a compiler for that language but they also received hundreds of library files. These were functions that they did not need to write. These were functions that they could just include into their programs as if these functions were part of the language. This became so common that they are considered today to be part of the language. Every language had its own set of independent libraries. Today, the .NET technology contains libraries that can be used within multiple languages. This means that you only need to learn one library, and then you can reuse it in multiple languages. But, this is still not perfect because, for example, .NET is a Microsoft product and works primarily with Microsoft language compilers. Other companies still have their own independent .NET-like libraries. It was actually Borland that began this .NET-like trend. There may be hope, though; competitive compilers sometimes allow programmers to use competitor libraries. So, in the future, we may need to only learn about one common library for all languages.

The next step in software development was in the programming languages themselves. They evolved from procedural and functional text-based user-interface languages to object-oriented, event-driven, graphical-based user-interface languages. From languages that, when compiled, could run only on the computers for which they were compiled, to languages that are computer independent, like Java and Python. This book attempts to introduce the reader to many of these popular, computer-dependent and computer-independent languages.

So far, we have seen assembler turned in to functions that were turned in to libraries. We then saw libraries included into languages to make the languages more robust and powerful. Finally, the languages themselves were becoming more sophisticated. But, more advances were coming.

Primitive programmers only had a text editor to write their code with and a compiler to convert their code into the machine's internal language (appropriately called Machine Language). This was fine, but in the 1980s, integrated development environments came into being. These programming environments allowed the programmer to do multiple things within a single program. Programmers could write their code, just like in a text editor. They could also compile their code and see the errors on the screen without having to leave that text editor-like program. They could trace and debug their errors from that same program. There were even tools, called Screen Generators and Report Generators, that automatically generated code for them. These programs drew screens and formatted reports and then wrote the program, as well. Many powerful development environments exist today: Eclipse, Code::Blocks, the Microsoft Visual environment, and the Borland Builder environment, to name a few.

But things continued to advance. Not only were the programming languages and development environments becoming more elaborate, but the operating systems these programs ran on also changed. In addition, networks are now commonplace, and the Internet is everywhere. Our programs must now interface with touch screens, styluses, mice, and many other things.

Long has passed the time when programs were simple.

But this does not mean that programming has become hard; the reality is the contrary.

A programmer needs to understand three important systems in these modern times: the Internet, the run-time environment, and the software development environment. The next two sections introduce two of these systems: the Internet and the run-time environment. We will look at development environments in another chapter.

THE INTERNET AS AN EXAMPLE

The Internet is the quintessential system. It is so interdependent that it is amazing it works at all. To fully explore the inner working of the Internet goes beyond the scope of this text. Entire volumes have been written, and I would refer you to those tomes. But, we do need to understand it in a general way since a portion of this book relates to programming over the Internet.

Let us begin this explanation from a place we understand well: sending email with your personal computer or laptop. Did you know that your email program does not know how to send email? Or, maybe I should say that it does not know how to get your email to the intended recipient. The only thing the email program knows how to do is to ask for the recipient's email address, provide a space to input the message, and then provide a button labeled SEND. At this point, if you pressed the SEND button, the program would close the screen and put your email into a data structure called a *packet*; it then passes that packet off to the computer's operating system. That is the extent of what the email program knows how to do. At this point, it is trusting that the operating system knows what to do next.

The operating system itself does not know that much, either. For example, it does not know how to communicate over a network; but, that is exactly what we need to do. Fortunately, a computer has a hardware device called a *network card* that knows how to communicate with a network. The job, then, of the operating system is to take the packet data structure it received from the email program and convert it into something the network card knows how to handle. Without going into too much detail, the operating system formats that packet in a standard way using binary to encode all the information in the packet. This packet, once formatted, is passed to the network card for transmission over the network.

A packet is a data structure that has the following fields: source address (the sender's address), destination address (the recipient's address), the message size in bytes, the message itself, and a *check-sum* field to help adjust for damages that may occur when transmitting a packet over a network. A simple check-sum would be to count the number of 1 bits in the packet. That number would be saved in the check-sum field. Once the packet arrives at its destination, the computer there would compare the number of 1 bits in the packet with that check-sum. If they match, then the computer would assume that all is well. There are more elaborate techniques. A packet data structure written in C could look as follows:

```
struct PACKET_REC
{
        String source;
        String destination;
        int size;
        String message;
        int checkSum;
};
```

The operating system would have converted the above structure into a standard binary format and passed it on to the network card.

FIGURE 1.0: Physical Connections

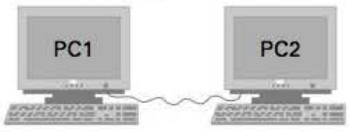


FIGURE 1.1: Radio Connections

To communicate over a network requires the conversion of the packet data structure into signals. A network is defined to exist when one computer is physically connected to at least one other computer. This physical connection could be literal, as when a computer is connected to another computer using a wire or cable (figure 1.0). Or, the connection could be more ethereal and exist in the form of a connection through radio (figure 1.1). In any case, communication occurs through signals. Signals are waves, like sign waves. They have crests and troughs. They also can travel quickly from one end of a wire to the other end. Waves and binary share a common property. They are both binary.

Binary 1 is analogous to a wave crest, and binary 0 is analogous to a wave trough; see figure 1.2. Once the operating system converts the packet data structure into binary, then the network card can quickly convert the binary into signals, resulting in a packet of data being transmitted over the network. This signal-set of information is strictly referred to as a packet. One common way of converting binary into signals is to convert the binary into sound waves. The binary 1 would be a high-pitch sound, and the binary 0 would be a low-pitch sound. When networks existed over the phone lines, this is how it was actually done.

Let us assume that we have only two computers connected together in a simple network using a wire. One end of the wire is plugged into the network card of one computer, and the other end of the wire is plugged into the network card of the other computer (figure 1.0). One individual writes an email and sends it to the other computer. Since there is only one other computer, the network card will simply convert the packet data structure into a packet of signals and transmit that over the wire. The other computer is passively listening to the wire. Once it hears something, it converts the sounds into binary, creating the packet data structure; then the reverse process occurs on the receiving computer. The binary packet is sent to the receiving computer's operating system, which converts it to the format the recipient's email program can handle. The email program then displays it to the recipient.

This setup is not the Internet yet. Assume we have more than one computer connected together into something we call a Local Area Network, or LAN. A network of this form is when we have many computers connected together through a special computer known as a server. Figure 1.3 shows an example of a network configured as a Star (even though it does not look like that exactly). Notice that each computer in this network, including the server, is a PC. Also notice that they are each uniquely identified by a number. In this example, they are numbered from 1 to 4. Notice further that the non-server computers are not directly connected to each other. Instead they are connected directly to the server through a special connection box called a Hub or a Router. This connection box is directly connected to the server. These connection boxes come in many forms, depending on how smart they are. This simple box is the hub, and its basic property is to connect the non-server computers to the server. On the other hand, the hub is designed to connect

FIGURE 1.2

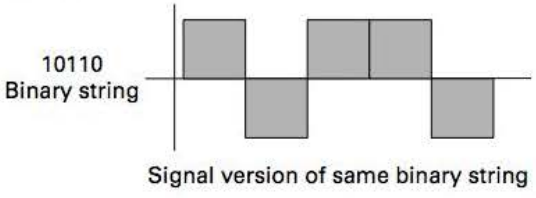
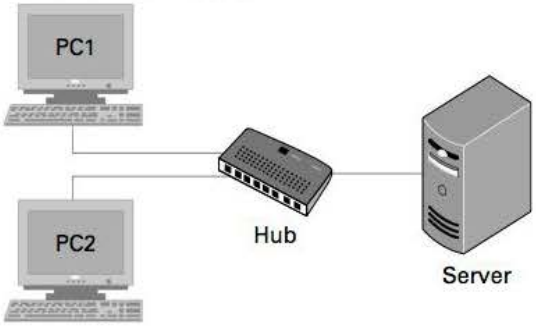


FIGURE 1.3: Star Network



the server to all the other non-server computers. It is a many-to-one connection from PC to server and a one-to-many connection from server to PCs.

If the user at PC1 wants to email the user at PC3, the user would create a packet that looks like this:

Source = 1
Destination = 3
Size = 2
Message = Hi
CRC = ...

Notice that the unique PC numbers are used to specify who created this packet and to whom it is intended. Since our star-shaped network connects PC1 only to the server, the packet will arrive at the serve. The server will open the packet to discover that the intended audience is not the server. The server will look at the destination number to see if it is a known number. If it is not known, then it generates an error message; but, if it is aware of the existence of the recipient, then it will forward the packet to the recipient. There is a problem. The server is connected to a hub that only has a one-to-many connection from the point of view of the server. This means that the server needs to broadcast the packet over the local network because it does not have finer control. In other words, all the non-server PCs will receive this packet. All of them will open the packet. Now, if the operating systems on these PCs are honest, PC1 and PC2 will notice that they are not the intended recipient and will delete the packet before it goes any higher up the chain. In other words, the user will not be aware that this packet was present. The story is different for PC3, since it is the recipient. The packet in this case is not deleted; instead, it is passed on to the user for viewing.

The above is actually how it works. If you noticed, the packet is composed of only strings and integers. These are very easy to read, even in binary form. Programs are available to help diagnose network problems; these programs allow you to see and read the packets since they are in text. This is not good if you are concerned about privacy. It turns out to be impossible to stop or prevent people and computers from looking at a packet. The only thing we can do is scramble the letters in the message portion of the packet so that it makes it very hard, or even impossible, to read. Encryption is the Internet's technique of choice to secure your packets.

Above, we have looked at a local area network. This is still not the Internet. We need to step things up and look at the next level of networking known as Wide Area Networks, commonly known as WAN. Figure 1.4 shows a simple WAN. In this network, we have two simple Star LAN networks connected together through an additional PC known as a Bridge. A Bridge is a spatial software program that: (1) knows when a packet is for another network, and (2) can convert the format of a packet in one network to the different format of a packet in the other network (if this is necessary). If the Bridge is connected between the two servers, as it is in figure 1.4,

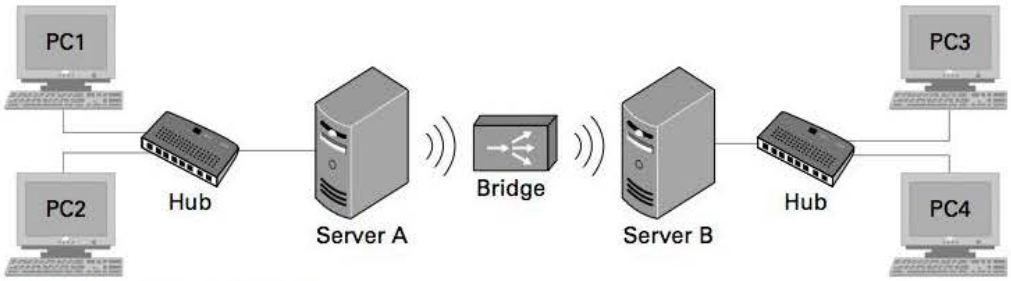


FIGURE 1.4: Wide Area Network

then the server has a choice when it broadcasts its messages. It could broadcast only over its LAN or only through the Bridge, or it could use both routes at the same time. This control can limit the scope of a broadcast and make things a little more private.

We do have a problem we need to address in this WAN. Notice that the PC's ID numbers are no longer unique. The two individual LAN networks' ID numbers overlap. We need to determine a new addressing method. This isn't that hard to solve. What we will do is assign special unique network server ID numbers. This requires the network administrators to have a meeting with each other to pick a unique ID number for their network. Once that is done, they can then format their addresses. They will format it in the following way:

network ID . PC ID

So, if the LAN on the left has the network ID number 00 and the LAN on the right has the network ID number 01, then PC1 from network 00 can send an email to PC3 in network 01 by sending this packet:

Source = 00.1
Destination = 01.3
Size = 2
Message = Hi
CRC = ...

This packet would leave PC1 and arrive at the server. After opening the packet, the server would realize that the intended audience is another network. It would then forward the packet to the Bridge. The Bridge would perform any formatting corrections and then pass it on to the server of network 01. That server would verify that the packet was indeed for the network it manages; it would then check to see if it knows the destination computer. If that computer does exist, it would then broadcast that packet over its local network. The recipient's computer would receive the packet and show it to the user.

This setup is still not the Internet. We need one last piece of the puzzle. You have been introduced to many different kinds of computers. All are PCs, but each of them had different

software operating on them. The user computers were made from a PC running an email program. Then there are the servers and the bridges. The last important computer is the Internet Service Provider, known as the ISP. This computer is a special kind of server that performs two vital services. It allows individuals and servers to register as members and receive a unique ID number, called the IP address. An IP address looks like this: 111.111.111.111. ISPs are connected to other ISPs over a peer-to-peer semi-randomly ordered hierarchical network (see figure 1.5). These ISPs behave like bridges. These ISPs also behave like routers. They are aware of some of the topology of the global ISP network. If they do not know about a particular destination address, they forward the packet to a higher-order ISP that hopefully knows. If it too is not aware, then it also passes the packet to another higher-order ISP until the destination is identified or an error message is returned.

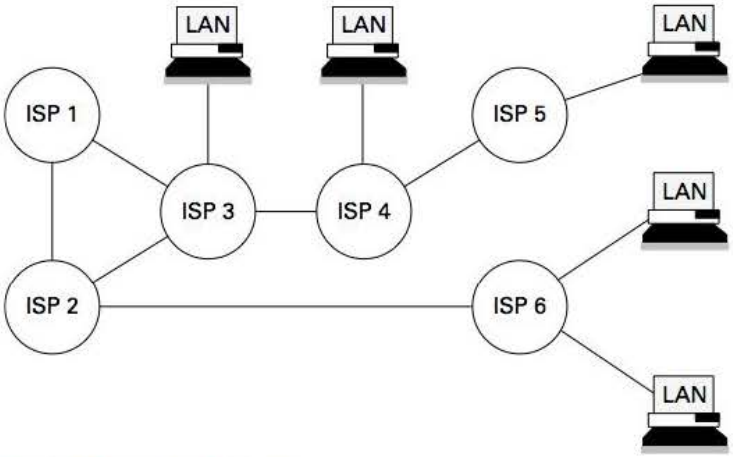


FIGURE 1.5: The ISP Network

So, for example, if you are emailing from McGill University to the University of Tokyo, the email would be sent to an ISP in Montreal that would send it to a higher-order ISP servicing Quebec. This ISP would send the email to an even higher-order ISP servicing Canada. From there, it would probably be forwarded to a North American ISP that would, finally, know about Japan. The process would then go downward through more and more local ISP machines until the recipient received the message in Tokyo.

Now, and this is the important part, as the programmer of the email program, you don't need to worry about any of this. You just need to format the email as a packet and give it to the operating system. This is what a system means. You build your part and trust that the rest was also built well.

RUN-TIME ENVIRONMENTS AS AN EXAMPLE

The environment your program executes within is known as the "run-time environment." The run-time environment is very important. It will affect how you think about your program. For example, how much of the code do you need to write yourself? How much of the code is in a library? Is the library good? Are their tricks you can take advantage of given the hardware connected to the computer? Can the operating system do some work for you? What limitations does the operating system impose on you? Is your network peer-to-peer or client-server? These, and more, are questions that need to be considered before you start writing your program.

The run-time environment consists of those things that can directly impact the way your program runs. Broadly, they fit into the following categories: available libraries, the operating system, the CPU (Central Processing Unit), the peripherals, and the network architecture. The three most important are the operating system, the CPU, and the network. Let us look at each one individually.

Libraries are programs written by someone else that you can include in your program. This means that you do not need to write this code. Instead, you include it in your program and then call it when you need it. This ability makes program development easier. Libraries are known as a cross-over environment. They properly belong to the development environment category since they affect code development and often come packaged with your compiler. Some libraries are extensive and are viewed to be a *framework*. A framework provides to the programmer a new way of thinking, a new paradigm. For example, XNA is Microsoft's game development library. Calling it a library is a little degrading because it provides programmers a "game loop." This game loop runs on its own. It interfaces with the graphics screen and manages (provides hook to) the different stages of a game. To write a game for XNA requires programmers to insert their code into the XNA game loop. Programmers do not have direct control over the game loop. This game loop becomes a run-time environment that programmers must incorporate into their plans. In contrast, JMonkey is a library that provides graphics functionality for Java. JMonkey can also be used to develop cross-platform games, but JMonkey is more properly thought of as a library, because code written using JMonkey is not forced to follow a particular execution framework. Programmers are left to implement their game in any way they want. So, JMonkey should not be called a framework. But, JMonkey is very complete. Programmers can ignore the graphics hardware and can use JMonkey as an abstraction of the graphics hardware. This JMonkey abstraction reflects the run-time environment to which programmers are subject. It affects both how the program will be built and the limitations to the program's abilities. Programmers limit themselves to the functionality and features provided by the library / framework.

Operating systems come in many shapes and sizes, with many names. An operating system's job is to manage the resources of the computer on which it is running. One of these resources

is the user programs. Another resource is the peripherals. The operating system governs access to peripherals and defines rules about what a running program can do. So, the operating system affects the programmer in two important ways: in how it lets your program run and in how it lets your program access peripherals.

Describing the run-time rules an operating system imposes on a program can be summarized using the following terms: *single-user single-process*, *single-user multi-process*, *multi-user multi-process*, *multi-CPU*, *distributed processing*, and *code migration* processing.

A single-user single-process computer only allows a single person to use the computer at any time, and this computer can only execute one program at any time. A dishwasher has a single-user single-process CPU and operating system. That machine can only do one thing at a time. Personal computers in the 1970s and 1980s only permitted a single program to run at any time. Those were fun machines, but were quite limited. A single-user multi-process computer is what we are accustomed to today. Our personal computers behave this way. Only one person can use your personal computer at any moment, but that computer can run multiple programs at the same time. You might be using a word processor and a browser at the same time, while seeing a clock tick on your screen. So your computer is running three programs in total. A programmer on a single-user single- process computer may want to write a program to run a second program, but this cannot be done. That run-time environment does not provide that capability. In the multi-process environment, not only can you run more than one program at the same time, but these programs can communicate with each other (through various means). This, too, is an impressive run-time capability. Programmers in a multi-processing environment can divide their program into multiple programs that all run at the same time and talk to each other

Multi-user multi-process computers are commonly known as servers. A server is a single computer that can allow more than one person to log onto it. Each of these logged in users can run multiple programs at the same time. The server's job is to make sure that everyone is in his own space and not interfering with anyone else. This run-time environment allows the existence of this new class of computer known as a server, but this run-time environment allows programmers to write programs that can communicate with other user's programs. An example of this capability would be to have an electronic white board that is shared between multiple user computers.

A multi-CPU computer is similar to a multi-process computer except that it has more than one CPU. Many computers today have dual-core (2) or quad-code (4) CPU . The run-time advantage we receive now is in being able to truly run more than one program at the same time. Traditional multi-process computers run on machines with only one CPU. This meant that they could not actually run more than one program at any moment because the CPU was physically restricted to one activity at any time. In traditional multi-processors, this problem was overcome with a strategy called *task-switching*. Task-switching is a technique that shares multiple programs with a single CPU. Each program runs on the CPU for a short period of time.

The program is paused, and the next program then runs for a short time. This process would repeat over and over again. Like in the movies where we are watching still pictures changing rapidly, which gives the illusion of motion, here too, we have the illusion that all the programs are running at the same time when they are not. In a multi-CPU machine, we can actually run programs at the same time. But, this is limited to the number of CPUs or Cores in the machine. If you have more programs than the physical Cores, the machine reverts to task-switching. But, in this case, the task-switching is faster because we can task-switch multiple programs at the same time for each physical Core installed in the machine.

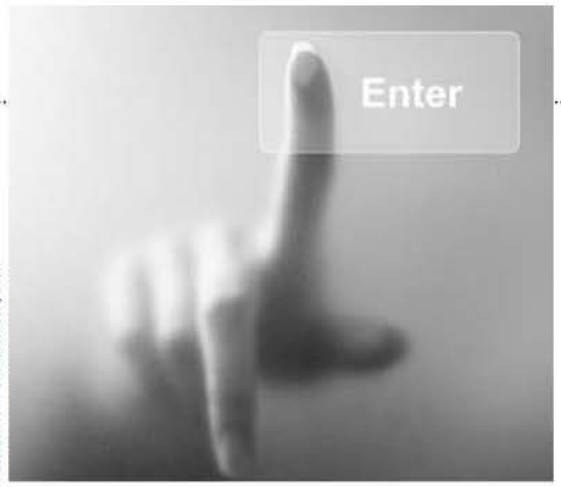
Distributed processing is when a single application is divided into multiple programs. Then, each of these programs is installed on a different machine (possibly at different locations). This single application is attempting to solve a single problem but using many computers, at the same time, with a divide-and-conquer strategy. For example, SETI (Search for Extra Terrestrial Intelligence) is using the Arecibo Radio Telescope to search for radio transmissions from space. This radio telescope is operating at a capacity that is greater than the amount of computing power the SETI servers can handle. They came up with an interesting distributed-processing solution. They would create pretty screen savers that regular people could download onto their computer. When these people are not using their computer, the pretty screen saver would turn on and contact the SETI servers. It would download a packet of data. Then, when you were not using your computer, it would do work for SETI. When your computer fully analyzed the data from the packet, it would return a result back to the SETI servers. SETI increased their computing capabilities dramatically.

Code Migration processing is new and experimental. This is the future of computer programming. Imagine double clicking on an application. The program launches and discovers that your computer does not have enough resources to run. Maybe you don't have enough memory, or CPU power, or anything else you can imagine. Instead of the operating system stopping the application and displaying an error message, the operating system chops up your program into pieces and, using the network/Internet, looks for other computers with available resources. It then transmits parts of your program onto those machines. Now your application exists in many places. (This is not the cloud. The cloud is a distributed system.) Code migration's programs exist as single application, but in pieces across a network invisible to the users who have their computer resources being shared. If someone closes their computer or needs more resources, the migrated programs automatically redistribute themselves. The system would only generate an error if were no additional resources were found.

As you can see, run-time environments are also systems. The programmer can write the program without needing to know, for example, how code migration was being implemented exactly. The programmer would only need to understand how to use the run-time environment. The program would take advantage of the features provided by the run-time environment's many sub-systems.

CHAPTER TWO

Understanding Unix



It all began because of a video game, if you can believe it. In the late 1960's, Ken Thompson wrote a computer game called Space Travel designed to run on a newly developed state-of-the-art operating system called Multics. AT&T Bell Laboratories, General Electric, and MIT were developing Multics, and they needed a program that would show off the new operating system's multiprocessing capabilities; but, the project failed. As the story goes, the executives could not imagine why someone would need a computer to run more than one program at a time. After the project failed, with nothing to do in 1969, Ken enlisted Dennis Ritchie (inventor of C) to help convert the game to run on a Bell Labs PDP-7 minicomputer. This turned out to be a difficult thing to do. The PDP-7 needed an operating system, so Ken built one. It was a simple operating system that had many features from the failed Multics project. They decided to name the new operating system Unics, as a spoof on Multics. Employees at Bell Labs liked the little operating system and, in 1971, asked Ken and Dennis to write a word processor that would run on Unics for the PDP-11. Dennis created the C Programming language to help in the conversion of Unics from PDP-7 assembler into the new and portable C Programming language. This was then compiled on the PDP-11. The new word processor was also a success. Now both the C Programming language and Unics were gaining followers. Since AT&T was prevented by laws from selling their in-house operating system invention they made it available at no cost to anyone who wanted it. Over the years, Unics became known as Unix.

This free version of the operating system was very crude, but it became a big hit at universities and research institutions. During the late 1970s and early 1980s these universities and research institutions took the free source code from Bell Labs and created new and improved versions of the operating system. By this time, the operating system was known as Unix. Each new version was given its own special name: Berkeley Software Distribution (BSD 1982), Microsoft's Xenix (1980), SCO Unix (1993), and Linux (1991 by 21-year-old Linus Torvalds), to name a few.

Writing code that would be able to run on all these versions of Unix was becoming difficult. Between 1983 and 1988, the Institute of Electrical and Electronics Engineers (IEEE) developed a series of standards called Portable Operating System Interface (POSIX) that was intended to help control the different Unix run-time environment. The latest attempt in standardizing Unix was ratified by the International Organization for Standardization (ISO) in 2003. This combined the latest IEEE 1999 POSIX specification with the Open Group task force Single Unix Specification.

THE UNIX OPERATING SYSTEM

Every operating system is composed of many components. These components, working together, are called a system. Unix is a no-nonsense program that maintains a keep-it-simple philosophy. Many modern operating systems, like Microsoft's Windows, employ a complex multilayer-with-bells-and-whistles approach. Unix, on the other hand, builds its system on four simple building blocks. (See figure 2.1.)

- The **Kernel** is the core operating system module. It resides in the computer's random access memory (RAM) for the entire time the computer is powered up. Its job is to manage how memory is organized, how a process executes, and how communication is maintained between peripherals and software. It provides the mechanisms for multitasking.
- The **Shell** is the interface between the user and the operating system. It allows the user to interact with the operating system using two techniques: the command-line prompt or a windowed environment. It passes the user's requests to the operating system. The user must learn the shell's syntax (the way to communicate with). The shell is loaded into RAM when a user logs in and removed from RAM when the user logs out.
- The **File System** manages information stored on secondary storage. Secondary storage includes any memory not part of the computer's RAM, for example: hard disks, tape drives, CD and DVD drives, and memory sticks, to name a few. The information is organized into structures called **files** and **directories** (also known as folders), which are assembled into memory units called **blocks**. A secondary storage device has fixed number of blocks. These blocks are created when the device is formatted. The larger the medium, the more blocks it has. Files contain data, while directories group related files together. Files and directories are created by users. User's use files and directories as a method of organizing their information and programs. Operating systems come with their own default files and directories when installed that define the operating system's complete set of utilities and its run-time environment. The top (or main) OS directory is called the **root** and is designated by the forward slash (/) in Unix. Common Unix directories

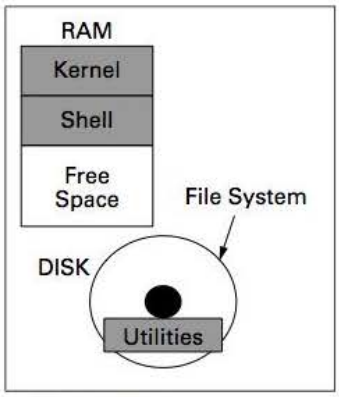


FIGURE 2.1: Unix Components

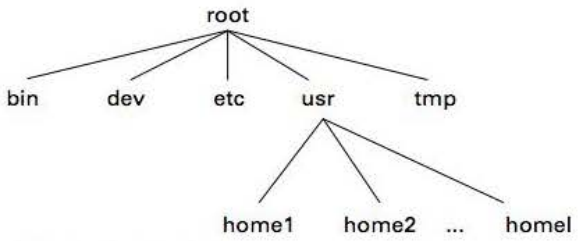


FIGURE 2.2: Standard Unix Directory Structure

found below the root directory are the following: bin, dev, etc, usr, and tmp. Other directories can also exist, created either automatically by the particular installation of the operating system or by the system administrator or the users; see figure 2.2.

- The **bin** directory (or /bin) contains all the installed utility software and other specially installed programs that can be shared by all the users. This is like the Program Files and Windows folders in MS Windows.
- The **dev** directory (or /dev) contains all the device drivers and information needed to maintain all the system peripherals. This directory is like the Windows\System folder in MS Windows.
- The **etc** directory (or /etc) stores administrator files like password files.
- The **usr** directory (or /usr) is the location where all user data is stored. Each user is given an individual **home directory** below /usr. The home directory name is normally the same name as the user's login name. The user can set permission access to the files and directories they create. Permissions are broken down into three categories: private, shared within a specified group of users, and public to all users who have access to the system.
- The **tmp** directory (or /tmp) is used to store temporary files.
- File and directory names can have any alphanumeric values, including the underscore (_), the period (.), and the comma (,). If a file name begins with a period, then it is considered to be a hidden file and is not displayed when the user asks to see all the files in the directory. A special switch must be set to view these hidden files when listing the directory files.

- The **Utilities** subsystem exists in two forms: the first form are independent programs stored in the /bin directory that perform additional operating system functions. They are often called operating system **commands**, but they are not; they are C programs. Examples of these would be the **ftp** or **rlogin** commands. The second form are programs also stored in the /bin folder, but they do not belong to the operating system. They are supplied by third-party vendors. These are often called **tools**. Examples of these would be word processors or database applications.

THE UNIX SHELL

The operating system is a key component of every computer. Without it, programmers and users would have to communicate directly with the hardware, and hardware only talks binary. This difficulty was quickly resolved by building special software that converts the user's instructions into binary. This special software is called the operating system *shell*. The shell accepts instructions typed in by a keyboard or a mouse; it would then map that into functions that the computer would understand. The shell's execution cycle is depicted in figure 2.3.

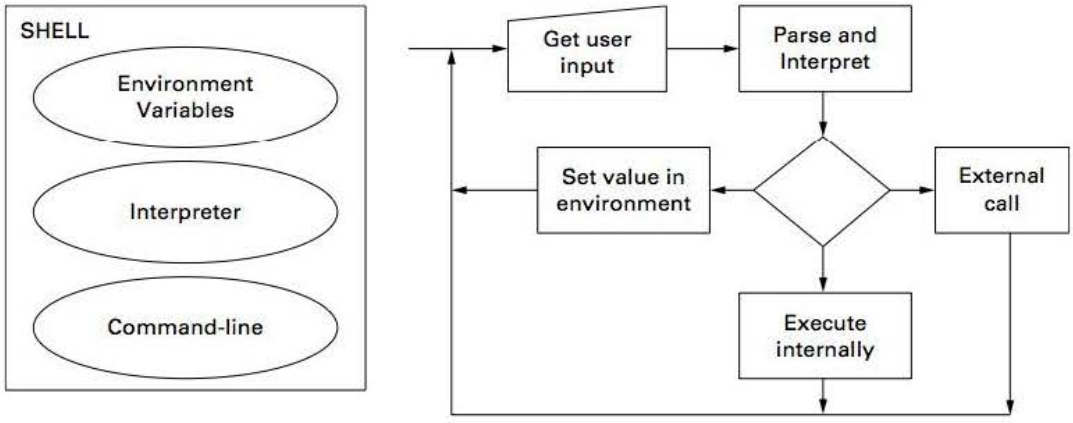


FIGURE 2.3: The OS Shell and Execution Cycle

A shell is composed of three subsystems: the command-line processor, the environment memory, and the interpreter. The command-line processor is the primary user interface (UI). This UI could be a text-based command-line prompt, or it could be a windowed user interface. In either case, they behave in a similar manner, but we will focus on the text_based command line.

The command line displays a prompt and waits for the user to type a command. The command is sent to the shell's parser, whose job it is to figure out what was input. Based on that determination, the shell will attempt to carry out the user's request. If the shell knows how to do the requested task, it will call an internal shell function to perform the task and return the result

to the user by displaying the result on the screen, saving it to a file, or sending it out some data port (as per the user's request). An example of an internal request occurs when modifying the shell's environment variables.

If the shell does not recognize the command, it then attempts to find a program external to the shell that matches the name of the command. These programs are either operating system utilities or third-party programs. The shell uses a default search strategy to locate these external programs. The shell's default search strategy begins with the **current working directory**. The current working directory is the directory you are working in at that time. It is searched first. If that fails, the shell's environment memory maintains a variable called **PATH** that defines other directories the shell can use to locate the external program. If the command, program, or file was not found using the PATH, an error message is displayed, and it stops searching. Users can modify the PATH variable to remove or add addition directories. We will talk about this soon.

A shell can also accept commands from a text file. This means you can input commands manually at the prompt, or you can put commands in a text file and have the command line "run" the text file. This is handled by the shell's **interpreter**. The shell's interpreter understands a **scripting language**. This language allows you to write any regular command-line command plus programming-like statements, like conditional and iteration statements. In some cases, subroutines are also possible. These text files are known as **scripts**.

Script files come in two types: user and system. **System scripts** have reserved file names. At special times during your interaction with the operating system, these scripts may be invoked automatically by the operating system. **User scripts** are unknown to the operating system. They will only be invoked when the user specifically requests it. Example system scripts are known as start-up scripts, login scripts, scheduling scripts, and logout scripts. Start-up scripts are executed automatically when the computer is turned on. Login scripts are executed automatically when a user logs in. Logout scripts execute when the user logs off. Scheduling scripts execute automatically at some hour on the clock. Example user scripts can be: backup scripts, compiling scripts, and program setup scripts. The user must invoke these scripts specifically.

THE UNIX SESSION AND COMMAND-LINE INTERFACE

Unix creates a special environment in its memory to track the events and current state of a user who has logged into the operating system. This is known as a Session. It begins when you log in, and it is deleted when you logout. Sometimes, the System Administrator turns on the operating system's history logging feature. This causes the operating system to automatically record information about your session when you log out. Information includes: who accessed the computer, when it was accessed, and if the user did anything important or improper.

Starting a Unix Session

To begin a Unix session, the user must first log into the system. This is a very simple process. Immediately after the computer is turned on, the screen will display the following prompts:

Login:
Password:

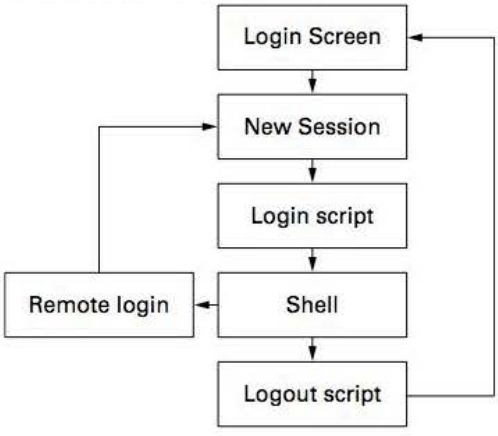
Users must enter their public user name at the Login prompt and their private password at the Password prompt. The system will permit you to make three errors before it logs the failed attempt to access the Unix system. If the user name and password exist, the user is recorded as entering the system and is sent to their home directory. Then, the default OS shell is started.

```
Welcome to Unix
Bash version 2.05
Wed Sep 10 2008 08:10:05 EST
$_
```

FIGURE 2.4: The Unix Command-Line Prompt

If a login script is present, it is executed. Then the command-line prompt is displayed (figure 2.4). Figure 2.5 shows the login session flowchart. Unix has more than one shell, but we will talk about that later.

FIGURE 2.5: Unix Login Session



A common command-line prompt is the dollar sign. So, you would see something likefigure 2.4.

The dollar-sign prompt would have a flashing cursor beside it. I have tried to depict the flashing cursor with an underscore to the right of the dollar sign, in figure 2.4. Once you see an interface like this, you know that you are in the operating system's shell. This command-line prompt is waiting for you to enter a command. You could enter the command `logout` to return to the login screen (see figure 2.5). If a logout script is present, it is executed first before you return to the login screen.

If the computer you are using has a windowed interface, there will be an icon called Terminal or Command-line either on your desktop or in the Applications/Accessories menu. Clicking it will give you the same welcome and command-line prompt described above.

Entering Commands at the Command-Line Prompt

The default command-line prompt in Unix is often the per cent sign (%) or the dollar sign ($), depending on the shell. It is easy to use. You simply type in a command beside the prompt and press enter to tell the computer to car ry it out. The only hard part is memorizing all the commands. But Unix gives you some help. We will look at that soon. At the prompt, all commands are entered using the following syntax:

```
[PROMPT] COMMAND {SWITCHES} {ARGUMENTS} [ENTER]
```

Where:

- **[PROMPT]** is the current prompt displayed by the operating system, like % or $.
- **COMMAND** is any Unix oper ating system comm and, script interpreter instruction, or executable program or executable script name.
- **SWITCHES** are special par ameters that modify how the comm and will execute. They are optional; hence, the curly br ackets in the abo ve syntax description. The syntax of a switch is always a dash followed by a flag (a character). For example, **–a**, is a switch. Its meaning depends on the command it is being used with. For example, **ls –a**, lists all the files in a directory. The -a includes all hidden files.
- **ARGUMENTS** are additional parameters that are required by the command. It is also optional. This is different from switches. Switches are used to alter the beh avior of a command. Arguments are extra information that the comm and would nor mally prompt the user for when executing. N ote that not all comm ands will prompt the user. Some will simply display an error message indicating that some information was missing. There is no syn tax for argumen ts, other than that they are not per mitted to have the same for mat as a sw itch. For example, **cal 2011** is a utility th at displays a calendar. The utility needs the user to specify a year to displa y the calendar. The year is the argument.
- **[ENTER]** is ho w the comm and is ter minated. This is per formed by pressing the **enter** key on the keyboar d. This tells the shell to pr ocess the command. If it is not a shell interpreter command, the shell passes the comm and to the oper ating system

for execution. If the OS cannot execute it , then an error is displayed, followed by the prompt (once again).

Unix Command-line Commands

This section lists, in tabular form, some commonly used commands. This list is not exhaustive. Instead, it presents popular and commonly used commands. The special Unix command **man** (i.e., manual; this is the Unix help command) can be used to find additional information about every command in Unix. Each command is described in the **man** manual in detail. Each command contains many switches and arguments, too many for the scope of this text. It is recommended that after you are familiar with using the commands as described here, you should then access the **man** utility to learn more about those commands you would like to use further. For example, the **ls** command lists all the files in the current directory. If we want additional information about the ls command, at the prompt, we can do the following:

```
$ man ls
```

The above command-line input will ask the man program to display information about ls. This will be displayed on the user's screen, page by page, and scrollable. Try it out!

Below is a table of useful Unix command-line commands. It is not a complete list, but it contains many useful commands for first-time users. A small subset of important commands will be described in detail following this table. Note that this subset of important commands is representative of how all Unix commands operate. Once you know these representative commands, then you will be able to figure out the rest on your own with help from the **man** command.

BASIC MANAGEMENT OF YOUR ACCOUNT		
Description	**Syntax**	**Notes**
Changing your password	passwd	The user is prompted for additional information.
Get your email	mail	The user is prompted for additional information.
Get news	news	The user is prompted for additional information.
Exiting Unix session	logout	The logout script is executed if present.
Exiting a Shell	exit	
Exiting Unix or a shell	CTRL-d	Where CTRL is the control key on the keyboard. Both the control key and the lowercase d key are pressed at the same time.

SYSTEM INFORMATION		
Description	**Syntax**	**Notes**
Get the current date	date	

SYSTEM INFORMATION		
Description	**Syntax**	**Notes**
Who is currently logged in	who	
Getting help	man TOPIC	TOPIC is a keyword to search.

COMMUNICATION COMMANDS		
Description	**Syntax**	**Notes**
Talk to a logged on user	write USERID MESSAGE	Where USERID is the user's login name, and MESSAGE is a simple one-line text message.

FILE-PROCESSING COMMANDS		
Description	**Syntax**	**Notes**
View files in directory	ls –l –a	LS does not show hidden files, –l shows files with full statistics (i.e., long form), and –a shows hidden files (i.e., all files).
View a specific directory	ls PATH –l –a	Where PATH has the following syntax /dir1/dir2. Use the forward slash to separate the directory names. Indicate the full path either from root or current directory. The –l and –a switches work here as well.
Move a file	mv FROM TO	FROM is the path and file name of the file you want to move and TO is the path and new file name that will be given to the file.
Copy a file	cp FROM TO	Same as move
Erase a file	rm FILENAME	Where FILENAME is the path and name of the file.

DIRECTORY COMMANDS		
Description	**Syntax**	**Notes**
Changing directory	cd PATH	Where PATH is the forward slash path from either the root or current directory to the destination directory.
Which directory am I in?	pwd	
Go to the root directory	cd /	
Creating a new directory	mkdir PATH/DIRNAME	Where PATH/ is optional and specifies the directory path to the directory where the DIRNAME directory will be created. If PATH/ is not provided, then the current directory is assumed.

DIRECTORY COMMANDS		
Description	**Syntax**	**Notes**
Deleting an existing directory	rmdir PATH/DIRNAME	Similar to mkdir description.

PROCESS COMMANDS		
Description	**Syntax**	**Notes**
To run a program	PRONAME [ENTER]	PRONAME is the filename of the program you want to run. [ENTER] executes.
To multiprocess	PRONAME & [ENTER]	Adding the ampersand (&) executes the program PRONAME and then displays the command-line prompt for you to enter another command. This step can be repeated. Each program is multi-processed.
To see your running processes	ps	Displays a table of all the running processes and their Process Identification Number.
To terminate a running process prematurely	kill -9 PSNUMBER	PSNUMBER is the executing ID number the operating system uses to identify a running process. This ID number can be obtained from the output of the ps command. The -9 switch is optional and terminates problematic programs.

MISCELLANEOUS COMMAND		
Description	**Syntax**	**Notes**
Display a calendar	cal YEAR	YEAR is the year to view
Redirect output to a file	cal YEAR > FILENAME	The redirection symbol (>) will redirect the output from cal to the text file FILENAME. The text file will be created. It overwrites any existing file with the same name.
View a file	cat FILENAME	
View a text file page by page	more FILENAME	
Redirect to a program	cat FILENAME | more	The pipe symbol (|) takes the output from one program and sends it as input to another program.
Print a text file	lpr FILENAME lp FILENAME	
Concatenate to a file	cal YEAR >> FILENAME	The concatenation symbol (>>) redirects the output from a program to a file called FILENAME. If the file already exists, then the new information is appended to the end of the file; otherwise, it creates a new file.

SPECIAL KEYBOARD KEYS		
Description	**Syntax**	**Notes**
To logout	CTRL-d	
To pause output on screen	CTRL-s	Program must be writing to the screen for this to work.
To resume output on screen	CTRL-q	The process resumes execution after a control s is issued.
To terminate output	CTRL-c	Program must be writing to the screen for this to work.

CHANGE FILE SECURITY		
Description	**Syntax**	**Notes**
Changing file security	chmod SWITCHES FILENAME	Files can be assigned security privileges. (See text for details.)

Detailed Command-line Command Descriptions

Unix File Commands

Unix uses two basic file types: **data files** and **executable files**. Each of these types of files can be in either of two for ms: **binary files** or **text files**. Users not familiar with this capability will find it strange that an executable file could also be a text file; but, we will see later how scripts use this ability (being both executable and text files).

A text file contains only ASCII (or UNICODE) characters. ASCII is a special standardized 8-bit encoding of characters. (UNICODE is 16-bits.) This standard encodes 256 different characters (like letters, digits, and symbols in AS CII) with their own unique bin ary code n umber. This means that every piece of information in a text file is an 8-bit binary code specifically encoded in the manner outlined by ASCII. ASCII encodes the entire standard alphanumeric and type-writer keyboard character keys plus some specialit y processing keys like the En ter, Delete, Backspace, and Escape keys. Included in ASCII are foreign characters like é and ô. Unix's Shell is also designed to understand and use text files as scripts.

Binary files are data files that do not have a predefined binary format. Data in a binary file can be 8, 16, 32, 64, or larger bits long. Each da ta item in a bin ary file can be of different binary length. The type of infor mation in every data item could be AS CII, binary arithmetic, pixels, or any other format. The Unix shell does not kno w how to interpret binary files. The shell will normally send a binary file to the operating system for execution. Binary files are assumed by the shell to be operating system–specific drivers, CPU executable programs, or data files. If this

assumption is incorrect, then the operating system will return an error message. The shell will then display the error to the user.

Data files are files that contain only information in the form of facts and text. Data files cannot be executed. They can only be read from and written to. An example would be a database, a phone book, or a list of valid users.

Executable files come in two forms: **text-file executable** and **binary-file executable**. Text file executable files are script files. Script files are special text files that contain operating system command-line commands and script programming commands. The shell interprets and executes these text files. Each instruction is placed on a separate line of the file, like when you do it at the command line. You write the single command and then press enter. You do the same in a script file. The shell extracts each instruction and tries to execute it at the command line using the shell's interpreter.

Binary executable files are operating system drivers, libraries, or compiled programs. A programmer using a compiler can create these files. The shell does not know how to execute these files, and therefore, sends them to the operating system for execution.

File Names and Extensions

File names in Unix can have a file extension or be extension-less. The general file name syntax is as follows:

NAME.EXTENSION

Where:

NAME is any alphanumeric string plus _ and $
.EXTENSION is optional and if present must be alphanumeric and/or _ and $.

NAME and EXTENSION can be of any length.

File extensions are used to group common files together. For example, .DOC could designate all your word processing files and .TXT could designate text files. Then you could ask the OS to show all your .TXT files or all your .DOC files. Other than that, Unix does not consider file extensions. It is present solely for the convenience of the user. For example, if you used the file extension .EXE to identify a program as being executable, then type the file name without the extension Unix would return an error. More specifically, imagine you have a program named WORD.exe and you type WORD without the extension (as in MS Windows) at the prompt thinking that the OS would understand that the program is executable and would automatically start WORD.EXE. This is not the case. To run the above example, the user must enter WORD.EXE in full to get the program running. Because of this, you will find that Unix developers tend not to use file extensions for programs.

Valid file names: letter5 letter.txt letter.$$$ _letter.doc .letter

The ls Command

The list command displays all the files in your current directory. In its simplest form, you simply type **ls** at the command-line prompt and it displays all the files; for example:

```
$ ls
letter.txt word.exe index.html
$
```

Notice in the example abo ve, the **ls** command was typed beside the pr ompt and the output was displayed horizontally with a single space separ ating each file name. This pattern would continue and wrap around the screen if there were more file names. This series ends with the prompt once more.

The **ls** command has many switches. Use **man ls** to find out about all its sw itches and arguments, but two important switches are **-a** and **-l**. The **-a** switch displays all hidden files, and is often called the "all" switch. In Unix, any file without a file name having only an extension is considered to be a hidden file. For example:

```
$ ls -a
.cshrc letter.txt word.txt index.html
$
```

The above example is identical to the previous **ls** example except that we have a new file, .cshrc. This file has no file name. It only has an extension. It is considered to be hidden. The **ls** command on its own did not display that file; we needed the **-a** switch.

The **-l** switch is known as the "long" switch. This switch displays the file names in a long format. Note that we can combine switches, like **ls -l-a** ,to show "all" the files in "lon g"format. Example:

```
$ ls -l
-rw-------   1 bob -           1000    2011-10-01 12:15   letter.txt
-rw-------   1 bob team        345     2011-01-25 14:30   word.txt
-rwx-----x   1 bob students    120     2010-05-29 02:10   index.html
$
```

The same files are displayed in this example as when we did the **ls** command on its o wn. In this case, the files are displayed vertically with details. Every Unix file has "meta" information, information that tells us about the file. In the above case, we see the following:

`-rw-------`	file type + access privileges (we will talk about this soon)
`1`	file links
`bob`	owner of the file
`-`	group name file is shared with (in this case the file is not shared)
`1000`	size of file in bytes

`2011-10-01`	the date file was last updated
`12:15`	the time file was last updated
`letter.txt`	file name

The file type can be either dash (-) or the letter " d." Dash means the file name is a file. A "d" means the file name is actually a directory name. File links indicate the number of shortcuts pointing to this file. At file creation time, it will say 1 since the real file name points to the file. After that, you can create multiple "soft" and "hard" links to the file. The number changes, reflecting the shortcuts. The owner is the login user name of the individual who has owner's rights to the file. Normally, the person who creates a file becomes the owner automatically, but you can transfer ownership to another user. You can share your file with members of a group using the **chgrp** command. A dash (-) indicates that a file is not being shared. Group names are created by the system administrator, and these names can be in the form of a word or a number. Users become members of a group by asking the administrator to include them. Then the owner of the file can use **chgrp** to assign a group name to their file. The rest of the "long" information is intuitive. A discussion about **chmod** will address the access privileges, further below.

File Wild Card Characters and the Multiple Execution Operator

Unix commands can be further customized using "wild card" characters. The asterix, *, is used to match all occurrences of a string having any length. The question mark, ?, matches any occurrence of a single character. The square brackets, [ ], represent optional combinations, like an or-statement. This may be best described by using an example. The command **ls** is used to list the files within a directory. It can also be used to list a particular file in a directory. For example, the user can type **ls stuff.txt** to see if the file stuff.txt exists in the current directory. But, we can do more:

$ ls *.txt	Will list all the files that have the file extension .txt.
$ ls stuff.*	Will list all the files that begin with stuff but have any extension (even no extension)
$ ls *.?xt	Will list all files with file extension xt proceeded by any single character
$ ls *.[ab]xt	Will list all files with file extension axt or bxt.

The multiple execution operator, semicolon (;), permits the user to enter many commands on a single line, and they will get executed at the same time. For example:

```
ls; who
```

Will display the files in the current directory in short format while at the same time, displaying the user names of all the users currently logged in.

The ampersand (&) is also a multiple execution operator. Instead of writing all the commands on a single line, you write one command and terminate it with the ampersand and then press the enter key. This will execute the command, but it will also display the command-line prompt immediately, regardless of the run-time state of the previous command. You can now input an

additional command. If you terminate it with ampersand, you will be presented again with another prompt. All the commands are executing concurrently. The last command you input without the ampersand also launches the program concurrently, but a command-line prompt is not displayed until that last program terminates.

File Redirection Operators

The command-line prompt permits three additional ways to manipulate commands and files: redirection, pipes, and concatenation. Concatenation is a special form of redirection.

Redirection takes the output produced by a command or program, which normally is directed to the screen, and instead, stores it in a file. This only works with output that is directed to the screen. The greater-than operator (>) is used to redirect the screen output to a text file. The syntax is as follows:

```
[PROMPT] PROGRAM {SWITCHES} {ARGUMENTS} > FILENAME [ENTER]
```

Any command, with its set of switches and arguments, can be entered followed by the greater-than symbol and the name of a file. Any output directed to the screen by the program will be instead stored within the text file. If the file does not already exist, it will be created. If the file already exists, it will be overwritten. For example:

```
$ ls -l > test
```

The command **ls** lists the file names of all the visible files in the current directory with all the meta information for each file (-l). Normally, this would display the information on the user's screen. But in this example, the output will not be displayed; instead, the output is stored in a text file called *test*.

A **Pipe** is another form of redirection that takes the output destined for the user's screen and instead sends it to another program as input. The pipe symbol is the bar (|). The syntax is as follows:

```
[PROMPT] PROGRAM {SWITCHES} {ARGUMENTS} | PROGRAM {SWITCHES} {ARGUMENTS}
```

This operates similar to the redirection operator except that the destination is not a text file but another program. That program can use any of its SWITCHES and ARGUMENTS as well. The piped output from the first process is sent through stdout. It is then given to the second program as input through stdin. The data sources stdin and stdout are Unix-defined information sources. The stdout source represents the screen, and stdin represents the keyboard. Information from stdin, for example, can be read by a program using its standard programming keyboard reading commands. In C, this could be the commands `scanf` of `gets`. For example:

```
$ ls -l | more
```

The screen output from the directory listing command is piped to the program **more**. The program **more** displays the directory one screenful at a time. Once the screen has been filled with

information, the output pauses. The user presses the space bar to get the next screenful. In this way, if the directory listing is too long to fit on a single screen, it can be displayed one screenful at a time.

The next shell operator is **concatenation** represented by the double greater-than symbol (>>). This is a special case of the redirection symbol. It functions in the same way as the redirection symbol. The only difference is the method by which the destination text file is handled. In this case, if the file does not exist, the file is created and the output from the screen is stored in that file instead of being displayed on the screen. If the file already exists, the file is not overwritten. Instead, the file is opened and the new data is appended to the end of the file. The resulting file contains all the previous information first, followed by the new information as the last part of the file.

The last shell operator is the **input-redirection** character represented by the less-than symbol (<). Any program that would normally read (get input) from the computer's keyboard can be redirected to get its input from a text file. For example:

```
$ PROGRAM < FILENAME
```

would cause `PROGRAM` to read from `FILENAME` when it was originally supposed to get its information from the user through the keyboard.

We can also combine things:

```
$ PROGRAM < FILENAME | PROGRAM2 > FILENAME2
```

The above example executes `PROGRAM`, which receives input from `FILENAME` instead of the keyboard. The output is sent to `PROGRAM2` for further processing. The output that `PROGRAM2` procedures is stored in `FILENAME2` and not displayed on the screen.

You are free to mix wild card characters and redirection in any Unix commands.

Unix Directories

Unix directories are like the folders you find in Windows. They are objects that contain files and other directories (called subdirectories). The common use for directories is the organization and grouping of related files. Directories can be created, deleted, and have security permissions assigned to them. You can put files into a directory or remove them from a directory. Directories can have subdirectories within themselves, recursively to any depth.

The commands that affect directories are the following:

Create a directory	`mkdir NEW_DIRECTORY_NAME`
Delete a directory	`rmdir EXISTING_DIRECTORY_NAME`
Go to a directory	`cd PATH`
Copy a file into a directory	`cp FROM_PATH TO_PATH`
Move a file into a directory	`mv FROM_PATH TO_PATH`
In what directory am I in?	`pwd`

Other special "go to directory" commands:

Go to the root directory	`cd /`
Go up one directory	`cd ..`

Unix PATH syntax

The Unix shell always assumes a path. If you do not specify a path, then the current directory is assumed. The path is defined to be a list of directory names, each separated by the forward slash terminating, optionally, with the name of a file. All parts of the path are optional; therefore, you can write a path with only a filename present. You can also write a path with only directories and no terminating filename. What is required is that it makes sense within the context of the command. For example: `/usr/jsmith/letter.doc` says that a file called `letter.doc` exists with the directory `jsmith`. The directory `jsmith` exists with the directory `usr`, which is contained within the root directory.

If you provide a filename only without a directory list, the shell will assume you are referring to the current directory. If you provide a directory list and no filename, it will assume that you have provided the filename already somewhere else. If neither of these assumptions is true, the shell will display an error and nothing will be executed.

The shell assumes the root is named / (forward slash). If the path does not begin with the forward slash, then the path is assumed to begin from the current directory. For example: `data/file.txt` says that a file named `file.txt` exists within the directory `data`, which exists within the current directory.

Therefore, you can write paths from two points of view: starting from the root or starting from your current directory. Here are some examples of paths:

Full path	`cp data/file.txt /usr/jim/backup/new.txt`
Path without a filename	`cp data/file.txt /usr/jim/backup`
Path with filename only	`cp file.txt /usr/jim/backup`

In the full path example, we are copying a file from `data/file.txt` to `/usr/jim/backup/file.txt`. This says that the from-path is in the current directory's point of view since the path did not begin with a forward slash. This says that the current directory has a subdirectory called `data` and within that subdirectory is a file named `file.txt`. The to-path is in the root point of view. It says that `file.txt` will be copied into a directory named `backup` that exists within a parent directory called `jim`, who has a parent directory called `usr` that is in the root directory. The from-filename `file.txt` will be renamed in the to-directory as `new.txt`. It will contain all the same information; only the name of the file will change. We could have kept the same filename, but we decided to make a more interesting example.

The second example copies `file.txt` into the same `backup` directory. In this case, we did not define a to-filename. The shell will assume that we provided it somewhere else. Where could

this be? The shell will only look within the command you entered or the shell's memory. To look within the shell's memory, we would have had to use a shell variable name within the command. We have not, so the filename must be in the command. The only filename present is the from-filename. The shell will assume that one. So, in this example, we copy the file to the same directory and we keep its original file name.

The last example from-filename has no directory list. It assumes then that the file is in the current directory. This means that we must already be in the directory called `data`, for example. This is unlike the other examples where we had to have been in `data`'s parent directory.

Paths can be fully mixed together with redirection and wild card characters.

CHMOD (File Security Privileges)

Unix contains a special utility program that allows users to modify the security privileges of files and directories the user owns. For regular users, this would be only files they created themselves within their own home directory. For system operators who have a higher **security level**, this would include other directories or even the entire system. Every file and directory has three **access rights** and three **security levels**. The access rights are: **read, write**, and **execute**. A file with the read access right can be opened for viewing or used as input to a program. A file with the write access right can be created, appended to, overwritten, and a program can save information to it. A file with the executable access write can be executed by the shell interpreter if it is a text file or by the operating system if it is a binary file. The security levels are: **private, shared**, and **public**. A file or directory designated as private can only be read/write/executed by the owner of the file or directory. Shared files and directories can be read/write/executed by the owner and by those users designated as being in the same security group as the user who owns the file or directory. Public access means that all users can read/write/execute the file or directory.

The security levels and access rights can be combined in any combination. Therefore, a file can be designated as public but read-only. In this case, all users can read, view, and use the file as input to a program; but, they cannot write to the file, change its contents, or execute the file.

The Unix command used to assign security levels and access rights is **chmod** (change mode).

```
chmod SWITCH FILENAME
```

Where:

SWIT CH specifies the security level and access right for a file or directory
FILENAME is the name of the file or directory. It can be preceded with a path

Switches come in the following forms:

LEVEL + ACCESS	• this means give access
LEVEL – ACCESS	• this means take away access
LEVEL = ACCESS	• this means overwrite privileges to the new access

Where:

LEVEL is the security level assignment for the file or directory
ACCESS is the file access assignment for the file or directory

Where:

LEVEL can have the values:
a for public (all)
u for private (user), and
g for shared (group)
ACCESS can have the values:
r for read-only
w for write-only
x for execute-only
These can be concatenated together to give, for example, read/write privileges.

For example:

```
$ chmod a+rwx letter.doc
```

The above example will make the file `letter.doc` publicly accessible for reading, writing, and executing by any user currently logged in. If the file had any other privileges, they are not disturbed.

Another example:

```
$ chmod g-x letter.doc
```

In this example, users in the shared group are no longer permitted to execute letter.doc. It does not change any other privileges. So, for example, if the group was permitted to read the file previously, then it retains this privilege.

History

The shell's memory can be used to remember the most recent commands the user entered at the command-line prompt. This ability is useful since some commands are very long, and if the user wants to repeat the command multiple times, it would be nice that there was a way to do that quickly. The portion of the shell's memory reserved for the recording of the user's activity is called the *history*. The history can be accessed through four standard commands. At the command-line prompt, the user can input the following commands:

`history`	All the commands you recently entered are listed with ID. numbers
`! IDNUMBER`	Execute one of the commands you recently entered using the ID.
Up arrow	Display previous command (iterate back through history, each time).
Down arrow	Display future command (iterate forward through history, each time).

For example:

```
swing-shift:/var/log/httpd> history
  1  23:55  pwd
  2  23:55  pushd /var/log/httpd/
  3  23:55  ls -l access_log
  4  23:56  tail -1000 access_log | grep "index"
  5  23:56  tail -1000 access_log | grep "index" | wc
  6  23:57  tail -1000 access_log | grep "Diagonal" | wc
  7  23:57  ls -l
  8  23:57  cd jtidwel lnet/
  9  23:57  tail -1000 access_log | grep "index" | wc
 10  23:58  tail -200 access_log | more
 11  23:58  tail -200 access_log | grep "google"
 12  23:58  cd ..
 13  23:58  tail -1000 access_log | grep "google"
 14  23:59  tail -1000 access_log | grep "googlebot"
 15  23:59  history
swing-shift:/var/log/httpd>
```

At the top, the user typed in the comm and `history`. Then, 15 commands are listed. Notice that the 15th command is the `history` command just entered. The prompt is then displayed. The user could now type !10 follo wed by pressing enter to execute the comm and at line 10. Or, the user could have pressed the up arrow six times to scroll to line 10 and then pr ess enter. Each time the user presses the up or down arrow, the command is automatically displayed at the command line.

Unix Utilities

An infinite number of utilities run on Unix. It goes beyond the scope of this text to cover those utilities, but some popular comm ands that you could look in to with the **man** command are listed here in summary:

COMMAND	DESCRIPTION
`finger USER`	See information about a user on the system.
`finger USER@host`	See information about a user on another server.
`slogin -l USER HOST`	Login to another server connected on the network.
ssh `HOST`	Secure shell login program.
`ftp`	File Transfer Protocol (to copy files between servers)
`rcp`	Remote Copy (to copy a file between servers)
`telnet`	Log in to a remote machine.
`javac FILENAME`	To compile a Java program
Java `CLASSNAME`	To execute a Java program
`vi FILENAME`	To text edit a file
`sort FILENAME`	Sort the contents of a text file.

COMMAND	DESCRIPTION
`grep DATA FILENAME`	Search a text file for some data.
`cc -OPTIONS FILENAMES`	Compile a C program.
`gcc -OPTIONS FILENAMES`	GNU open source C and C++ compiler

In the above list, two commands are important, and we will look at them here: `grep` and the `rlogin/slogin/ssh` set of commands.

Network Access

An important theme in Unix is the operating system's ability to interact freely with networks. This ability is expressed through many commands like `rlogin`, `ftp`, `rcp`, `telnet`, and `ssh`, to name a few.

The program `rlogin` is known as Remote Login. The program `slogin` is known as Secure Remote Login. The program `ssh` is known as Secure Shell. These programs allow users to log in from their shell into another server. The server can be on a local area network, a wide area network, or the Internet. It does not matter where the server is. Remote Login connects you to a server on an unsecured text-based communication line. This means that all the data sent to and from you and the server is transmitted in text and is not encrypted. Secure Remote Login encrypts the text. SSH is similar but possesses a stronger encryption technique and data-passing scheme. Here is an example:

```
$ ls
file1 file2 file3
$ rlogin -l jjsmith mimi.cs.mcgill.ca
login: jjsmith
password: *******
Connected to: mimi.cs.mcgill.ca
% ls
file4 file5 file6
% slogin -l joev skinner.cor.com
login: joev
password: *******
Connected to: Skinner International
[joev] $ logout
% logout
$ logout
```

In the above example, the user is logged into a server that prompts with a dollar sign. The user lists the files in the current directory. The using `rlogin -l` then user asks to log in remotely

to the server named mimi.cs.mcgill.ca with the user name jjsmith. If the server exists and is online, the user is prompted only for the password. It will assume the username is, in this example, jjsmith, as specified. If the password is correct, the new server displays a connect message. Notice the new prompt; it is a percentage sign. The new prompt reflects the defaults of the new server. Now the user has access to both servers: the one she started with and the one to which she has now connected. Here the user lists all the files on this new machine. Now she does a secure remote login with a third server called skinner.cor.com with the username joev. Again, if the server is valid and is online, the user is prompted only for the password. The username joev is assumed since it was specified with the -l switch. Upon successful login, the new prompt is displayed. In this case, the username is in square brackets with a dollar sign. Now the user is connected to three servers at the same time. In this example, the user enters the command `logout` in skinner and the prompt changes to the mimi prompt, indicating that the connection to skinner has been closed. The user is returned to her next most recent login session, which in this case, is the mimi server. The command `logout` is entered again, and the prompt changes to the original prompt from the original server. The mimi connection has been closed.

In a non-windowed interface, each login session is stacked on top of each other, and the user has only access to the most recent session. In a windowed environment, the user can open a window for each login session and can have access to all of the sessions at the same time.

The `grep` Command

The `grep` command is a powerful Unix command that has been copied by many other programming languages like Perl and Python, to name a few. In this section, I will only introduce the basics of this command. You can `man grep` to find out more details, or you can read the Perl and Python chapters for more information.

The `grep` command is a powerful text file searching tool. Its power comes from its method of expressing a query using a syntax known as **regular expressions**. Let's start with some syntax, followed by examples, and then we will look at regular expressions in some more detail.

Syntax:

```
[PROMPT] grep {SWITCHES} SEARCH_KEY FILE{S}
```

Where:

`grep`	is the name of the program, displaying lines that match
`SWITCHES`	are optional: -i ignore case -c report only on a count for successful matches -v report on lines that do not match

	-n display the line number with the line
	-l list the file names that have the search key
SEARCH_KEY	is what you want to find (in regular expression format)
FILE	is one or many file names to be search (separated by spaces)

Consider the following text file:

```
Alex
Marc
Micheal
Ting
Juan
Jeremy
Jessica
Yannick
Nicolas
Jean-Sebastien
Nadeem
```

Consider the following `grep` commands and their results:

```
$ grep 'Je' demo.txt
Jeremy
Jessica
Jean-Sebastien
```

Lines that begin with Je

```
$ grep -n 'Je' demo.txt
6:Jeremy
7:Jessica
10:Jean-Sebastien
```

Lines beginning with Je together with line numbers

```
$ grep -c 'Je' demo.txt
3
```

Three lines have Je at the beginning

```
$ grep -i '^[aeiouy]' demo.txt
Alex
Yannick
```

Lines beginning (^) with an a,e,i,o,u, or y ([ ])

```
$ grep -i '[aeiouy]$' demo.txt
Jeremy
Jessica
```

Lines ending ($) with an a,e,i,o,u, or y ([ ])

```
$ grep -i '[aeiouy]{2,}'
demo.txt
```

```
Micheal                    Lines with 2 or more a,e,i,o,u,y anywhere in a row
Juan
Yannick
Jean-Sebastien
Nadeem

$ grep -i '^.e' demo.txt
Jeremy                     Lines beginning (^) with any letter (.) and an e
Jessica
Jean-Sebastien

$ grep -i '^.e|a.$' demo.txt
Micheal                    Lines beginning (^) with any letter (.) and an e or (|) ending
                           with the letter 'a' followed by a letter
Juan
Jeremy
Jessica
Nicolas
Jean-Sebastien
```

This is only a small example of the power of this command. Some of the regular expressions are defined here. Note that these would appear in the SEARCH_KEY.

`\f`	Line contains a form feed
`\n`	Line contains a carriage return
`\t`	Line contains a tab
`\w`	Line contains one of these: a to z, A to Z, 0 to 9
`\W`	Line contains one of these: the opposite of \w
`\s`	Line contains a white space: space, \n, \r, \t
`\S`	Line contains one of these: the opposite of \s
`\d`	Line contains a digit
`\D`	Line does not contain a digit
`*`	Zero of many occurrences of ...
`?`	Zero of one occurrence of ...
`+`	One or many occurrences of ...
`.`	Exactly one character of anything at this spot
`^`	Must match at the beginning of the line
`$`	Must match with the end of the line
`|`	Or
`{a,b}`	String must occur minimum a time to the maximum of b times

( )	Group these letters together as a single word
[ ]	These letters are treated as OR; any of them can match
-	This designates a range: a-z, A-Z, 0-9 are defined.

Unix Archiving and ZIP Files

Unix has two ways of archiving files. The first way is to merge a group of files into a single file. This is known as archiving. This single file can then be compressed in size. This is called zipping. An archive, being one file, is easier to manipulate (move, store, copy, backup, etc).

The three most popular archive tools used on Unix systems are **tar, gzip**, and **gunzip**. Tar allows you to combine several files into a single file. gzip allows you to compress a single file. gunzip unzips a file. To compress a collection of files, you need to use both tar and gzip. Other archive tools are also available. Most of these will both combine and compress files. For example: Zip, bzip2, 7z, rar, and arj. But, tar, gzip and gunzip come standard with the operating system.

TAR

The archiving tool, tar, allows the creation and extraction of archive files. An Archive file is understood to be a single file that contains many unzipped files. You can imagine this as a single word processor document where you have cut and pasted a series of correspondences and inserted them one after the other into this single document.

The tar command's switches:

- The -c switch indicates that a new tar file will be created.
- The -r switch indicates that an existing tar file will be updated in some manner.
- The -x switch indicates that the files from an existing tar file will be extracted.
- The -f switch is important because it specifies the archive filename. If not supplied, a default name is used.
- The -v switch activates verbose mode, which means it will output a lot of information as it archives.
- The -z switch allows you to compress the archive file (it uses gzip and gunzip).

A file ending with the .tar extension is a tar archive file. A file ending with the .tgz extension is a compressed (gzipped) tar archive file.

Here are a few example of the tar command:

```
$ tar -cvf log.tar *.log
$ tar -zcvf log.tgz *.log
$ tar -xvf log.tar /tmp/log
$ tar -zxvf log.tgz /tmp/log
```

The first two commands create an archive file named log.tar by combining all the log files in the current working directory. The first command does it uncompressed, and the second one does it compressed. The two following commands show how to extract those two archives.

EXAMPLE UNIX SESSION

In this section, a sample Unix session is obser ved with comments. Many interesting commands are explored.

Below is a simple example session a t the comm and prompt. The prompt has already been changed from $ to the name of the server [mimi], and it displays the path of the current directory as part of the command-line prompt. This was created using the environment commands:

Set prompt = [%m] [%~]
Where %m is the machine name and %~ is the current directory.

Look at the example session below:

```
[1] Welcome to Ubuntu
[2] $ set prompt=[%m][%~]
[3] [mimi] [~/jjsmith] clear
[4] [mimi] [~/jjsmith] ls -l -a
    total 8
    drwxr-xr-x   6  jjsmith 12521        512 Jul 17  2003 .
    drwx--x--x  15  jjsmith 12521       2560 Feb 10 12:24 ..
    drwxr-xr-x   2  jjsmith 12521        512 Jul 24  2003 CSC204
    drwxr-xr-x   2  jjsmith 12521        512 Jul 17  2003 CSC206
    drwxr-xr-x   2  jjsmith 12521        512 May 31  2002 CSC317
    drwxr-xr-x   2  jjsmith 12521        512 Jul 17  2003 Csc302
    -rwx------   1  jjsmith -            50 Jan 12  2011 .cshrc
    -rwx------   1  jjsmith -           201 Sep 15  2010 friends.doc
[5] [mimi] [~/jjsmith] cd CSC206
[6] [mimi] [~/jjsmith/CSC206] ls
    206Final2003.doc csc206c2003.xls
[7] [mimi] [~/jjsmith/CSC206] cd ..
[8] [mimi] [~/jjsmith] cd CSC317
[9] [mimi] [~/jjsmith/CSC317] ls
    CSC317 Project.doc
```

```
[10] [mimi] [~/jjsmith/CSC317] cd ../Csc302
[11] [mimi] [~/jjsmith/Csc302] ls
     BU302Mid2000.doc Bishop 302 Outline.doc bu302ass3.txt
[12] [mimi] [~/jjsmith/Csc302] cat bu302ass3.txt
     User data for Bishop's and McGill Universities.
[13] [mimi] [~/jjsmith/Csc302] _
```

The following has occurred:

1. The OS displays welcome, system, and version information to the user who just logged in. The user has arrived in her home directory called jjsmith.
2. The user uses the `set` command to change the prompt to display the server name, %m, and current directory, %~.
3. The user the clears the screen.
4. The user then asks for a directory listing of all objects in long format.
5. The current directory is changed to CSC206.
6. The changed directory name is displayed by the prompt. The user asks to see the files.
7. The current directory is changed to the parent directory (the one above).
8. The current directory is changed to CSC317.
9. A directory listing is requested again.
10. The current directory is changed to ../Csc302 (up to parent, down to Csc302).
11. User asks to see the files in this folder.
12. Using the command cat, the user displays the contents of file bu302ass3.txt.
13. Prompt is waiting for user's next command.

Another session example:

```
[1] $ date
    Tue Feb 10 14:59:53 EST 2011
[2] $ (date; cal 2004; who) > temp
[3] $ who | sort > temp2
[4] $ MyCalcOfPI3000Digits &
[5] $ ps
    PID    TTY    TIME    COMMAND
    3140   P0     0:01    csh
    3271   P0     0:05    MyCalcOfPI3000Digits
    3290   P0     0:00    ps
[6] $ kill 3271
    Terminated
[7] $ grep little poems.txt
```

```
[8] $ history
1     10:00 date
2     10:01 (date; cal 2004; who) > temp
3     10:03 who | sort > temp2
4     10:05 MyCalcOfPI3000Digits &
5     11:00 ps
6     11:01 kill 3271
7     11:02 grep little poems.txt
8     11:05 history
[9] $ !4
```

The following has occurred:

1. The user asks to see the current date.
2. The user asks to run three commands at the same time: date, cal 2004, and who. Date will display the date as in [1]. The command cal 2004 will show the 12 months in 2004. The command will display the currently logged in user. All this information will be placed into the file temp.
3. The user asks for all the user names who are currently logged in to be sorted as saved into a text file called temp2.
4. The user then runs her own program writing the digits of PI to some file. This program will never end. Notice that the command uses the ampersand. This is important so that the prompt is displayed, given that the program will never end.
5. Using the processor status command, the user asks for the currently running programs in its account. We see csh, the PI displaying program, and the ps command just issued. Each program has its own process ID number called PID. The computer it is running on, in this case, P0 and how long it has been running for.
6. The user wants to stop the PI program.
7. The user wants to find the word "little" in the file "poem," but it is not there.
8. The user wants to see the commands she has done.
9. The user re-issues command 4, which runs the PI program again.

THE UNIX SCRIPTING ENVIRONMENT

The Unix scripting language is a simple pr ogramming language based on the Unix comm and-line commands and command-line environment. If you are familiar with the Unix command-line commands and you have some basic knowledge of programming, then you can write scripts.

Unix scripting combines simple text files and the Unix comm and-line commands. The idea is simple. Write Unix commands in a text file, imagining that you are at the command-line

prompt. Then save that text file and chmod it to executable. When you type the file name at the command-line prompt, Unix will open the file and then issue each command, one at a time, to the command line automatically.

Two types of script files exist: system scripts and user scripts. System scripts are script files that the Unix operating system looks to launch. These scripts have specific names. There are Start-up scripts that are executed when the operating system is booted. There are login scripts like .cshrc that are automatically launched when a user logs in, and then there are logout scripts. User scripts are never automatically launched by the operating system and so can be given any name. The user must specifically request to launch these scripts by entering its name at the command-line prompt.

Below is an example of a Unix start-up script:

```
#!/bin/sh
# Executed at login
echo "Welcome Home!"
set prompt = "$home>"
alias dir ls -l -a
set history = 100
who
pine
```

The above script displays "Welcome Home!" and sets the prompt to the user's home directory followed by a greater-than sign. The **alias** command substitutes the word DIR for the command **ls –l –a**. The environment shell is set to remember the user's last 100 commands. Then the script displays all the users currently logged into the computer. Lastly, the script executes the **pine** program so the user can read email. The script uses the SH-BANG (sharp followed by the exclamation point, #!) to specify which shell to use to execute the script.

The Unix login start-up file language is directly related to the shell being used. In Microsoft operating system products, there are only two shells: DOS and Windows. In Unix, there are many shells: **Bourne, Korn, C-Shell** and **X-Windows**, to name a few. Each shell expects the start-up file name to be a specific name (as in DOS with autoexec.bat). In Unix, a script is a text file and does not need to have a special file extension to indicate that it is a script file (unlike DOS, which requires the .BAT extension). In Unix, a text file can be designated as executable (`chmod a+x filename`). This may sound strange, but if a text file is marked as executable, then the shell will try to interpret it. Here is a list of standard Unix files and their purposes:

.cshrc	The login file for the C-Shell (i.e. /bin/csh)
.forward	Your email forwarding address
.kshrc	The login file for the Korn shell (i.e. /bin/ksh)
.login	The login file when you login (for csh)

.logout	Executed when you logout (for csh)
.plan	Information about yourself display when someone **fingers** you
.profile	The start-up file for the Bourne shell (i.e. /bin/sh)

As in DOS, the Unix shell's environment memory uses some special variable names. Here is a list of the more commonly used ones:

HOME	Stores the pathname to your home directory
PATH	The search path when a program is not in the default directory
SHELL	The pathname to the shell
TERM	The termcap code for your terminal
USER	Your login name
PWD	The current working directory

To assign values to the environment variables, three commands can be used depending on the shell and the script:

System Script Syntax:		User Script Syntax:
setenv VARIABLE VALUE	set	VARIABLE=VALUE

If the shell is sh or ksh, then there is a simpler syntax:

```
VARIABLE=VALUE
```

Examples:

```
TERM = vt100
setenv TERM vt100
set TERM=vt100
```

The Unix shell performs many functions:

- It is a script interpreter.
- It provides a user interface (the command-line prompt).
- It has a global shared-environment memory that all processes and the user can access.

There are many shells. Three popular shells are:

- The Bourne Shell (/bin/sh)—This is the standard shell for Unix and the oldest one.
- The Korn Shell (/bin/ksh)
- The C Shell (/bin/csh)—This is popular with programmers since its interpreter operates something like the C and Perl programming languages.

The shell, in general, performs the following activities in the order presented:

1. Execute start-up script when shell is first activated.
2. Prompt the user.

3. Process the command internally, if possible.
4. If internal processing is not possible, then assume it is a program and locate it using the shell variable PATH.
5. Goto step 2 until the user enters the **exit** or **logout** command and goto step 6.
6. Terminate the shell program by executing the logout script.

Scripts are programs that the shell can in terpret and execute. Scripts are text files that have been **chmod**'ed to be executable. Four types of instructions can populate a script:

- Operating system command-line prompt commands
- Environment variable memory commands
- Third-party software execution
- Script programming language commands

Each script is a text file containing instructions written in a top-down, left-right manner. Each instruction exists on its own line in the scr ipt, except when instr uctions are separated by the semicolon (the multi-execution operator).

Below is a series of user script program examples. Each one begins with code, sample output, and then a line-by-line description.

EXAMPLE 1

```
#!/bin/sh                                                    (1)
# This example shows how variables are used in scripts.
# Saved as filename eg1.
echo 'how old are you?'                                      (2)
read age                                                     (3)
echo "You are $age years old"                                (4)
echo "You entered $# arguments at the command prompt"        (5)
echo "The arguments where: $*"
echo "Your first arguments were $0 $1"                       (6)
```

Screen Output

```
$ eg1 a b c d                                                (7)
How old are you?
10
You are 10 years old
You entered 5 arguments at the command prompt
The arguments where: eg1 a b c d
Your first arguments were eg1 a
```

There are three types of variables accessible from a script:

- **Environment variables**: The **set** command is used to assign a value or cr eate a variable with a value in the en vironment memory. For example: set var = value . You can access the stored environment variable value by simply using its variable name in the script.
- **Script defined variables**: These are variables created within the scr ipt. Unlike other programs where you need to declar e your var iables, in scr ipts, you simply star t using them where you need them and the in terpreter creates them for you a t that moment. Script variables are used at line **(3)** and then again at **(4)**. Variables are used in the script by simply assigning a value to its n ame using the equal symbol (=) or b y some input command like **read**, as in line **(3)**. When the value is required, the variable is preceded by the dollar symbol ($), as in line **(4)**.
- **Positional parameters**: These are special built -in variables that store the arguments entered at the comm and-line prompt when the scr ipt was initia ted. When, a t the command-line prompt, the scr ipt is execut ed, extra arguments can be supplied beside the script's filename. These arguments will be passed to the scr ipt as values for processing. Your script can ignore them without any erroneous effects. In our example, line **(6)** uses positional parameters. Notice that the script's filename is part of the positional variables.

In the programming example abo ve, line **(7)** shows how the scr ipt **eg1** is called w ith four arguments: the characters a, b, c, and d. The script loads a shell to interpret it at line **(1)**. The sh-bang symbol (#!) is used to r equest the shell. The sharp symbol (#) when used alone is treated as the comment marker for scripts. All text, on that line, after the comment marker, is ignored by the interpreter. In line **(2)**, the **echo** command is used to pr int on the user 's screen. Notice that line **(2)** and **(4)** use different **quoting symbols**. The single quote tells the script to display the infor mation between the quotes exactly as it appears . The double quote tells the scr ipt to preprocess the information between the quotes. All escape ch aracters are preprocessed first. This generates a new str ing. This new string is then used b y the command. Valid escape characters are the dollar sign ($) and the backslash symbol (\). The string is modified by replacing any word beginning with the dollar symbol ($) with the value stored in the cor responding variable having the same n ame. The backslash defines formatting rules like \n is car riage return and line feed. W e have seen this befor e in C; the same rules apply. Next, the user is prompted for his/her age. The interpreter at line **(3)** spontaneously builds the variable **age**. The next few lines experiment with outputting different types of information. Line **(4)** outputs the value in the variable **age** (note the dollar symbol within the double quotes). Line **(5)** encompasses two sour ce lines. In one line, the symbol $# is used. The other line uses $*. The $# symbol outputs the number or arguments passed to the script. The $* symbol outputs the actual values of the passed arguments. Line **(6)** shows the script

accessing the individual arguments. Each argument is accessible using its position from the command line. Therefore, $1 refers to the first argument, $2 the second argumen t, $3 the third, and so on. The positional parameter $0 refers to the actual name of the script as it was entered at the command line.

EXAMPLE 2

```
#!/bin/sh
# This example demonstrates the back-quote and arithmetic
# The filename for this file is eg2
set `date`                                                    (1)
echo "Day: $1"                                                (2)
echo "Date: $3 $2 $6"                                         (3)
days = `exptr $6 * 365 + $3`                                  (4)
echo "Total days = $days"                                     (5)
```

Screen Output

```
$ date
Wed Feb 11 16:45:01 EST 2004
$ eg2
Day: Wed
Date: 11 Feb 2004
Total days = 731471
```

This script uses two special commands:

- The first is the back-quote (`) symbol. Script files like **echo** and **set** can execute operating system commands only when they ex ist on their o wn line in the scr ipt. But, when you want to execute an operating system command within another command or script statement, you need to use the back-quotes. The back-quotes will execute the operating system command first and then apply the r esult to the remainder of the sta tement. We see that done on lines **(1)** and **(4)** of the script.
- The second new command is the **expr** program. The **expr** program is a simple application that takes arguments from the comm and-line as text, converts them into mathematical values, computes, and then returns the result. Check **man** for a full definition of how it functions. It can perform the following operations: + (addition), - (subtr action), * (multiplication), / (division), and % (modulo).

Line **(1)** executes the **date** command and asser ts the result into the comm and-line memory space of the script. This is interesting because we are doing this *after* the program has already

started to execute. We are not at the command-line now. The **set** command without a variable name will assert the value at the command-line instead of the shell 's global memory space. The program then can have access to these values using the positional parameter variables [as in lines **(2), (3)** and **(4)**]. Line **(2)** extracts the name of the day and displays it. Line **(3)** extracts the date values and orders them day first. Line **(4)** uses the year and day values in a calculation to determine the total number of days. Since scripts are text-processing environments, mathematical calculations are not possible. Therefore, a special program exists as standard in the Unix library that parses a line of text and attempts to find a mathematical solution for it. If one exists, the result is returned; otherwise, an error message is displayed. Line **(5)** displays the result.

EXAMPLE 3

```
#!/bin/sh
# Script Control Structures
# Saved as file eg3
echo 'Enter number of loops:'
read count
if test $count=0                                                    (1)
then
  echo "The count can not be zero. Enter a number again:"
  read count
fi                                                                  (2)
while $count>0
do
  echo "Loop number $count"                                         (3)
  count=`expr $count - 1`
done
```

Screen Output

```
$ eg3
$ Enter number of loops:
3
Loop number 3
Loop number 2
Loop number 1
$
```

The above example shows two script-control structures. Bash actually has five control structures in total:

- The syntax for the if-statement is:

```
if CONDITION
then
   EXPRESSIONS
elseif CONDITION2
   EXPRESSIONS
else
   EXPRESSIONS
fi
```

if..fi: is how you begin and end the if-statement.
then: is mandatory and is executed when the CONDITION is true.
elseif: is optional and permits nested if-statements.
else: is optional and is executed only when all CONDITIONs are false

EXPRESSIONS: are multiple valid script commands or statements.

CONDITION and CONDITION2: are conditions formulated with the **test** program (described later). Conditions can be compounded using the or-expression (||) and/or the and-expression (&&).

- The case statement does pattern matching with an input word and the word located at each case. The syntax is as follows:

```
case WORD in
PATTERN1 ) EXPRESSION; EXPRESSION; … EXPRESSION;;
PATTERN2 ) EXPRESSION; EXPRESSION; … EXPRESSION;;
…
esac
```

WORD can be a string or a variable containing a string. WORD is compared with each of the PATTERNi strings. Each PATTERNi can be either a string or a variable containing a string. The open round bracket separates the PATTERNi from the set of EXPRESSIONS that will be executed when WORD matches PATTERNi. Each EXPRESSION is separated by a semi-colon, and the entire set of EXPRESSIONs are terminated by a double semicolon.

- The while-loop operates much like its counterpart in C. Its syntax is:

```
while CONDITION
do
    EXPRESSIONs
done
```

CONDITION operates as it does in C. You can use the >, <, >=, <=, !=, &&, || operators. The comparison operator is =, not = = as in C. The condition does not need the **test** program;

but, it is often used with the **expr** program for integer calculations. EXPRESSIONs are any legal script command or statement, each on its own line.

- The for-list-loop syntax is as follows:

```
for VARIABLE in LIST
do
    EXPRESSIONs
done
```

VARIABLE is an empty variable. LIST is a string containing many words separated by spaces, or it can be the $* script operator indicating all the arguments from the command-line. VARIABLE is assigned each word from LIST, one word for each iteration. EXPRESSIONs are any legal script command or statement each, on its own line. Normally, EXPRESSIONs uses the VARIABLE in some way.

- The until-loop works much like the while-loop but tests the opposite condition. In the case of the while-loop, the loop iterates when CONDITION is true. In the until-loop, the loop iterates when the CONDITION is false. Its syntax is as follows:

```
until CONDITION
do
    EXPRESSIONs
done
```

As before, CONDITION is as defined in the while-loop, and EXPRESSIONS are any legal script command or statement each on their own line.

We now need to describe **expr** and **test** in some detail:

- The **expr** program parses a string to perform integer, string, or logical computations. Use the **man** command to get a complete description of it. Its basic syntax is as follows:

```
VARIABLE =‘expr STRING‘
```

VARIABLE is any script variable and is optional. If it is not used, the script will try to execute the result of the computation. This can be useful when other program names match the result of the computation.

- STRING is any of the following statement types:
    - Integer expression of the form VALUE OPERATOR VALUE where VALUE is either a constant or a variable and OPERATOR is +, -, *, /, or %. For example: expr 5 + 2.

  - Boolean expression of the form VALUE OPERATOR VALUE where VALUE is either a constant, word, or a variable, and OPERATOR is <, >, <=, >=, !=, =, &&, or ||. STRING in this case can be a compounded expression using the && or || operators. For example: expr (5 + 2) > 10.
  - String expression: I leave this to be explored by the student using the **man** command.

- The **test** operator syntax is as follows:

```
test EXPRESSION
```

  Where EXPRESSION can be either a CONDITION or a SWITCH-CONDITION. A CONDITION is a normal conditional expression as we have seen in the while-loop and the until-loop.

  A SWITCH-CONDITION can take on many forms. It is left to the student to explore all the forms using the man command, but some useful ones are listed here:

Switch Condition	Test is true when
-d filename	filename exists in the current directory
-f filename	filename is a legal file
-r filename	filename is a readable file
-w filename	filename is a writable file
-x filename	filename is an executable file

The example program is very simple. Line **(2)** and **(3)** demonstrate an important property of scripts, that an expression is a series of characters *not* interpreted by the separation of spaces. Therefore, if we take line **(3)**:

```
count=`expr $count - 1`                    (A)
```

or

```
count = `expr $count - 1`                  (B)
```

Example (A) is the correct way to write the expression, without any spaces between count=expr. In example (B), spaces were inserted on both sides of the equal sign. This will be interpreted by the script as a request to execute the result of the expression, instead of assigning it to count. Line **(1)** shows how to use the **test** operator.

EXAMPLE 4

```
#!/bin/sh
# A final mixed example, saved as eg4
filename=$1
if test ! -f $filename                                          (1)
then
     echo "$filename does not exist"
else
     chmod a+rwx $1
     echo "$1 will now be executed…"
     $1                                                         (2)
fi
finger $2
set `date`                                                      (3)
case $1 in                                                      (4)
   Mon) echo "Activities for Monday…";;
   Tue) echo "Activities for Tuesday…";;
   Wed) echo "Activities for Wednesday…";;
   Thu) echo "Activities for Thursday…";;
   Fri) echo "Activities for Friday…";;
   Sat) echo "Activities for Saturday…";;
   Sun) echo "Activities for Sunday…";;
Esac
```

Screen Output

```
$ eg4 test.exe jack
test.exe will not be executed…
Login name: jack         In real life: Jim SMITH
Directory: /u1/prof/jack     Shell: /usr/local/bin/tcsh
On since Feb 12 09:38:20 on pts/21 from toronto-hse-ppp3718083.sympatico.
ca
No unread mail
No Plan.
Activities for Thursday…
$
```

Line **(1)** shows the use of the oper ator **test** in an if -statement. In this case, it tests if the file provided by the user from the command-line actually exists. If it does not, an error message is displayed; otherwise, the file is made executable, and then in line **(2)**, it is executed. Once the execution is complete, the script continues. It fingers the second command-line argument and then at line **(3)** resets the command-line argument values with the current system date. This is used in line **(4)** to select the day's activities.

TEST YOURSELF!

Try writing the following programs:

1. Write a script that asks for your age and then displays the number of days you have lived.
2. Modify the age calculation from question 1 to accept your date of birth from the command-line. It will use the **date** command to find the current date and display the exact number of days between your birth and today.
3. Write a script that would execute some useful daily operations you would like to have performed for you. Add this to the shell start-up script .cshrc.
4. Write a script that will **finger** everyone who is currently logged into the system.
5. Write a script that will **grep** a text file for a specific word and then display a message indicating at what line in the file the word was located.

CHAPTER THREE

Understanding C

Dennis Ritchie originally developed C in 1972 at Bell Laboratories from a language called B, which was itself a simple but robust version of a programming language called CPL. He used it to rebuild Unix. Unix was originally programmed in assembly, but Ritchie used C to rewrite almost its entire code base, except for the bootstrap portion that remained in assembler. This new version of the operating system was running on the DEC PDP-11. Unix, the C compiler, and all the tools Ritchie used were essentially programmed in C, which was found to be powerful and very easy to use. It produced compact and optimized code. Its ability to have direct access to the computer's operating system and hardware while maintaining a simple high-level syntax with powerful operator usage has caused C to endure. C has become so popular that it has inspired three offspring: C++, Java, and C# (pronounced C sharp). C++ and C# are object-oriented extensions to C, while Java has brought C++ to the Internet. C still maintains the simplest syntax (in this family).

C is three languages in one: the actual C language, the pre-processor language, and C's huge library. You can always identify pre-processor directives since they all begin with the sharp (i.e., #) symbol. The pre-processor statements are used to adjust how the compiler will process the C program. This can influence which C statements get compiled and which are ignored. Actual source code text can be replaced with other pre-processor text. This can also influence which source code file is included or is ignored by the C compiler. There is probably no one who knows all the commands in all the libraries that are compatible with C. In this text, we will just introduce you to some of the more popular and common libraries. We will learn all of the standard C language.

COMPILING UNDER UNIX

One of the first things to know is how to create a C program. We will show you the simplest way here. Start by writing your program with any text editor. In MS Windows, this could be

Notepad. In Unix, this could be vi. Simply open your text editor, write your C program, and then save it with any name you like. The file extension must be .c.

The next step is to compile your .c text file. We are using the GNU Tool Set in this text, which includes the C and C++ compiler called gcc. The GNU Tool Set is a free Open Source product that you can download from the Internet. Assuming that you have GNU installed on your computer, you can use any of its tools from any command-line prompt. Assuming we are in the directory where you saved your .c file, at the prompt, you can type the gcc command to compile your program. The gcc command has the following syntax:

```
$ gcc -o EXECUTABLENAME FILENAMES
```

In one of its more simple forms, gcc takes only two arguments. The first argument is optional and is designated by the `-o EXECUTABLENAME` portion of the above syntax. It specifies the name of the compiled executable file produced by gcc. This file is only created when no syntax errors are found. If no executable name is provided, then a default executable name is used. In Unix, the default executable file name is called `a.out`. The second argument, `FILENAMES`, is mandatory and consists of a list of source file names and/or object file names separated by a space. Source file names are identified by either a `.c` or `.h` file extension. A `.o` file extension in Unix, or a `.obj` file extension in MS Windows identifies an object file name. Each source file is compiled and then linked with any object files in this command-line argument. The compiler, if no errors were found, then links the C library and operating system library specific routines into your program. This merged set of routines is saved to disk with the provided `EXECUTABLENAME` or the default `a.out` name when no executable name is provided.

For example:

```
$ gcc -o myprog f1.c f2.c menu.o
```

The above is an example of the gcc compiler command invoked from the Unix command-line (the $ prompt). The executable file will be called myprog if no errors are found while compiling. The two source files, f1.c and f2.c, will be compiled and then linked with the menu.o object file, plus any needed C library and operating system specific functions. The gcc compiler, on its own, determines what operating system functions to link. The C program must specify which C library functions to link. If everything goes well, then the program can be invoked by typing `myprog` at the prompt.

EXAMPLE 1: SIMPLE PROGRAM

```
#include <stdio.h> /* This is a comment in C */
#include <stdlib.h>                                    (1)
int main( int argc, char*argv[] )                      (2)
{                                                      (3)
     printf("Hello World\n");                          (4)
}
```

Our first program demonstrates how to construct a simple but complete C program. The above program will print the words *Hello World* on the computer screen and then move the cursor to the next line. Assuming this program was saved in a text file called sample1.c, it can be compiled using: `$ gcc -o ex1 sample1.c`. The program can then be launched by typing `ex1` at the command-line prompt.

1. Pre-processor directives are identifiable by the preceding hash symbol (#). In this example, the pre-processor directives are specifically requesting that the Standard IO (i.e., stdio) and Standard Library (i.e., stdlib) header files (i.e., .h) be included with the program. In other words, the program wants to use the stdio.h and stdlib.h files and their corresponding C libraries. C has many such libraries. #include merges the .h text files with your source file at the exact position where the #include is placed. Stdio.h contains definitions for many of the basic input and output commands. Stdlib.h contains definitions for many commonly used functions and literals. They will be introduced to you slowly in this text.
2. Immediately following the pre-processor directive is the main program. **It is the function that is executed first when the program begins**. It is identifiable by the reserved word `main`. One copy of the main function must be present in all programs. It can be placed anywhere in the source file.
3. C uses the open and close curly brackets (i.e., { and }) to denote the **grouping of elements into a single unit**. We will see this used often in C. In this case, it indicates that any programming statements within these brackets are part of the **main** function. The open bracket indicates the beginning of the function. The close bracket indicates the end of the function.
4. The only statement in the main function is the **printf** statement. We will talk about it in greater detail later. For now, we will just mention that it is the **basic output command** that writes ASCII text to the screen, in this case, the words *Hello World*. As can be seen from the example, the text to be output is enclosed in double quotes. The text can also contain special codes, like \n. This code denotes that **printf** will move the cursor to the next line after it prints the words *Hello World*.

Resulting Output for Example 1

```
Hello World
_
```

Assume that the above box is the computer screen. Assume further that the screen was blank at the beginning and that the cursor was at the top left corner of the screen. Our example program would display as indicated in the box above. The H of hello would appear at the top left-hand corner of the screen. The cursor would be positioned below the H on the next line.

Syntax Details for Example 1

The Main Function

```
RTYPE main (int ARGC, char *ARGV[])
{
   STATEMENT(S);
   return RESULT;
}
```

Where:

- You can identify the main function by its use of the reserved word **main**. Every program must have only one of these. All programs begin execution at the main function.
- It has an optional return type (identified as RTYPE). RTYPE can be assigned either the reserved word **int** or **void**. In some compiling environments, you cannot use **void**. The RTYPE **int** indicates that the main function will return an integer number to the calling environment. The **void** indicates that nothing will be returned to the calling environment.
- To the right of the reserved word **main** are two round brackets. Within these brackets are the parameters of the **main** function. This is information sent to the main function from the calling environment. ARGC is a positive integer number denoting the number of tokens stored within the array ARGV. Each token is stored within one cell of the array. A token is defined to be a single word separated by a preceding and trailing white space (i.e., Blank, tab, carriage return, etc.). For example:

  ```
  $ MyCopy a:fluf.txt db
  ```

  Will give: ARGC = 3, (i.e., Copy, a:fluf.txt, db). Each token is stored in its own cell within ARGV. Since C's array cell index starts with zero, then "copy" would be in cell zero, "a:fluf.txf" in cell one, and "db" in cell two:

  ```
  argv [ ] =  0            1            2
             "copy"     "a:flif.txt"   "db"
  ```

  The main program's arguments are optional and can be left out and replaced with the reserved word **void**. If the arguments are used, then they **both must appear** as described. The round brackets are mandatory.

About Comments in C

Comments in C are denoted by the /* and */ combinations. The compiler will ignore text between these two markers. Comments can be placed anywhere in your program. Their

purpose is to help explain things using a r egular human language. Take advantage of it. You will not always remember why you did something. It is also useful to other programmers looking at your code for the first time.

These markers can be placed on the same line as code, between code, and over multiple lines, as in this example:

```
/* this is
   a multi-lined
   comment.
*/
```

Since gcc is also a C++ compiler, then C++'s comment syntax will also work in your program. C++ comments use the double-slash, //. For example:

```
//
// This is a
// multi-lined
// comment
//
```

Notice how you have to put the double-slash on every line, unlike C's style.

EXAMPLE 2: VARIABLES AND EXPRESSIONS

```
int main (void)
{
   const int x = 5;                                          (1)
   char c;
   float y;
   double z;

   c = 'a';                                                  (2)
   y = z = 10.2;
   y = y + 1; y = x + z;

   printf("x=%d c=%c y=%f z=%f\n",x,c,y,z);                  (3)
}
```

Our next pr ogram introduces variables and simple expr ession processing. The above program initializes thr ee variables and one constan t. It then carries out multiple mathematical calculations and prints a result on the screen. The output uses the printf command in a more comprehensive manner, which we will describe in detail in example three, below.

1. Identifiers, literals, types, variables, and constants

   Identifiers: An identifier is a user-supplied name used to identify a portion of a program that does not have a reserved word identification label. In the example above, this would be the variable names: c, y, and z. and the constant named x.

   Literals: A literal is something that possesses an inherent value. Numbers, for example, are literals and possess a value that has an inherent meaning. The value 5.2 is a floating-point number and is understood to be so by the compiler. Literals must be used by the programmer in ways that agree with the meaning of the literal's value. **Like reserved words, they possess a meaning that cannot be changed by the programmer**.

   Types: A type is a reserved word that tells the computer about the "kind" of information being stored in memory. A type defines the amount of space reserved in memory for that information and the valid operators and functions that can be invoked to manipulate and modify it. The reserved words **char**, **int**, **float**, and **double** in the code above are examples of types. For example, the char type can only store a single character.

   Variables: A variable is a space in memory (RAM) that has been named, using an identifier, and designated to store a particular type of information, like char or int. In the example above, **c**, **y**, and **z** are variables. The syntax is `TYPE IDENTIFIER_LIST;` notice that the identifier list can enumerate many variable names but each is separated by a comma, for example: `int x, y, z;` the identifier list also allows the variables to be initialized, for example: `int x, y=2, z;` the initialization can happen in any order or combination.

   Constants: A constant is a special form of variable whose value cannot be modified once it has been set. The value can be accessed, but it cannot be changed. In the example above, **x** is a constant. The syntax is similar to the variable except for the addition of the reserved word `const` and the requirement to initialize the constant, as seen in the example.

2. Expressions and Statements

   A statement in C is considered to be a single sentence that can stand on its own. Statements are identifiable because they always terminate with a semi-colon. In the program above, `y = y + 1;` is a statement. Notice that a statement can appear on a line alone or with other statements (on the same line), as in the example above `y = y +1; y = x + z;` This, though, is not considered good coding practice. Mathematical expressions have the following syntax: `VARIABLE = EXPRESSION;` where expression is a standard mathematical expression, as seen in math courses. A math expression follows all the standard calculation rules: multiply (*) and divide (/) are processed first, and then addition (+) and subtraction (-) are processed next. You can force the order of processing by using round brackets; for example: `x = y * (a + b) / 3;` says that the computer with do the addition first, then it will multiply, after that divide by three, and finally, the result will be stored in x.

3. Printf displays the results from all variables and the constant to the screen. We will look at the syntax of printf is detail in example 3, below.

Syntax Details for Example 2

Literals

For integers: 5, –10, and 3000 are examples of integer literals. They are composed solely from digits and an optional proceeding positive or negative sign. Notice that no decimals or commas are used. These are stored in memory as 16-bit signed integer numbers (the leading bit is used as the sign bit).

For characters: 'a', 'B', '7', and '&' are examples of character literals. They are strictly single-character values sandwiched between single quotation marks. As in the example, a character is not only a letter but can be any symbol or digit from the keyboard (actually, the Extended ASCII table). It is important to note that characters are stored in memory as 8-bit unsigned integer numbers; i.e., no sign bit is used. These numbers are based on the ASCII table's assignment of integer values for each character. Each number represents a binary code used by the computer to store that data in memory. For example, 'A' is 101, ' ' is 40, and '9' is 71.

For real numbers: 1.0, –52.749, 5., and 70918.7723 are examples. They are composed of digits with an optional preceding positive or negative symbol. These numbers must include a decimal point (i.e., the period on the keyboard) and optionally have a fraction composed of digits. Commas are not used. These numbers are stored in memory as 16- or 32-bit values The leading bit is the sign bit, and the remaining bits are divided into two groups: the mantissa and the exponent.

The string: "house", "My name is Bill", and "%>&*#" are examples of strings. They are composed from any sequence of characters. The double quote begins and terminates a string. Each individual string element is a character. The string is not a built-in type, but a special construction used to group characters together in memory as a contiguous sequence of characters terminated by a special character called the NULL character, represented as '\0'. You do not need to write this character. It is automatically added to the end of the string. For example, the string "bob" is represented in the string as: 'b', 'o', 'b', '\0' where each character is actually stored as the ASCII values: 142,157,142,0 (the commas are not stored—I use it here for clarity in the text).

Identifiers

Identifiers must begin with a letter or the underscore character (i.e., _), which can be followed by any combination of letters, digits, and the _ character. C is case sensitive. Therefore, words spelled the same but in different case are treated as different identifiers.

Example of legal identifiers:	X	sum	Sum3	_total
Example of illegal identifiers:	4times	total-sum	xy&z	

Example of identifiers that are viewed as different in C even though they are spelled the same:

sum	Sum	sUm	SUM	

Types

Description	Type	Bits	Range
Integer	`short, byte`	8	– 128 to + 127
	`int`	16	+/– 32,768
	`long`	32	+/– 2,147,483,648
Floating Point	`float`	32	+/– 3.4 x 10^{38} (with 7 significant digits)
	`double`	64	+/– 1.7 x 10^{308} (with 15 significant digits)
Boolean	`short, int, long`		0 is false, other true
Character	`char`	8	0 to 256
String Ptr	`char *`	32	address in memory (special case of pointers)
Pointers	TYPE*	32	address in memory pointing to TYPE

Constants

Syntax:

```
const MODIFIER TYPE IDENTIFIER = VALUE;
```

Where:

const	is the reserved word designating that the stored value is unchangeable
MODIFIER	any built-in type modifier (it is optional)
TYPE	any built-in or user defined type
IDENTIFIER	any legal user defined identifier
VALUE	the literal that will be assigned to the IDENTIFIER

For example:

const int limit = 100;

This defines a constant called limit that contains the number 100. The value in limit cannot be changed. The constant value can only be initialized when the constant is defined. The initialization cannot be postponed to a later time. The constant *limit* can be printed, assigned to another identifier, and compared with other values, but it cannot be changed.

Variables

Syntax:

MODIFIER TYPE IDENTIFIER = VALUE, ...;

Where:

MODIFIER	any built-in type modifier, like `short`, `unsigned`, and `long` (it is optional)
TYPE	any built-in or user-defined type

IDENTIFIER	any legal identifier
VALUE	can either be a literal or another identifier
, ...	indicates that any number of variables can be defined

Each identifier can be optionally initialized to a value.

Example:

```
int x;
int z = 5;
int y = 2, h = 3;
int sum = 0, average;
```

Variables are constructs that allow their stored value to change. Their value can be optionally initialized when defined (as with **int sum = 0**), or they can be initialized later (as with **int x**). If the variables are initialized later, the compiler will assign a default value to the uninitialized variable. Normally for numbers, this would be zero and for characters, this would be the space; but, this is not guaranteed. In any case, the value stored in variables can be modified as often as needed.

Statements and Expressions

Legal Operators:		
Standard ones:	+(add), –(minus), * (multiply), /(divide), =(assign)	
Modulo: %		
Short hand:	++(increment), ––(decrement) +=, –=, *=, /=, and %=	
Examples:		
X=X+1;	Same as:	X++;
Y=Y* 5;	Same as:	Y*=5;
X=X/(5+Y);	Same as:	X/=(5+Y);

Syntax:

VAR = RHS;

Where:

VAR is always a variable.

RHS is any legal mathematical or logical expression.

Legal mathematical expressions can be:

x = 5 + 2;	Stores 7 in x
x = x + 3 * y;	Multiplies y and 3 first, and then adds to x and stores result in x
x = (x + 3) * (y / z);	Adds x and 3 first, then divides y and z, and then multiplies results
x = y = 2;	Multiple assignments. Assigns 2 to y and then the value in y (2) to x

EXAMPLE 3: INPUT AND OUTPUT

```
#include <stdlib.h>
#include <stdio.h>
/* ============================================================
   This program shows how to use standard input and output
   functions. These functions are contained within stdio.h
============================================================ */

int main (void)                                                    (3)
{
   double product, tax, total, received, change;

   printf ("Please enter the product cost: ");
   scanf ("%f", &product);
   printf ("What is the tax rate: ");
   scanf ("%f", &tax);

   total = product * ( 1.0 + (tax / 100.0));

   printf ("Please pay $ %.2f\n Amount received: ",total);          (1)
   scanf ("%f", &received);                                        (2)

   change = received - total;

   printf("Your change = %.2f\n", change);

   return 0;
}
```

This is our first pr ogram that does something pr actical. It demonstrates how to construct a program that carries out an input-calculation-output problem. In this example, the program assumes someone wants to purchase something. It asks for the amount and applicable tax rate. It responds by declaring the total that should be paid. It gives place to input the actual amount paid. It then ends by displaying the change to be returned to the purchaser.

1. Point 1 demonstrates how the printf can be formatted using special escape sequences and control codes. C uses the same control-codes and escape-sequences for both the printf and scanf commands. The percent (%) and the backslash (\) are called escape characters. Normally, printf and scanf interpret their argument string as is,

except when it comes across an escape character. An escape character tells C that the characters following are codes that require special interpretation. In the case of the backslash, the next character following the escape character is the **escape-code**. In this example, we have \n; the code is the letter n, which means carriage return and line feed. The escape character percent indicates that a specially formatted set of characters will follow. In this example, %.2f is the **escape-sequence**. It consists of the escape character, %, and the escape-sequence, .2f. The escape-sequence indicates that the output will be a floating-point number (f) with two decimal places (.2). There is no limit to the size of the whole number (more below). Printf is defined within stdio.h.

2. Scanf is the input command; it is used to read data from the keyboard. Its string argument defines the type of information to read, and it comes first. Following this is a list of arguments, each proceeded by the ampersand (**&**), which are the variables where the keyboard data will be stored. The escape-sequence and the ampersand-variables must match in type and in the order that they appear from left to right. In our case, %f indicates that we are inputting a floating-point number in any legal form, and `&total` is a variable of type **double**. The **double** and the %f match. Scanf is defined within stdio.h.
3. Comments in C are traditionally represented with the escape-sequences slash-star (/*) to start the comment and star-slash (*/) to end the comment. As shown in example 1 and here in example 3, comments can be on a single line or on multiple lines. They must be bracketed with the comment escape-sequences. Today, many C compilers are implemented with C++ engines. In C++, comments are represented with a double-slash (//). Therefore, many of today's modern compilers accept both comment escape-sequence forms. To be consistent with C, we will maintain the slash-star and star-slash for our examples.

Syntax Details for Example 3

The `printf` Statement

Syntax:

printf (DESCRIPTION, ARGUMENTS);

Where:

ARGUMENTS is optional and consists of one or mor e literals, variables, or expressions to be displayed on the screen. The arguments are entered as a comma-separated list.

DESCRIPTION is a m andatory string. It contains both text to be displa yed as well as formatting codes of two forms: escape-characters and escape-sequences.

Escape-Characters:

\n	Carriage return & line feed (new line)	\r	Carriage return
\f	Form Feed	\t	Horizontal tab
\a	Announce (beep)	\b	Backspace
\\	Displays the Backslash	\"	Displays the double quote
\o##	Bit pattern where ## is 2 octal digits proceeded by the letter o, like \030		
\x##	Bit pattern where ## is 2 hex digits proceeded by the letter x, like \x2F		

Escape-Sequences:

%#d	int	%#g	Use %e or %f, whichever produces a shorter output
%#c	char	%#e	Printed in the form: [–]m.nnnnnnE[–]xx (where m, n and x are digits)
%#s	String	%#o	Octal
%#f	float or double	%#u	Unsigned decimal
%#x	Hexadecimal	%#l	Long

All the escape-sequences can be optionally adjusted using the formatted text **%[–]n.m**. Note, this has been designed by the # symbol, above. The **minus sign** signifies reverse justification. The **n** signifies the minimum width reserved for output. The **.m** signifies the text precision. For example: %-5s says a string of length 5 will be outputted using right justification (the standard justification for strings is left).

The `scanf` Statement

The **scanf** function operates similarly to printf but is for input from the keyboard.

Syntax:

scanf (DESCRIPTION, ARGUMENTS);

Where:

ARGUMENTS is a mandatory comma-separated list of pointers to variables. There can be one or more arguments. The pointers can be written as a *simple* variable preceded by an ampersand (&)—more discussion about this will be presented later.

DESCRIPTION is a mandator y string describing the input for mat, consisting of value descriptors in the form of escape-sequences.

For example:

```
scanf ("%c", &letter);                    (A)
scanf ("%d %f", &number, &real);          (B)
```

In the first example (A), the `scanf` is waiting for a character to be read from the keyboard. The enter key must be pressed to indicate that the input is complete. If the enter key is not pressed, then the program waits until it is pressed. The input value is stored within the variable **letter**. In

the second example (B), two numbers must be input, separated by a space. The enter-key must be pressed to indicate that the input is complete. If the enter key is pressed before the second number, the `scanf` will still wait for the missing value to be inputted and for the en ter key to be pressed again. No message is displa yed, indicating that the program is waiting for mor e input. The program simply displays a blinking cursor. The first input value is stored in variable **number,** while the second input value is stored in the variable **real.**

The library stdio.h has many more input/output functions. We identify a few here:

Syntax	Example	Description

gets (char *)
puts (char *)
	char array[100];
	printf("Name: ");
	gets(array);
	printf("You said: ");
	puts(array);
		The functions **gets** and **puts** are string I/O functions that use arrays or string pointers. They do not check for the array out of bounds error.

int getch ( )	char c;	**getch()** reads a single character, but does not echo to screen.
int getche ( )	c = getch();	**getche()** reads a single character and echoes it to the screen.
int getchar ( )		**getchar()** functions like gets() but returns a single character at each call.
sscanf (char *, FORMAT, VARS) sprintf (char *, FORMAT, VARS)	char array[100]; int x;	These two useful functions help perform string format printing.
	sscanf(array, "%d", &x); sprintf(array, "%d", x);	They function exactly like **scanf()** and **printf(),** except that the input and output are carried out from the array. **sscanf** reads from and **sprintf** writes to the array.

Reading from and Writing to Files

To access files for either reading or writing, C requires a special file-pointer type called FILE. A pointer of this type can reference any file on the disk. The syntax for declaring a variable of type FILE is given as: `FILE *INDETIFIER;` for example: `FILE *ptr;` declares that the variable ptr is a pointer to a file. The asterix states the variable ptr is a pointer.

Connecting the poin ter to an actual file requires a call to the `fopen` function. The function `fopen` has two str ing arguments: the file's name and the mode of access . The file name can include the path to the file. For example: "C:\MY DOCS\STUFF.TXT" is an example of a pa th and file name. Without the path, `fopen` assumes the current working directory. You can open a file in read-text mode ("rt"), in write-text mode ("wt"), in append-text mode ("at"), or in simultaneous read and wr ite text mode ("r+t " or "w+t"). You can r eplace the 't' with 'b' to r ead,

write, append, or read/write in binary mode. The file must exist for read mode. If it does not exist, then `fopen` returns a NULL. If you open a file in write mode and it exists, `fopen` will first delete the file and then open a new version of the file in write mode. Append mode is like write, but the file is not deleted. Instead, you add characters to the end of the existing file. If the file does not exist, append creates a new file in write mode. The syntax for fopen is : `IDENTIFIER = fopen(FILENAME, MODE);`

There are many library functions to read from and write to files: `fgets, fputs, fscanf, sprintf, fgetc, fputc, fread, fwrite,` etc. They all operate similarly to their console versions (which we have already discussed above) with only minor syntax changes.

The function `fclose(ptr)` is used to close the file.

You can position your **reading cursor** in a file with `fopen` and `fseek` functions. The function `fopen` always positions you at the first character or byte in the file. The function `fseek` positions the cursor anywhere in the file. Once in the new position, the program proceeds from that point in the file.

Here is an example of how to do some file manipulation:

```
FILE *in = fopen("names.txt","rt"), *out = fopen("backup.txt","wt");
char buffer[100];

if (in == NULL || out == NULL) { fclose(in); fclose(out); return; }

while(!feof(in))                /* while we have data */
{
 fgets(buffer,100,in);          /* read the data from one file */
 fprintf(out,"%s",buffer);      /* and write it to another file */
}
fclose(in);                     /* close our files when done */
fclose(out);
```

In the above example, two files are opened, in and out. The pointer **in** is connected to the file "names.txt" and it is opened for reading. The pointer **out** is connected to the file "backup.txt" and it is opened for writing. No paths were given, so the program will access both of these files from within the same directory from where the program came. If the file "backup.txt" already exists, it will be automatically deleted. An array is also created to store information we will read. Before any work is performed, the program tests to see if the input and output pointer were connected to the requested files. If fopen failed, the pointers would have been initialized to NULL. If there were no problems, the while loop repeats until the input pointer gets to the end of the input file (i.e., !feof(in) means not eof-of-file for pointer **in**). Each time it loops, it reads a line of text into the array buffer and echo buffer to the screen. When the loop is done, both files are closed.

EXAMPLE 4: FLOW CONTROL STRUCTURES

```
#include <stdio.h>
int main (void)
{
   double product, sum = 0.0, tax, total, received, change;
   char more = 'y';

   while (more == 'y')                                                    (2)
   {
      printf ("Please enter the product cost: ");
      scanf ("%f", &product);

      sum += product;

      printf ("More? (y/n): ");
      scanf ("%c", &more);
   }

   printf ("What is the take rate: ");
   scanf ("%f", &tax);

   total = product * ( 1.0 + (tax / 100.0));

   printf ("Please pay $ %.2f\n Amount received: ");
   scanf ("%f", &received);

   if (received >= total)                                                 (1)
   {
      change = received - total;

      printf("Your change = %f\n", change);
   }
   else printf("You did not provide enough money!\n");

   return 0;
}
```

Control structures are those sta tements in a pr ogramming language th at define how a language proceeds to the next instr uction after it h as just finished executing a pr evious

instruction. C's default control structure is **sequential flow**. This means that once the program finishes executing an instruction, the next one will be the instruction immediately below or to the right, as in reading English text. C has three other control structures: **decision, iterative, and jump**. **Decision flow** is represented by three statements: the if-statement, in-line-if statement, and the switch-statement. **Iterative flow** is implemented by three statements: the for-loop, while-loop, and the do-loop. **Jump flow** is implemented in many ways and will be looked at later.

The above example program expands on example 3 by allowing users to enter more than one product and validating if the amount received was greater than what they have to pay.

Below is discussed the two control structures from the example program:

1. The above example uses the C if-statement in its block-structured form. It is testing whether **received** is greater or equal to **total**. If this is true, then the two statements the change calculation and the display on the screen will be executed; otherwise, an error message is displayed.
2. The above example shows the while-loop. This loop continues to iterate as long as the variable **more** is equal to the lowercase letter 'y'. As soon as this is not the case, the loop ends.

C control structures can be expressed as **singular statements** or as **block statements**. In the case of the example if-statement, we find its block form. Its singular version would consist of a single statement with no curly brackets and would look like this:

```
if (received > total)change = received - total;
```

When a control structure is in its singular form, it can only control one statement. The block form has no limit to the number of statements in can control within the curly brackets. The open and close curly brackets are also known as **block scope**.

Syntax Details for Example 4

Decision Control Statements

Decision control statements use **conditions** to determine when the decision control is true or not. Conditions are also known as **logical expressions.** Logical expressions are defined as follows:

Logical expression

Syntax:

```
EXPR LOP EXPR          or              LEXPR
```

Where:

EXPR is any legal mathematical or logical expression including function calls. The function calls must return a result that can be used in the expression.
LEXPR is any legal logical expression including function calls. The function calls must return an integer result.
LOP is any legal logical operator. These are defined below:

Logical Operators:

<	Less than	>	Greater than	<=	Less or equal
>=	Greater than or equal to	==	Equal to	!=	Not equal to
&&	Logical AND	||	Logical OR	!	Logical NOT

Logical Expression examples:

`z < 2`	**(A)** Tests z to see if it is less than 2
`c != 'A'`	**(B)** Checks to see if c is not equal to 'A'
`x < 2 && c == 'A'`	**(C)** True only when x is less than 2 and c is equal to 'A'

The in-line-if decision statement

C has a simple decision structure called the in-line-if. It is useful when you want to perform an either-or calculation based on a simple logical decision. In other words, if the logical expression is true, then you do a particular calculation. If the logical expression is false, then you do another calculation. The calculation must return a result. The result is assigned to the variable or statement in which the in-line-if is imbedded. Here is its syntax:

Syntax:

```
LHS = (COND) ? TRUE_STATE : FALSE_STATE;
```

Where:

COND is any legal logical expression.
TRUE_STATE is a single calculation that is executed when COND evaluates to true.
FALSE_STATE is a single calculation that is executed when COND evaluates to false.
LHS = is optional and will be assigned the result of the TRUE_STATE or FALSE_STATE.

For example:

```
int x = 0;
x = (x < 5) ? 10 : 100;                          (A)
printf("%d", (x == 0) ? x*10 : 0);               (B)
```

In the example above, statement (A) shows how a new value can be assigned to x based on the condition that x is less than 5. If it is less than 5, then x will be assigned the value 10; otherwise, it will be assigned the value 100. In this example, x would receive 10. The second example

(B) combines the in-line decision statement with a **printf** statement. Here, if x is 0, then x is multiplied by 10 and displayed; otherwise, 0 is displayed. In this case, the number zero will be displayed because x changed to the number 10 in statement (A).

The if-statement

The if-statement is the building block of many programming languages. Its purpose is to allow the user to select between two execution paths based on a condition. The if-statement can appear is a simple form or a blocked form. In the simple form, the if-statement selects between two single statements. In the blocked form, multiple statements can be enclosed in curly brackets. If the condition is true, then the true statements are executed; otherwise, the false statements are executed. The false statements are optional. This means the **else-block** can be left out.

Syntax:

```
if (COND) TRUE_STATEMENT;
else FALSE_STATEMENT;

if (COND) {
   TRUE_STATEMENTS;
} else {
  FALSE_STATEMENTS;
}
```

Where:

STATEMENT is any legal single general statement.

STATEMENTS are one or more legal general statements.

COND is any legal logical expression.

Note the else-block is optional in both versions of if.

Blocked Statements

The curly brackets { and } are used to define a block of statements that belong together. This block can then be placed where a single statement would normally be placed.

Examples:

```
if (age < 20) printf("welcome\n");                                    (A)

if (sum < 100 || sum > 500)                                           (B)
    counter++;
else
    printf ("Illegal series");

 if (balance < 0)                                                     (C)
{
       printf ("Your withdrawal amount is too large.\n ");
       printf ("Please try again.");
}
```

The first statement (A) displays the message `welcome` and then a new line only when age is less than 20. The second statement (B) increments the variable **counter** when **sum** is less than 100 or when **sum** is greater than 500. If this condition is not tr ue, i.e., **sum** is between, and including, 100 to 500, then the message `Illegal series` is displayed. The last statement (C) displays two messages `Your withdrawal amount is too large`, then a new line, and then the next message `Please try again` (since they have been blocked together using the curly brackets), but only when the balance is negative.

The Switch Statement

The switch-statement is a useful, if cumbersome, statement. Its purpose is to handle those situations where you may have many if-statements following one another in a long series of optional execution pathways. In this situation, it would be nice th at a control statement would exist that allows you to structure a statement that provides multiple execution paths based on a single condition. This is wh at the switch-statement does. Look at the syntax:

Syntax:

```
switch (I_IDENTIFIER)
{
    case I_LITERAL1:
        STATEMENT(s);
        break;
    case I_LITERAL_n:
        STATEMENT(s);
        break;
    default:
        STATEMENT(s);
}
```

Where:

STATEMENT(s) are zero or more legal statements.
I_IDENTIFIER is any integer or character identifier.
I_LITERAL_n is any integer or character literal.
(Each **case** must have a different literal.)

There are some limitations. For example, the I_IDENTIFIER can only be integer or character. The I_IDENTFIER is not a logical expression but an s simple equivalence expression. In other words, I_IDENTIFIER can only equal one of the I_LITER ALs. You cannot ask gr eater-than or less-than questions, and there is no and-or oper ators. But, given this simplicity, you can still do a lot. For example, the `break` is optional. Normally, the `break` identifies where the `case` ends. If you lea ve it out, then the pr ogram executes immedia tely into the next `case` block without performing a condition test. This behaves like an or-operator. If we lea ve out the break-statement between the I_LITER AL1 and the I_LITER AL_n in the syn tax above, then the statements in the I_LITER AL_n block will be executed for both I_LIN TERAL1 and I_LITERAL_n.

Example:

```
int x;
scanf ("%d",&x);
switch (x)
{
 case     0:
          printf ("No customers");
          break;
 case     2:
          printf ("Max");
 default:
          printf ("Illegal value");
}
```

In this example, the user enters an integer number from the keyboard into variable x. This variable is then used in the switch-statement. There are three cases in this switch-statement. The first tests x's equivalence to zero; the next one tests x with integer 2; and the last case catches all values of x that were not zero or two. The statement `case 0:` executes two inner statements. The `printf` displays - No customers - and then breaks the switch-statement (i.e., completes execution of the switch and proceeds to execute the first statement after the switch-statement block). The statement `case 2:` has only a single printf-statement, and it displays the word - Max -. Since there is no break-statement, the `printf` in the `default:` is also executed (resulting in the error message displaying), and then execution proceeds after the switch-statement block.

Iterative Statements

Iterative statements are the next class of control structures. These allow the program to repeat statements multiple times. The repetitive nature of these statements is controlled by a logical expression. As long as the expression is true, then the loop continues.

The while-loop

The while-loop is the simplest loop in C. The statement (in the simple form) or statements (in the blocked form) are repeated as long as the logical condition is true. Once the condition becomes false, then the loop ends and execution continues with the first statement after the while-loop.

Syntax:

```
while(COND) STATEMENT;
```

Or

```
while(COND) {
    STATEMENT(s);
}
```

Where:

COND is any legal logical expression.
STATEMENT is zero or one legal statement.
STATEMENT(s) is zero or more legal statements.

Example:

```
scanf("%c",&code);
while(code != 'A')
{
    count++;
    scanf("%c",&code);
}
```

In the example above, the program reads a character from the keyboard until the capital letter 'A' is entered. Each time a letter is entered, a counter variable is incremented, keeping track of the number of letters entered before 'A' was input.

The for-loop

The for-loop can be used in multiple ways. Its most common format is as a counting loop. The for-loop will start counting from an initial number and increment to the last number you specify. It will increment based on a mathematical expression you provide. In the simplest case, you could ask it to start from the number zero and count to the number 10 incrementing by 1 each time it loops. But, you could be more adventurous and try more complex expressions. The for-loop permits you to provide complex mathematical and logical expressions. This transforms the for-loop into a more robust iterator. Look at its syntax:

Syntax:

```
for(INIT_LIST; COND; INC_LIST) STATEMENT;
```

Or

```
for(INIT_LIST; COND; INC_LIST)
{
    STATEMENT(s);
}
```

Where:

INIT_LIST is a comma-separated list of initialization statements.
COND is any legal logical expression.
INC_LIST is a comma-separated list of mathematical expressions.
STATEMENT is zero or one legal statement.
STATEMENT(s) are zero or more legal statements.

Example:

```
for(i=0; i<10; i=i+1) printf("%d\n",i);                    (A)
```

```
for(i=0, j=10, i<10 && j>0; i=i+2, j=i*(j-1))                    (B)
{
     if (j>i) j = i;
     printf("%d %d\n", i, j);
}
```

In the above example (A), this loop starts by initializing 'i' to zero and then progressively increments it (i=i+1) until it reaches 10 (i < 10). At this point, the loop terminates. Within the loop, the printf is executed 10 times with 'i' having the values: 0 to 9. The variable 'i' is incremented to 10, but the loop condition prevents execution of the printf by terminating the loop (since the condition i<10 fails). Hence, the number 10 is never printed.

In example (B), a more elaborate for-loop is presented. Two variables are initialized: i starts with zero, and j starts with 10. The stopping condition makes sure the loop ends then either i becomes 10 or j becomes zero. The variable i increments by 2, while j increments in a more complex fashion. The body of the for-loop is in block-form and therefore has more than one statement. Notice that the if-statement modifies the j variable. This is permitted in C.

The do-while Loop

The do-while loop is similar to the while-loop . The only difference is when the condition is tested. In the while-loop, the condition is tested first before the statements within the loop are executed. In the do-while loop, the opposite is true, the statements are executed first, and then the condition is tested. There are many examples where this is needed: menu or password programs are examples. In a menu program, you would like to display the menu and ask the user to input a menu selection, at least once before terminating the loop. The same reasoning goes with password loops. You always want users to input the password at least once. Maybe you would give them three chances before kicking them out, but the first chance is always given. Look at the syntax below:

Syntax:

```
do {
  STATEMENT(s);
statements.
} while(COND);
```

Where:

COND is any legal logical expression.
STATEMENT(s) is zero or more legal general

Example:

```
do {
     printf("Menu: (1)Days in your life, (2)Guess password, (0)Quit\n");
     printf("selection : ");
     scanf("%d",&choice);

     switch (choice)
```

```
    {
          case 1:
                printf("What is your age?: ");
                scanf("%d", &age);
                printf("Days alive, without leap years: ");
                printf("%d\n", age*365);
                break;
          case 2:
                do {
                      printf("Guess password: ");
                      gets(buffer);
                      if (strcmp(buffer,"abc123")!=0)
                           printf("Incorrect\n");
                      }
                      while(strcmp(buffer,"abc123")!=0);

                      printf("Good! You guessed it!\n");
                      break;
    }
} while (choice != 0);
```

In this example, a simple menu is displayed asking the user to select from three options. The first option displays the number of days, without leap years, the user has been alive. The second option is a simple game requiring the user to guess a password. In this case, the password is "abc123". The last option stops the loop.

EXAMPLE 5: ARRAYS AND STRINGS

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

int main (void)
{
  char sentence[80], key2[5], output1[80], output2[80];                (1)
  int key1, j, k;

  printf ("This will encrypt your sentence using two methods.\n");
  printf ("Please enter your sentence (up to 79 characters): ");
  gets ("%s", sentence);

  printf ("A Caesar cipher uses an integer number to scramble letters.\n");
```

```
printf ("Please enter an integer number to scramble the letters by: ");
scanf ("%d", &key1);

printf ("A modified Caesar cipher uses a word to scramble the letters.\n");
printf ("Please input a 4 characters to scramble with: ");
scanf ("%s", key2);
for (j = 0, k = 0; j < 80 && sentence[j] != '\0'; j++, k = (k+1)%4)
{
   output1[j] = (sentence[j] + key1) % 256;                          (2)
   output2[j] = (sentence[j] + key2[k]) % 256;
}

printf ("Your input sentence is %s", sentence);
printf (" and key1 is %d while key2 is %s.\n", key1, key2);
printf ("The cipher with key1 is %s ", output1);
printf ("and with key2 is %s.\n", output2);
return EXIT_SUCCESS;                                                 (3)
}
```

In C, there is no string-data type. Instead, C takes advantage of some special properties of the character-data type and the array **data structure**. A data structure allows data types to be combined into novel ways. The **array** is a structure that permits the association of identical data types into a fixed contiguous space in a list-like manner. Each item in an array can be accessed sequentially or randomly. A string-data type is a sequential list of characters with a fixed length in a contiguous space. This sounds and looks remarkably similar to a character array. Because of this similarity, C does not have a string-data type, but instead, uses the array. Thankfully, C does give us some additional tools to make string manipulation easier. We saw an example of this with `strcmp` in the previous example. There is an entire library, string.h, that contains functions that know how to manipulate strings as character arrays.

The above example demonstrates how to use arrays, strings, and characters to encrypt a sentence inputted by the user using two methods.

1. C arrays are defined similar to other variables, with one addition: square brackets to define the size of the array. The number in the brackets indicates the number of **cells** the array has. Each cell has the same type and can store one value. In the above example, all the arrays are type char, and therefore, each cell of these arrays can only store a single character. Multi-dimensional arrays can be defined by repeating the square brackets as in char sentences[80][10] This defines 10 sentence rows of 80 characters each, maximum. Keep in mind that C does not do array-range checking. This means that you can specify an incorrect cell index position, and C will try to access that location—even if it is not in the array—be careful!

2. Since characters are stored as ASCII integer numbers in C, we can easily scramble the letters by adding or subtracting an integer number to them. This new integer number will be the ASCII code of some other symbol. An important thing to remember is that integer numbers can have values from negative to positive 32,768. ASCII has values from 0 to 255. The example statement uses modulo arithmetic to ensure the resultant is still within the ASCII range.
3. EXIT_SUCCESS is an integer value defined in the stdlib.h include file. It is the standard way to say that your program terminated correctly. If your program is terminating with a problem, you can use EXIT_FAILURE.

Syntax Details for Example 5

Arrays

A C array is a structure composed of two elements: a reference pointer and a set of contiguous memory locations. The array's identifier is actually a C pointer and not the actual array structure. The array is an unnamed fixed-sized contiguous block of memory with a pointer being the only reference to the structure. The identifier (pointer) references the first cell of that block of memory. To create an array, you simply declare the type and identifier name you would like to use, along with the size of the array. The computer does the rest. You can identify and use any of the cells in the array by specifying the cell's index number. C array index numbers are integer numbers starting from zero and incrementing by one. For example, if your array has 10 cells, then the index numbers for each cell are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, in that order. The syntax is shown below:

The Array

Syntax:

```
TYIPE ID[SIZE];
Or
TYPE ID[SIZE1][SIZE2]...
[SIZEn];
```

Where:

TYPE is any legal type.
ID is any legal identifier.
SIZE is the number of cells in the array (indices: 0..SIZE-1).
[SIZEi] is a dimension of the array.

Example:

```
int x[10];                                    (A)
x[2] = 35;                                    (B)
```

gives:

Line (A) constructs an array reference named "x" pointing to an array with 10 cells where each cell can store one integer number. Line (B) places the number 35 in cell 2.

0	1	2	3	4	5	6	7	8	9
		35							

Example:

```
float y[10][5];
y[5][3] = 12.4;
```

gives:

A two-dimensional array named "y" having 50 cells where each row has 10 cells. Each cell stored one float number.

Y	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3						12.4				
4										

Example:

```
main(void)
 {
    int values[10];
    int i, sum=0;

    for(i=0; i<10; i++){
        printf("%d:",i);
        scanf("%d",&values[i]);
    }
    for(i=0; i<10: i++) sum += values[i];
    printf("Sum is %d\n", sum);
 }
```

In the example above, the one-dimensional array "values" receive 10 integer numbers inside the for-loop. Each integer is placed in its own cell. A second for-loop sums all the values. Then the sum is displayed.

Strings

Compared to more recent languages, C's implementation of strings is primitive, but its simplicity is its beauty. In reality, C does not have an implementation for strings. Instead, it simulates strings using arrays. In C, a string is defined to be a contiguous block of characters terminating with a special end-of-string character, called the null character. The null character is expressed in an escape-character-sequence: '\0'. The single quotes are how we designate a character literal and the \0 (back slash zero) designate the null character. Since in C, all characters are stored as ASCII integer codes, the string is a contiguous block of ASCII codes terminating with the 8-bit binary number zero (known as the null character).

There are three ways to declare a string in C: as an array, as a pointer-block combination, or as a literal. The array version of strings and the pointer-block version are very similar except that we manipulate one implementation using array syntax and the other using pointer syntax. They both look the same in memory and suffer from similar limitations and liberties.

To define a string as an array, we declare a regular character array, for example, char array[20]. This is just a normal character array with 20 cells. To distinguish this array from any other character array, we must store our string with a null character terminating the sequence of characters—nothing more. Therefore:

```
array[0] = 'a';
array[1] = 'b';
```

is an array with two characters. By adding a null character in the third cell:

```
array[2] = '\0';
```

we have transformed this to a string! What this really means is that a C string function will now know how to use your character array properly because it will now know where the string ends.

To define a string using pointers, you do this:

```
char *x = "This is my string";
```

There is no array index to let us travel through this string, but the pointer **x** points to the first character 'T' in the string and can be used by the string library functions. Since we used the double quotes, the complier automatically adds the null character. The double-quoted string is itself an example of a string literal. String literals are constants and cannot be changed.

EXAMPLE 6: ABOUT BITS AND BOOLEAN

```
int main (void)
{
     unsigned int flag = 6; /* i.e. 0000000000000110 */
     unsigned int mask = 5; /* i.e. 0000000000000101 */
     unsigned int result;

     result = flag & mask;                                          (1)
     printf("%d", result); /* output is 4, i.e. 0100 */

     result = flag ^ mask;
     printf("%d", result); /* output is 3, i.e. 0011 */

     result = flag | mask;
     printf("%d", result); /* output is 7, i.e. 0111 */
```

```
    result = flag << 1;
    printf("%d", result); /* output is 12, i.e. 1100 */

    result = flag >> 1;
    printf("%d", result); /* output is 3, i.e. 0011 */

    if (result % 2) printf("Odd number.");                    (2)

    return EXIT_SUCCESS;
}
```

In the last example, we saw that C does not have a string data type, but we could simulate it using character arrays. C also does not have a boolean data type. Instead, it uses the integer data type to simulate boolean. This decision has some advantages. We can do integer mathematical calculations and use the results as boolean solutions. This was a novel idea for the time and advantageous to do even now.

The integer data type can also be manipulated at its bit level using bit-operators. For example, the integer number five looks like this: 5. Its bit counterpart in binary looks like this: 00000101, with eight more leading zeros not shown here (since the integer data type is 16 bits long). Each bit can be thought of as a boolean value: 1 for true and 0 for false. A 16-bit integer number could then be used to represent 16 boolean values, if represented in bits.

The example program presented here performs binary bit operations on the variables **flag** and **mask**. The answer is placed in the variable **result**. The final result value is then tested to see if it is an even number. C permits you to access the bits within a variable using as simple masking technique:

1. Bits can be manipulated with the bit operators: &, |, ^, <<, and >>. You cannot directly influence a bit; but, you can take an entire variable and mask it. The mask protects some of the bits in the variable from changing, while giving you access to the remaining bits. The bit operator & performs a pair-wise bit AND operation between two matching variables' bits (i.e., the first bit of one variable with the first bit of the second variable, then the second bit from each variable, etc). The OR bit-wise operation symbol is |. The ^ is XOR and >> is shift right and << shift left. The comments in the program show what the results of the operations will be.
2. C has no boolean type. Instead, it uses int. This provides for interesting capabilities. The number zero is treated as false and any other number as true; we can do mathematical computations in the place of conditional questions. The result of the mathematical operation will be the boolean result. If the answer is zero, then it is false; otherwise, it is true. In the example above, modulo 2 is only zero when the result is an even number.

Another common application for bit operations is writing software that manages peripheral devices (printers, mice, robots). Every peripheral device has a control chip that serves as the peripheral's brain. That control chip has small memory locations called **registers** that the programmer can manipulate. These registers are limited in size and therefore are limited in what they can store and represent. It is a common engineering technique to associate the bits in a register to different peripheral actions. This means that if a particular bit is set to 1, then that would trigger a particular function. If the same bit is set to 0, then the device will stop performing that function. If a peripheral's register is 16 bits long, then each of those bits could represent a separate action. Setting the bits to 1 and 0 would cause the peripheral to perform actions at your request.

Bit operators are very useful.

Syntax Details for Example 6

Bit Operators

Bitwise AND Expressions

Syantx: `EXPRESSION & EXPRESSION`

Where: EXPRESSION is any valid C numerical expression.

Example: `x = 0110 & 0101;` Gives 0100 in x.

The resultant bit contains a 1 only when the corresponding bits (i.e., bits in the same position) in the two EXPRESSIONs are set to 1; otherwise, the bit is set to zero. The ones behave as a protective mask. Zero forces a clear. In the example above, the two expressions have a 1 in the second position; this is why the second bit in the resultant is set to 1 and all others are set to 0.

Bitwise OR Expressions

Syntax: `EXPRESSION | EXPRESSION`

Where: EXPRESSION is any valid C numeric expression.

Example: `x = 0110 | 0101;` Gives 0111 in x.

The resultant bit contains a 0 only when the corresponding bits in the two EXPRESSIONs are set to 0; otherwise, the bit is set to 1. The zeros behave as the protective mask. The number 1 forces a set.

Bitwise EXCLUSIVE OR Expressions

Syntax: `EXPRESSION ^ EXPRESSION`

Where: EXPRESSION is any valid C numerical expression

Example: `x = 0110 ^ 0101;` Gives 0011 in x.

The resultant bit contains a 1 in a certain position only when exactly one of the corresponding bits in the two EXPRESSIONs is set to1. The protective mask in this case is a bit more complicated. Zero is the protective mask, while 1 causes the corresponding bit to toggle. Toggle means 1 changes to 0 and 0 changes to 1.

Bitwise LEFT SHIFT Expression

Synatx: `EXPRESSION << EXPRESSION_N`

Where: EXPRESSION is any valid C numeric expression.

EXPRESSION_N is any valid C integer expression.

Example: `x = 0110 << 1;` Gives 1100 in x.

Bits in the EXPRESSION are shifted to the left by EXPRESSION_N bit positions. Vacated bits are filled with zeros. Bits passing the left boundary of the number are discarded. If the EXPRESSION is a signed number, then shifting does not affect the signed bit.

Bitwise RIGHT SHIFT Expressions

Syntax: `EXPRESSION >> EXPRESSION_N`

Where: EXPRESSION is any valid C numeric expression.

EXPRESSION_N is any valid C integer expression.

Example: `x = 0110 >> 1;` Gives 0011 in x.

Bits in the EXPRESSION are shifted to the right by EXPRESSION_N bit positions. Vacated bits are filled with the signed bit if the number is signed or with zeros if the number is unsigned. Bits passing the right boundary of the number are discarded.

EXAMPLE 7: DATA STRUCTURES AND TYPEDEF

```
struct STUD_REC                                                    (1)
{
     char name[30];
     int age;
     double gpa;
} john;

struct STUD_REC studs[10];                                         (2)

int main(void)
{
     int i;
```

```
      printf("Enter info:\n");
      scanf("%s",&john.name);                                    (3)
      scanf("%d",&john.age);
      scanf("%f",&john.gpa);

     printf("Enter info for all students:\n");

     for(i=0; i<10; i++)
     {
       scanf("%s",&(studs[i].name));                             (4)
       scanf("%d",&(studs[i].age));
       scanf("%f",&(studs[i].gpa));
     }
}
```

C has a useful data-type grouping construct called the *structure*, which allows the creation of complex data structures. There are two kinds of structures: the `struct` and the `union`. In either case, they are used to build new data structures called records. A record is a data structure that behaves like a regular variable but is composed of many variables. In the programming example above, a `struct` is used to group variables that contain information about students. In this case, the `struct` actually represents an individual student. Item (1) shows how to define a structure as a variable, and item (2) shows how an array can be created where each cell is a structure. Items (3) and (4) show that the ampersand (&) must still be used with `scanf` when inputting data into a structure. Items (3) and (4) also show how to reference the fields using the dot-operator.

Programmers can define their own type names using the reserved word **typedef**. A simple example of this is:

```
typedef int MONEY;
MONEY x;
```

In the above example, x is of type MONEY, which in reality is data type int. It is common in C to write typedef'd identifiers all in uppercase letters. We can do the same for `struct` and `union`, as in this example:

```
typedef struct STUD_REC
 {
      char name[30];
      int age;
      double gpa;
 } STUDENT;

STUDENT array[10];
```

The array has data type STUDENT. Each cell of the array contains a student record. In this case, the usefulness is not only in making the code easier to read, but it also serves as a shorthand notation for the structure definition.

Syntax Details for Example 7

The structure statement builds a **single record** containing many fields. Each field can store information. The union statement, on the other hand, builds for us a **single variable** that can represent many different types of data but can store only one value of one type at a time.

The Structure Statement

Syntax:

```
struct STRUCT_NAME {
       FIELDS;
 } VAR_LIST;
```

Where:

STRUCT_NAME is an optional type identifier.
FIELDS one or more legal variable, structure or union definitions.
VAR_LIST is a comma-separated list of variable identifiers.

Or

```
struct STRUCT_NAME VAR_LIST;
```

Alternative way to defining variables assuming STRUCT_NAME is already declared somewhere.

Assigning data to a structure: Assigning a value to a field

```
VAR_NAME.FIELD_NAME = EXPR;
```

Where:

VAR_NAME is an identifier to a structure.
FIELD_NAME is the name of one of the fields in the structure.
The DOT (.) operator is used to distinguish the field name from the structure identifier.
EXPR is any legal expression having the same type as FIELD_NAME.

Example:

```
john.gpa = 3.6;
```

Retrieving data from a structure: Accessing the value at a field

```
VAR = VAR_NAME.FIELD_NAME;
```

Where:

VAR_NAME is an identifier to a structure.
FIELD_NAME is the name of one of the fields in the structure.
The DOT (.) operator is used to distinguish the field name from the structure identifier.

Example:

```
int x = john.age;
printf("%s", john.name);
```

The Union Statement

Syntax:

```
union UNION_NAME {
      FIELDS;
} VAR_LIST;
```

Or

```
union UNION_NAME VAR_LIST;
```

Where:

UNION_NAME is an optional type identifier.
FIELDS is one or more legal variables, structure or union definitions.
VAR_LIST is a comma-separated list of variable identifiers.

Example:

```
union grade_rec
{
      float numeric;
      char letter;
} grade;
```

The example above creates a variable called `grade` that can be used to stor e a student grade regardless of how the professor chooses to represent the mark: i.e., as a real number from 0 to 100 or as a letter grade from F to A.

Accessing the contents and assigning values to unions is identical to the structure statement.

Examples: (note: only 'B' is printed out in this example, since only one value is stored.)

```
grade.numeric = 75.8;
grade.letter = 'B';
printf("%c",grade.letter);
```

More complex structures can be built by combining structures and unions.

Example:

```
typedef struct stud_info
{
      char student_type; /*'F' for full-time 'P' for part-time*/
      char name[30];
      int age;
      float gpa;
      union full_or_part
```

```
        {
                struct full_time
                {
                        float discount_rate;
                        union grade_rec grades[90]; /* more classes */
                } full;
                struct part_time
                {
                        float balance;
                        union grade_rec grades[30];
                } part;
        } type;
} student;
student classroom[50];
```

The above structure defines a classroom of 50 students. All students have some common information they all share. Then, depending on the type of students they are, they have additional information. A variable, student_type, is used to remember what kind of student is recorded within the union. Each element of this structure has structure type names—**full-time** and **part-time** and structure variable names- **full** and **part**. The structure variable names are used to reference the data stored in the structure.

Usage examples:

```
scanf("%s",class[2].name);
class[2].student_type = 'F';
class[2].type.full.discount_rate = 0.30;
class[2].type.full.grades[7] = 83.2;
class[2].type.full.grades[8] = 'A';
```

This reads a string from the keyboard and assigns it to the name field in location 2 in the array. Then, it assigns the character 'F' to the field student_type, the value 0.30 to the discount_rate, which is part of the full-time structure: 83.2 to the grades at index 7 and 'A' to the grades at index 8.

EXAMPLE 8: POINTERS & DYNAMIC DATA STRUCTURES

```
struct NODE
{
      int data;
      struct NODE *next;
};
```

```
int main (void) /* add the following integers into a linked list */
{
      struct NODE *head = NULL, *temp;                                    (1)
      int value;

      printf("Enter integer numbers to the list. Enter zero to quit: ");
      scanf("%d",&value);

      while(value != 0)
      {
            temp = (struct NODE *) malloc(sizeof(struct NODE));           (2)
            temp.data = value;

            if (head == NULL)     temp.next = NULL;                       (3)
            else temp.next = head;

            head = temp;

            printf("Next value: ");
            scanf("%d",&value);
      }

      printf("Dump the contents of the linked list:\n");
      for (temp=head; temp!=NULL; temp=temp.next)
            printf("Data = %d\n", temp.data);
}
```

Dynamic data are structures that can be created while the program is running. All the structures and variables we have seen to this point are compile-time structures. This means, for example, when a variable is defined called x and another one called y as `int x,y;` they are created when the program was compiled. When the programming is running, we cannot ask the computer for another x and y. We have only one x and one y to use. These are known as compile-time data structures. A dynamic data structure can grow and shrink as the program runs. An example is the dynamic array that can change its size while the program is running, getting bigger when you need more space and shrinking when you don't need as much space. In C, to do dynamic operations requires you to understand another topic as well: pointers.

The above program demonstrates two capabilities: pointer manipulation and dynamic data structures, which, in this example, is the linked-list data structure. The program operates in the following way: the program asks the user for numbers. The user can enter as many numbers

as the user wan ts. The loop ter minates when the user inputs the n umber zero. Every number entered, except the zero, is placed in to its own `struct NODE`. Then the program links all the nodes together in to a chain. The `head` pointer references the first node. Each node r eferences the next node in the ch ain. The last node does not r eference anything. Its `next` pointer is assigned the value NULL that indicates it is not pointing to anything. The chain is now complete. We can add more nodes anywhere we like, as long as the chain is maintained.

Bellow we highlight some points:

1. Pointers are declared the same way as other variables. To designate a variable as a pointer, we add the asterix. In our example, the variables `head` and `temp` are pointers. `Head` is initialized to the NULL value since it will be used to refer to the concept **linked-list;** but, at this point, it does not exist yet, so the NULL is used to indicate that. `Temp` is not initialized at this time and is used as a temporary pointer referencing the newly created structures before they are attached to the linked-list.
2. The function **malloc** (and its cousin **calloc**) are used to provide additional memory to a program at run-time. This additional memory is known as dynamic memory, and hence, in this example, we are creating a dynamic data structure called a linked-list. The programmer must specify the number of bytes needed for each structure. Then malloc attempts to find this amount of memory. The returned memory is then typecast to the same type as the pointer. In this case, this is the structure called NODE. NODE is a common way of naming the elements of a linked-list.
3. At this point in the program, a decision is made about how the newly created node should be added to this linked-list. There are two choices only. If the list is empty, then just attach the new node to the `head` pointer as is. If there is a list, then insert the newly created node into the list, also at the beginning of it. This is done by making the temporary node point to the first node in the list and then having `head` point to the new temporary node.

Syntax Details for Example 8

Pointer Declaration and Usage

The Pointer

Syntax:

```
MODIFIER TYPE *IDENTIFIER = EXPR;
```

Where:

MODIFIER is any legal type modifier.
TYPE is any legal primitive type (e.g., char) or any user-defined type (i.e., struct or union).

IDENTIFIER is any legal C identifier.
EXPR (and the = operator) is optional. EXPR is any valid expression of the same TYPE.
* (ASTERIX) is a symbol with multiple meanings. In a variable declaration statement, it serves to distinguish the identifier as a pointer, instead of as a normal variable.

A pointer behaves like a variable (it can be used exactly like a regular variable); but, the value stored in this variable is an address of memory (RAM). Regardless of the TYPE, the pointer contains an unsigned 32-bit integer number representing an address of a memory location. That memory location contains data in the TYPE. The address must reference a location in memory that contains information of that TYPE; but, the pointer itself is always an unsigned integer. Pointer sizes change as our computer memories get larger. Today, the 32-bit pointer is changing to 64-bit and 128-bit. Regardless of the size, pointers behave as described.

The pointer variable identifier on its own gives you access to the address number (32-bit unsigned integer number). To get access to the data, the pointer is referring to you need to use the asterix symbol. The example below shows this:

Usage Example:

```
(A)  char *p = "sentence";
(B)  p = p + 1;
(C)  printf("%d, %c, %s", p, *p, p);
(D)  *p = 'b';
(E)  if (*p == 'b')
(F)        *p = *p + 2;
```

Where:

A. Here, **p** is declared as a pointer whose address references a character. The string "sentence" is created and assigned to **p**. This actually causes the *address* of the first character 's' to be stored in **p**. Since a string is a consecutive series of characters terminated by the NULL character, **p** still has access to the subsequent characters after 's' by incrementing the address in it to the next memory location (this is shown in (B)).

B. The address in **p** can be incremented. Now **p** points to the second letter of the string, in other words to 'e'. The incrementing operation is not by 1 even though we said p + 1. The "+1" refers to the desire to go to the next memory location. But, the next memory location depends on the type of data. For example, integer numbers use up 2 bytes. **p = p + 1;** would cause p to be incremented by 2 bytes if **p** were of type int. Since in our example, **p** is of type char and characters are only 1 byte long, therefore, p = p + 1; would increment **p** by 1 byte. Similarly, if **p** were of type double, then **p** would be incremented by 4 bytes.

C. The **printf** function demonstrates all the ways **p** can be output. Respectively, **p** outputs each of the following: the unsigned integer value of the address stored in **p** (%d), the

actual single character in RAM that **p** is pointing to (%c), and the complete string until the NULL character (i.e., a'\0') is encountered (%s) (hence, all of the string is displayed). **Note!** If, for any reason, the NULL character is not present at the end of the string, then the printf will continue printing past the end of the string into memory until a NULL character is found or until it goes out of legal memory space. This is also true when the string is stored in an array. This can produce very strange output if you are not careful.

D. ***p** is used to refer to the "contents of p." In other words, ***p** refers to the location in RAM where **p** is pointing. In this case, that location will be assigned the letter 'b.' The effect is that the second character in 'sentence' is changed to 'b.' The string 'sentence' now says 'sbntence.'

E. Here, a test is made to see if *p points to the character 'b.' In this case, it does, because of (D).

F. Unlike (B), where the address stored in the variable p is incremented, this statement will increment the data stored at the location p. This takes advantage of the fact that characters in C are stored as integer values (that represent ASCII codes). Here, the value is being increased by 2. The effect is to change the character 'b' into 'd,' which is two characters away from 'b.'

There is an addition al symbol associa ted with pointers. This is the ampersand symbol (&). Besides being able to stor e the address of a memory location in a poin ter variable and using the asterix to reference the data stored at that address is memory, sometimes it is impor tant to find out wh at the addr ess of a par ticular variable is. Below we h ave a summ ary of these operations:

Syntax	Description	Example
TYPE *ID;	The declaration of ID to be a pointer.	int *p;
ID = &VAR;	The pointer ID is given the address of the variable VAR.	char x; int *p; p = &x;

VAR = *ID;	The variable VAR is given the value stored at the address pointed to by ID.	char x = 'a', y;
		int *p;
		p = &x;
		y = *p; /* y has 'a' */

Dynamic Programming

A dynamic data structure is a block of con tiguous memory automatically created by the oper-ating system at run-time when your program requests it. This block of memory can be a string, a variable, a structure, a union, or an undesignated block of memory. Arrays and structures are

examples of compile-time data structures. They take up space in memory and have identifiers that refer to them. A dynamic structure is built at run-time and therefore does not have an identifier since the programmer was not aware of it when the program was compiled. Instead, we build the data structure using one of two special commands, called malloc and calloc, that only return an address to the first byte of the newly created block. We can then store this address in a pointer. The pointer is the surrogate for the identifier.

Dynamic memory can be attached and referenced by the program through pointers. We can also construct complex chains of pointers, like the linked-list example we have seen. The advantage is that we need only one pointer referencing the beginning of the list. Each list node is responsible to know where the next node is. This allows lists of infinite length.

Dynamic Structures

Syntax:

```
PTR = (TYPE *) malloc(BYTES);
PTR = (TYPE *) calloc(QTY, BYTES);
BYTES = sizeof(TYPE);
```

Where:

PTR is the identifier for the pointer. It must be the same type as the structure being built.
TYPE is the type name of the structure (i.e., int, char, struct stuff_info).
(TYPE *) performs a type cast of the structure being built. It must agree with PTR.
BYTES is the total number of bytes that will be built.
QTY is a multiplication factor. Therefore, the total number of bytes built is QTY * BYTES.
sizeof(TYPE) counts the number of bytes in TYPE and returns that as an integer number.

Example: (These four examples are equivalent.)

```
char *s = (char *) calloc(10, sizeof(char));
char *t = (char *) malloc(10 * sizeof(char));
char *u = (char *) malloc(10);
char *v = (char *) calloc(10, 1);
```

There are many kinds of dynamic data structures. It goes beyond the scope of this text to cover them all. We have looked at the most common ones. These are: dynamic strings, dynamic structures, and linked-lists. Additional dynamic memory constructions are: buffers, dynamic arrays, stacks, doubly linked-lists, trees, heaps, and graphs.

EXAMPLE 9: PRE-PROCESSOR DIRECTIVES

```
#include <stdio.h>                                          (1)
#include <stdlib.h>
```

```
#define FRENCH                                                    (2)

void main(void)
{
      int score[10];
      int name[10];

      #ifdef FRENCH                                               (3)
            printf("Votre Nom: ");
      #else
            printf("Your Name: ");
      #endif

      scanf("%s", name);
}
```

C is composed of two languages: the C Language and the C Pre-Processor Directives. The Pre-Processor Directives can be identified by the sharp (#) symbol that precedes the commands and because they do not end with a semi-colon. The Pre-Processor is used to control how the source file will be compiled and how it will be presented to the compiler for compilation. These directives can affect the actual contents of the source code before compilation.

In the example above, the program can be compiled into either an English or a French executable, without the need of keeping two different source files. This is a very nice feature. The programmer simply needs to comment out **#define FRENCH** and the program will compile with all the English printf statements.

1. **#include** comes in two forms: **#include <FILE>** and **#include "FILE".** This directive inserts the specified text file into your source file at the exact position the #include appears. The other text file could be anything (C code or not). This insertion occurs before your program is compiled. If you use the angle brackets, then the compiler will use the default library directory to find FILE. If you use the double quotes, then the compiler expects you to provide a path. If no path is provided, then the compiler uses your current working directory.
2. **#define** comes in three forms: as a word definition (as in this case), as a text replace, or as a macro. In this example, we are defining the word FRENCH. If the line were commented (which is not the case here), then the word FRENCH would not be defined. Since the word is defined, the compiler keeps it in its memory until the compilation process is done. This example directive is used in (3).
3. **#ifdef** is like a regular if-statement, except that it tests to see if a word has been defined using a **#define** directive. If it was defined, then the first part of the **ifdef** is compiled. If it was not defined, then the **else** part of the **ifdef** is compiled.

Syntax Details for Example 9

Pre-Processor Directives

Syntax	Description	Example

`#define TEXT VALUE`	TEXT is a word that will be substituted by the word VALUE before compilation.	`#define TRUE 1`
		`while (TRUE) …`
		
		`The above is substituted as:`
		
		`while(1)`
		
		`And given to the compiler.`

`#define MACRO EXPR`	MACRO looks like a function name with arguments. The MACRO is replaced by EXPR before compilation with the arguments replacing the variables in EXPR.	`#define max(a,b) (a<b)?b:a`
		`int x;`
		`x = max(2,3);`
		
		`The above is substituted as:`
		
		`int x;`
		`x = 3;`
		
		`And given to the compiler.`

`#include <FILE>`	FILE is the name of a text file on disk. The entire contents of that file will be inserted at the spot where the **#include** directive has been placed. The angled brackets, **<** and **>**, indicate that the C default include directory will be accessed.	`#include <stdio.h>`
`#include "\PATH\ FILE"`	This is identical to the previous **#include** except that the double quotation marks indicate that the user must supply both the PATH and FILE name to the text file. The direction of the slash depends on the UNIX or WINDOWS environment you are using.	`#include "\usr\ stuff.txt"`

(*Continued*)

Syntax	Description	Example

`#ifdef TEXT`
`  CODE1`
`#else`
`  CODE2`
`#endif`
	TEXT is a word that has been #defined previously. If TEXT is #defined then CODE1 is compiled else CODE2 is compiled.	`#define stuff`
		`#ifdef stuff`
		`  printf("hello");`
		`#else`
		`  printf("bye");`
		`#endif`

`#ifndef TEXT`
`  CODE1`
`#else`
`  CODE2`
`#endif`
	Like #ifdef but tests if TEXT was NOT defined previously.	`#ifndef stuff`
		`  printf("hello");`
		`#else`
		`  printf("bye");`
		`#endif`

`#undef TEXT`	Makes TEXT not #defined	`#undef stuff`

`#if EXPR`
`  CODE1`
`#else`
`  CODE2`
`#endif`
	Like #ifdef except that EXPR is an integer calculation. If it evaluates to zero, then CODE2 is compiled; otherwise, CODE1 is compiled.	`#define stuff 5`
		`#if stuff -2`
		`  printf("OKAY");`
		`#else`
		`  printf("CLOSED");`
		`#endif`

`#line CONSTANT TEXT`	Makes the compiler think that the source code starts at line number CONSTANT and that the source file name is called TEXT.	`#line 30 "newfile.c"`
`#pragma TEXT`	Provided so that implementations may support implementation-specific directives.	`Not supported by many C compilers.`

`#error MESSAGE`	Causes MESSAGE to be displayed within the compiler error messages.	`#ifndef A`
		`#error A is undefined`
		`#endif`

EXAMPLE 10: FUNCTIONS, RECURSION AND SCOPE

```
#include <string.h>                                              (1)
#define TRUE 1
#define FALSE 0

char theWord[50], newWord[50];                                   (2)

int IsPalindrome (char word[])                                   (3)
{
      char newWord[50];
```

```
        if (strlen(word) == 0 || strlen(word)==1) return TRUE;          (4)
        if (word[0] != word[strlen(word)-1]) return FALSE;

        strncpy(newWord,word+1,strlen(word)-2);                        (5)
        return IsPalindrome(newWord);                                  (6)
}

main()
{
        printf("input a word to test: ");
        scanf("%s", theWord);
        if (IsPalindrome(theWord))                                     (7)
                printf("YES it is a palindrome\n");
        else
                printf("NO it is not a palindrome\n");
}
```

In the introductory section of this chapter, there was a discussion about the **main** function, its syntax, and use. All function syntax is patterned after the operation of the **main** function. In other words, a function in C has a function name, input parameters, and a return value. The return value and input parameters are optional. The contents of a function are placed within its own **scope**.

A scope defines a self-contained area of your program; another term for this is a **local** area or local scope. Variables, statements, and commands affect the contents of the scope in which they are defined. Scopes can be overlapping. If they do overlap, then the most immediate scope is consulted first for the value of the identifier; after that, the next immediate scope is consulted, and so on, until there are no more scopes left. If we run out of scopes and the identifier is not found, then an error is generated. Generally speaking, C has the following scopes: **block**, **local**, **relative-file-position**, **external**, and **static**. Let us first look at some definitions and discuss the example program.

Function Structure

Syntax:

```
RTYPE FNAME (PARAMS)
{
        LOCAL;
        STATEMENTS;
        RSTATEMENT;
}
```

Where:

```
RTYPE function return type
FNAME function name
PARAMS a comma separated
LOCAL immediate scope
STATEMENTS are legal C statements
RSTATEMENT optional return statement:
                                    return VALUE;
```

In our above example, item (3), `IsPalindrome`, is a function. It returns an integer value. It is specifically invoked (or called) by the name `IsPalindrome` with the 'I' and 'P' in caps. It has one parameter, an array called `word` that is of type `char` and can be any length (i.e., [ ]). The function has an additional local variable called `newWord`. It is also an array of type `char` and can store up to 50 characters. Any reference to `newWord` within the function will access this occurrence of the variable. Notice that in (2), there is an additional declaration of `newWord`. This variable is in an outer scope and will never be accessed by `IsPalindrome`. We will say more about scope soon.

This function is special because it invokes itself. Because of this ability, it is called a **recursive** function. Notice how the function operates. It takes in a word (3). It then tests for a few cases that either rule out or conclude that the word is a palindrome (4). Built-in C string manipulation functions are used. The library function `strlen` returns the number of characters in the array. If there is 1 or 0 characters in the array, then the word, by definition, is a palindrome. A palindrome is a word that can be read from left to right and right to left and spells the same word; for example: BOB and ABBA. If the first letter of the word does not match the last letter of the word, then, by definition, it is NOT a palindrome. Then (5) and (6) use the string command `strncpy` to copy the array `word` into the array `newWord` without the first (i.e., **word+1**) and last (i.e., **strlen(word)-1**) letters. We do this because we have already tested these letters in (4). We continue to do this operation on the word until there are no more letters in the array or until there is only 1 letter in the array; then, by definition, it must be a palindrome.

Statements in (1) show how to access the string functions using the `#include` directive. The `#define` directive is also used to make the program more readable by assigning the word **TRUE** to the integer number 1 and **FALSE** to the number 0.

The `main` function (7) calls the `IsPalindrome` function using an if-statement. When a 1 is returned, the function displays YES; otherwise, it displays NO.

The example program uses only two kinds of scope: local and **e**lative-file-position. Local scope can be seen in statement (3) and the line below (3). There the variables `word` and `newWord` are declared. They both have immediate scope. This means that they can *only* be referenced by the `IsPalindrome` function. No other part of the program can reference the contents of these variables. There is a special case, though. These variables can be referenced when a pointer is assigned to their address. This is not the case in this example.

In statement (2), we see two variables defined with relative-file-position scope. Relative-file-position variables can be defined anywhere in a file as long as they are not within a function. This means that these variables can be defined near the top of the file (as in this example), or lower down between functions, or even, in the extreme case, at the bottom of the file after the `main` function. Relative-file-position variables can be accessed in all parts of the code (inside or outside functions) that come after the variable declaration, but they cannot be accessed

anywhere before the declaration. Local scope takes precedence over relative-file-position scope when they overlap.

Three additional scope types exist that were not presented in the example: block, external, and static. Block scope variables are defined within the open and close curly brackets, i.e., { and }, of a statement or within the statement itself, as in this example:

```
for(int I=0; I<10;I++) { printf("%d",I); }
printf("%d",I);
```

The above is a classic example of block scope. The 'I' variable is declared within the for-loop. This scope is the most immediate and therefore takes precedence over all other overlapping scopes. Once we exit a scope, the variables are no longer accessible. Therefore, the first print statement works. It will print all the numbers from 0 to 9 that 'I' is assigned. BUT, the second printf is outside of the for-loop and therefore would generate an error message for trying to print the value in a nonexistent variable 'I.' **IMPORTANT**: Local and Block scoped variables will actually lose the values stored in them when you exit their scope. The loss of value is because the program will actually build new variables each time it enters a Block or Local scope. This also means that the program deletes the variables when it exits their scope. As a result, the second printf code (in the for-loop example above) will generate an error because there is no 'I' variable at this point in the code!

External scoped variables provide for a primitive form of **object-oriented programming**. The GNU compiler GCC can compile source files separately, each being stored in its own .o file. The .o files are then linked into a single executable file. Relative-file-position variables compiled in this manner are treated as **private** variables. This means that other source files compiled in the same manner cannot access these relative-file-position variables, even if they are linked later (or after, or below the .o file). Normally, relative-file-position variables are accessible by all the code that appears after their declaration within the same source file. Now, if we take a relative-file-position variable declaration and prefix it with the **`extern`** modifier, then its scope changes to external. An external scoped variable can be accessed **publicly**. This means that any .o file can reference it. If you are familiar with object-oriented programming, you should not confuse `extern` with `public`. The statement `extern` should be viewed as global (i.e., it can be referenced in all classes without instantiating the object. There is an additional scope modifier known as `static`. The statement `static` is also a variable modifier. If used at a variable declaration, it causes the variable scope to change to `static`. This means that the variable is only created once and only destroyed when the program is terminated. This changes the behavior of Local and Block scoped variables considerably. Local and Block scoped variables still retain their characteristic of being accessible only from within their scope boundary; however, with this modifier, the variable is not destroyed or re-created when we exit or reenter the scope. The effect is that the variable still retains its last value when the scope is re-entered.

SOME USEFUL STANDARD C LIBRARIES

#include <string.h>
char *strcat(char *DEST,char *SOURCE); - concatinates source at end of destination, returns pointer to destination
char *strncat(char *DEST,char *SOURCE,int N); - same as strcat but concatinates first n characters of source
int strcmp(const char *STR1,const char *STR2); - returns 0 if same, >0 if str1>str2, <0 if str2>str1
int strncmp(const char *STR1,const char *STR2,int N); - same as strcmp but compares first n characters
int strlen(char *STR); - counts the number of characters in the string not including the NULL
char *strcpy(char *DEST,char *SOURCE); - overwrites destination with the contents of source
char *strncpy(char *DEST,char *SOURCE,int N); - overwrites destination with the first n characters of source

#include <match.h>

double cos(double X);	-returns the cosine of X (also sin and tan with similar syntax)
double acos(double X);	- returns the arc-cosine of X (also asin, atan with similar syntax)
double exp(double X);	- returns e to the power X
double fabs(double X);	- returns the floating point absolute value of X
double floor(double X);	- returns the floor of X (i.e. if X = 5.6 this returns 5.0)
double ceil(double X);	- returns the ceiling of X (i.e. if X = 5.1 this returns 6.0)
double fmod(double X, double Y);	- returns the floating point modulo of X % Y
double log(double X);	- returns log base 2 of X
double log10(double X);	- returns log base 10 of X
double pow(double X, double y);	- returns X to the power of Y
double sqrt(double X);	- returns the square root of X

#include <stdio.h>

FILE *fopen(const char *NAME, const char *OPTIONS);	-returns a pointer to an open file
int fclose(FILE *STREAM);	-closes the file pointed to by stream
int feof(FILE *STREAM);	-returns true if pointer at end of file
int fgetc(FILE *STREAM);	-return a single ASCII value from file at pointer

int fputc(int C, FILE *STREAM);	-write a single ASCII value to file at pointer
char *fgets(char *DEST, int LIMIT, FILE *STREAM);	-returns a line of text
int fprintf(FILE *STREAM, const char *FORMAT, ARGLIST);	-like printf but for files
int fscanf(FILE *STREAM, const char *FORMAT, ARGLIST);	-like scanf but for files
int scanf(const char *FORMAT, &ARGLIST);	
int printf(const char *FORMAT, ARGLIST);	
int sscanf(const char *SOURCE, const char *FORMAT, ARGLIST);	
int sprintf(const char *DEST, const char *FORMAT, ARGLIST);	

#include <stdlib.h>	
int abs(int N);	- returns the integer absolute value if N
double atof(const char *STRF);	- converts a string to floating point
int atoi(const char *STRI);	- converts a string to integer returns 0 if invalid
int rand(void);	- returns a random value between 0 and MAXINT
void srand(unsigned SEED);	- randomizes the random number generator
int system(const char *COMMAND);	- invokes the OS shell to execute an OS command from within C
long labs(long L);	- returns the long integer absolute value of L
void abort(void);	- prematurely terminates the program

#include <ctype.h>		
char toupper(char)	int isdigit(char)	int isupper(char)
char tolower(char)	int isalnum(char)	int islower(char)
int isalpha(char)	int isspace(char)	

Other Standard C Libraries

#include <time.h> - time and date functions	#include <signal.h> - interrupts
#include <errno.h> - run-time error messages	#include <float.h> - floating point definitions
#include <limits.h> - containing numerical limits	#include <assert.h> - basic debug feature
#include <stdarg.h> - variable parameter support	#include <stddef.h> - type definition macros

PROBLEMS

1. Write a program that asks the user for an integer number. Call this number N. The program will then draw a box using Xs. The box will be N Xs wide and N Xs long.
2. Write a program that asks the user for a floating-point number. This number will be the user's grade. The program will then convert the grade into a letter grade and display the letter grade on the screen. A grade 85.0 or above is an A. 75 to 85 (85 not included) is a B. 65 to 75 (75 not included) is a C. 55 to 65 (65 not included) is a D. Below 65 is an F.
3. Write a program that asks the user for 10 integer numbers and then displays the average of these numbers.
4. Write a program that continually asks the user for positive integer numbers. Once the user enters a non-positive number, the program displays the average of all the positive integer numbers.
5. Write a program similar to the one in question 4, but that gets its numbers from a text file called grades.txt.
6. Write a program that displays a bar graph on the screen of the frequency distribution of integer values. The values are read in from a text file called values.txt. Values greater than 500 are in column 1, 400 to 500 in column 2, 300 to 399 in column 3, and so on.
7. The program loads an integer array of size 20 with unsorted values from a text file. The program then asks the user for an integer number and checks to see if the number is in the array. If it is in the array, the program displays the message FOUND; otherwise, it says NOT FOUND. Then the user is prompted for another value. This process is repeated five times before the program terminates.
8. Write a program like the one in problem 7, but this time, the input file has numbers sorted in increasing order. Use, this time, a binary search algorithm to find the values.
9. The program loads an integer array of size 20 with unsorted values from a text file. The program then uses selection sort to order the values in the array in increasing order.

CHAPTER FOUR

Understanding Systems Programming

This chapter and chapter 5 will be your introduction to systems programming. Chapters 1 through 3 built up your background knowledge. Now we will be able to use what was learned in chapters 1 through 3 and apply it to real systems programming implementations. Specifically, this chapter will explore interfacing your program with computer systems that are one layer away from your program. This layer includes the computer's operating system and the software development environment. Chapter 5 will take you to the next layer, the Internet.

Like a Matryoshka doll, where one doll is within another repeatedly until the last tiny doll, a software system has this property as well. Your program is one of those dolls When the dolls are within each other, a particular doll can only touch the bigger doll it is immediately within; the same is true for programs. The systems that interact most intimately with your program are the systems that are closest to your program, and these are the computer's operating system and the software development environment (used to construct your program). Chapter 4 explores these two systems. Chapter 5 will take us to the layer just beyond, the Internet. Chapter 4 is preparatory for chapter 5.

The computer's operating system provides important services to your program. These services include some of the following: application interfaces to **peripherals** like the hard disk, random access memory, the screen, the keyboard, the mouse, and the printer. The operating system also allows for **inter-process communication**. This term refers to programs that are currently running, known as **processes**, that can be aware of each other and communicate together synchronously and/or asynchronously. Another important feature the operating system provides is an application interface that allows programs to launch and terminate processes. We will explore many of these features in this chapter.

But, we will begin our exploration with the software development environment. Chapter 3 ended with the basics of programming in the C language. We will continue here with developing complex C applications using the software development environment. In our case, we are assuming Unix and the GNU Tool set; but, these principles are commonly implemented in all development environments, including Microsoft's and Apple's environments. Unlike the Matryoshka dolls, programs are sandwiched between two immediate layers, one from the backside and the other from the front. The backside is the software development environment; the front side is the operating system. We will begin with the backside.

This chapter will explore the software development environment by first looking at an advanced C engineering technique called **modular programming**. Then we will look at the GNU Tool set and how it is helpful to programmers constructing industry-level applications. After this, we will look at the operating system and how your program can interface with that.

MODULAR PROGRAMMING IN C

There are three ways to construct a C program. These ways are commonly known as: single-source file programming, pseudo multiple-source file programming, and modular programming. Modular programming is true multiple-source file programming.

Single-source file programming is the method of choice for small to medium applications. The rational is that the program is small enough to exist entirely within a single .c file. Taking this idea to the extreme, we could say that all programs, no matter how large, could be placed within a single .c file. This is true, but it becomes unpractical since larger projects normally have more than one programmer. It becomes hard to share a single .c file between multiple programmers who would want to edit it at the same time. Chapter 3 assumed a single-source file technique. To enumerate this technique specifically, a programmer would perform the following activities:

1. Create and edit their single .c source file using any text editor.
2. Compile and debug syntax errors using a compiler.
3. Perform run-time testing to correct run-time and logic error using the command line.
4. Deliverthe final application program to the customer.

Assuming you used a random text editor to create the .c file, we would use the GNU compiler to build your program. You would compile your single-course file program in the following way:

Syntax:

```
$ gcc SOURCE.c
$ gcc -o PROGRAM SOURCE.c
```

Where:

- The $ represents the command-line prompt. It is assumed you are compiling from there.
- SOURCE is the name of the single source file you would like to compile.
- -o PROGRAM is the name of the executable program.

The above syntax shows two possible ways to compile. The first way uses the compiler's defaults. In this first case, the programmer must supply the name of the single-source file to be compiled. If there are errors, then error messages are displayed and nothing else happens. If no errors exist, then gcc builds the executable program and saves it to the default executable file name `a.out`. In the second syntax example, the same procedure is followed, but the programmer gives his or her own name to the executable file. The default executable name is replaced by the PROGRAM name. The -o switch (dash with lowercase letter o) represents the output file gcc creates, which is the executable file.

Pseudo multiple-source file programming takes advantage of the pre-processor directive #include. We saw in chapter 3 that this directive has two forms: #include <file> and #include "file". The angle bracket version accessed the compiler's libraries. The double quote version allows the programmer to merge a text file into the source file at the exact location where the directive was placed. This would replace the directive by the merged text file. The merged text file could contain any information, without restrictions. If this text file was a source file, then you could merge source files written by other people together. This is how it is actually used:

1. A unique .c file is created that contains the program's main() function.
2. All other programmers create their code within their own personal .c files.
3. All the personal .c files are #included into the main() function .c file.
4. We compile only the main() function .c file, since this would include all the others.
5. From the compiler's point of view, it receives a single merged source file and compiles it.
6. A single executable program emerges.

There are disadvantages to the pseudo multiple-source file-programming technique. Its major disadvantage is that the entire program is compiled each time you ask the program to compile. Even when all the source files are unchanged except for one source file, the entire **project** is compiled. In small to medium projects, this is not a problem. In medium to large projects, this could be long enough to walk down to the cafeteria and buy some coffee. If you need to compile multiple times in a day, or multiple times in an hour, this can become a problem. The second disadvantage is that everything becomes global. In chapter 3, we talked about the scoping rules of C. These rules provided for a concept called positional scope. This meant that a variable declared outside a function is considered to be positionally global. In other words, it would be global to those things declared below it in the file. Those things that come before it

would not be able to see it. Since we are using the #include directive to merge these other files into the main() function file, we can only insert these directives under the positional scope rules. Therefore, all variables and functions become positionally global. There is no concept of private variables or private functions. Private code is a useful idea. Object-oriented programming is based on this idea. Modular programming overcomes these two disadvantages.

Assuming you used a random text editor to create all the .c files in the project, the GNU compiler can be used to build the program. What is interesting is that you would build the application in the same way described for the single-source file version, for example:

```
$ edit main.c
$ edit source1.c
$ edit source2.c
$ gcc -o app main.c
```

Notice in the above example that three source files were created: main.c, source1.c, and source2.c. It is assumed that source1.c and source2.c are included in main.c. Since this is the case, only the main.c source file is compiled since it already includes the other two source files. The above example creates an executable named app.

The main.c file might look like this:

```
#include <stdlib.h>
#include <stdoi.h>
#include <string.h>

int globalVarX;

#include "source1.c"
#include "source2.c"

int main(void)
{
      :
      :
}
```

Notice where the source1.c and source2.c files have been placed. When compiled, these two source files will be inserted into the main.c file at the positionally global location immediately above the main function. Positionally global to everything will be the variable globalVarX. Since everything is global, in this technique, globalVarX can be used by source1.c and source2.c as if it was defined in those files (even though it has not been).

Modular programming: C uses two physical elements to facilitate modular programming: the #include directive and the `extern` statement. The #include directive works on source files, while the `extern` statement works on C language elements. Function prototypes are

external by default, but variables and structures are not and need the `extern` statement. The last element is the C language compiler. The C compiler builds the program according to the pseudo or modular ways of making programs. Understanding how the compiler does this will further help you exploit modular programming.

The C language compiler has a mode that permits a source file to be compiled in to what is known as an **object file**. This is not to be confused with object-oriented programming's objects. An object file is a compiled but unlinked program without a main function. A true program needs a main function defining where the application starts from. A true program needs to go through the operation of linking source code library references with the actual library programs. Object files do not have these two elements. But, there are two important features in object files. First, they can be linked in to any project at a later date. This ability makes object files portable between projects. Second, everything in an object file is within its own **named space**. A named space is a contiguous amount of memory that is private. Anything created in that named space, like variables and structures, can only be accessed by code that is also in the scope of that name space. This implements a rudimentary form of variable and function privacy.

The `extern` statement provides a way by which the programmer can selectively choose items in a named space and make them public, or more correctly, positionally global. So, if we think back to the pseudo multiple-source file programming method, each programmer has his or her own personal source file where code is written. In the pseudo method, the programmer knows that everything written in that file will be positionally global when included in the project. Using the modular programming method, the programmer knows that his or her personal source file is a named space and not position ally global after linking into the project. This allows the programmer to behave in a fashion that is more akin to object-oriented programmers. The project is now divided into what is known as modules. Each module has a programmer who is responsible for it. The programmer maintains an object-oriented way of dividing his or her module in to private data and function and positionally global data and functions. This permits information hiding and permits the use of private variables that have the same variable identifier as a variable in another named space but without scope errors.

To develop software in this mode, the following needs to be done:

```
$ edit main.c
$ edit source1.c
$ edit source2.c
$ gcc -c source1.c
$ gcc -c source2.c
$ gcc -c main.c
$ gcc -o app main.o source1.o source2.o
```

In the above example, we see the creation of three source files: main.c, source1.c, and source2.c. The GNU compiler gcc is invoked with its object file switch -c. The -c switch asks the compiler to only compile the source file. This means that it will avoid performing the link and main function link steps. Basically, what happens is the following: (1) check for syntax errors, (2) convert the source code to machine language, (3) create a linker table listing all unresolved linking issues with the source file, (4) save all of this in a file having the same name as the source file but with the .o extension. In Microsoft, the .obj extension is used. The last gcc statement in the above example shows how we can invoke the compiler with the .o files. Notice that its syntax is not much different from what we have seen before. The only thing different is the listing of all the .o files. Since there are no .c files in the list, the compiler does not need to compile anything; so the compilation step is skipped, and it proceeds directly to linking the .o files into the app program. For this to succeed, all the unresolved link references need to be connected to programs. These connections can be between the .o file reference and the library, and it can be between the .o file reference and another .o file function or variable, as long as `extern` was used (prototypes for functions).

Maybe you have noticed that object files are linked similar to how libraries are linked. This is not a coincidence. Libraries are actually C object files containing functions. When a library is linked into your program, the operation is identical to linking a .o file into your program and then connecting the link reference to one of the functions in the .o file.

Look at the following source code:

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

extern int src1Var;
int src1function(int);
char src2function(void);

int main(void)
{
      :
      :
}
```

The above code example shows how to connect the main.c file with the source1.c and source2.c files. It looks very different from the pseudo method. Notice that this time, we do not #include the source files. This is because we want them to stay as different named spaces. What we do need is a way to get access to the positionally global items (variables, structures, and functions) that exist in the two source files. We need to first discuss how to specify something as positionally global, or in other words, we need to specify which elements of the source file are not going to be part of the named space.

C gives us two ways to designate elements of source as outside a named space. By default, when creating an object file, all positionally global variables and structures are private within a named space. By default, all functions are not part of the named space; they maintain their positional global nature, but since we do not #include the source file into the main, the main does not know about these function names. In other words, the main program has access to them, but the -c switch does not know about them because the linking stage is skipped. We solve these two problems using the `extern` statement and **function prototypes**. Those variables and structures in the source file that you do not want to be part of the named space must be declared with the `extern` statement:

Syntax:

```
extern TYPE IDENTIFIER;
```

Where:

- `extern` is the reserved word indicating that the element it is attached to is external to the named space; in other words, it is positionally global.
- TYPE is any standard C type, structure, or union statement.
- IDENTIFIER is any legal C identifier name.
- You can initialize the identifier as normal. You can list identifiers in a comma-separated list of identifiers, with or without initialization, as normal.

What is important in the above description is that these external variables are not defined in the source1.c and source2.c files but in the main.c file. From the main.c file as well, we request access to the functions of the source1.c and source2.c files using function prototypes:

Syntax:

```
TYPE IDENTIFIER(PARAMETER_LIST);
```

Where:

- IDENTIFIER is the name of the function in the source1.c or source2.c file.
- PARAMETER_LIST is a copy of the parameters of the function.
- TYPE is a copy of the return type of the function.

What is important to notice here is that we do not specify from which source file the function comes. This is left for the linker to figure out. This poses a problem for the linker if two source files declare a function with identical signatures. An error message is displayed in that case. This confusion exists also for the `extern` statement. Therefore, as a rule, variables and function outside of the named space must have **unique identifiers**. In a team development environment, this can only occur from a team meeting.

Notice further that the two linking source files, source1.c and source2.c, did not need to do anything special. All the work was done in the source file wanting access to the variables and/or functions. This leads to an obvious question: how does the programmer who wants access to the positionally global variables/functions outside of the named space know the identifier names and signatures? Remember that in a large project, these are probably different programmers, each responsible for his or her source file. This is where **header files** come in. The owner of the source file who is sharing certain variables and functions with other programmers creates a special file called the header file and writes the external and signature declarations he wants to make public. Users who want to access the source file will #include this header file. The header file's name, by convention, is the same name as the .c file name, but it uses the file extension .h. For example:

Source1.c

```
const int MAX = 1000;
const int MIN = 0;

int factorial(int n)
{
      :
      :
}

int inRange(int n)
{
      :
      :
}
```

Source1.h

```
extern const int MIN;
int factorial(int n);
```

Notice in the above example that the programmer who is responsible for this module created two files: source1.c and source1.h. Using standard conventions, the .c and .h files share the same name to help with identification. The source1.c file has four members: MAX, MIN, factorial, and inRange. The programmer only wants MIN and factorial to be public. The programmer does not have control over what the other programmers actually do since the other programmers do not need to use the .h file, as in our first example using modules. But, if the team of programmers have agreed to work together, then they will access only what is provided in the .h file. Notice that the .h file contains two declarations: one for the MIN and one for the factorial.

The other programmer who wants to use source1.c will have to do the following:

Source2.c

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

#include "source1.h"

int src2function(void)
{
      :
      :
}
```

Notice that in the source2.c example, the programmer only included the source1.h file. The linker will do the rest. The source2.c programmer can now use the shared variables and functions without any further special syntax constraints. Remember that the #include is replaced by the contents of the source1.h file, which is identical to what the programmer would have done if she knew the identifier names and function signatures of the statements she wanted to access.

GNU TOOLS

There are five basic GNU tools: gcc, make, svn, gprof, and gdb. We will look at each one of these in order. The gcc tool you already know about. It is the C and C++ compiler The make tool helps manage large projects. The svn tools helps a team of programmers share, distribute, comment, and track bugs in a project. The gprof tool helps programmers discover where the program runs slow. The gdb tool is a special line-by-line debugging tool. It helps the programmer overcome run-time and logic errors.

The gcc Compiler Tool

The GNU C/C++ compiler is called gcc. It encapsulates all the steps in the C compilation process: pre-processing, compiling, and linking. It is a command-line compiler. It does not have a graphical user interface, but it can be connected to graphical user interfaces like Eclipse (also an open source free download from the Internet). It does not, by itself, manage projects. Source file projects will be looked at when we talk about GNU's companion program **make**.

The gcc command-line syntax is as follows:

```
$ gcc -OPTIONS -oEXECUTABLE SOURCELIST
```

Where:

- $ is the Unix command-line prompt.
- gcc is the compiler program, and it performs all the steps in the compilation process: first, pre-processing, second compiling, and last linking. It assumes that the libraries have already been installed.
- -OPTIONS is a set of optional switches that can be turned on and off. Each switch begins with a dash followed immediately by a letter that identifies the switch. Then an argument is provided (if needed).
- -oEXECUTABLE defines the name of the executable program (-oNAME). This file is only created when there were no errors during the compiling process. If not present, then the default executable name is used, a.out.
- SOURCELIST is a space-separated list of file names. These file names must have .c, .o and/or .s file extensions. Each file in the list will be compiled individually and then linked together. The final linked program will be stored using the EXECUTABLE file name.

The gcc command under its most simple operation (without using the -OPTIONS) will automatically generate and store, on your hard disk, the .o version of your .c file and the linked executable file (some versions of the compiler do not save the .o files unless the -c switch is present). These files are automatically generated if there are no errors during compiling or linking. If an error occurs, then the .o file, where the error occurred, will not be created, and hence, the executable will never be formed as well. An example of this most simple form is:

```
$ gcc source.c
$ a.out
```

In the above example, the file source.c is compiled and the executable a.out is created. If the -oExecutableName option is not used, then the default executable file name is a.out. This is what the compiler will name your linked program if you do not supply a name for it. At the command line, you can simply write a.out to see your program run.

```
$ gcc -omyprog source.c
$ myprog
```

In the above example, the compiler converted the source.c file into the myprog executable file. The next command-line prompt shows the invoking of myprog.

The gcc compiler has many switches; they are summarized here:

1. The pre-processor changes your original source file based on the directives issued. This creates an intermediate file that is not normally saved to disk. The output is a .i file. If you want to see it, then you need to enter the following:

   ```
   $ gcc -E main.c
   ```

2. Before the compiler converts the source file into an object file, it is temporarily transformed into an assembler file. The output is a .s file (in some versions, an .a file) To see the assembler file, you need to do the following:

   ```
   $ gcc -S main.c
   ```

3. The compiler converts your source file into an object file before it is sent to the linker for the merge and connection to library step. To keep the .o file, you need to issue the following:

   ```
   $ gcc -c main.c
   ```

4. To enable verbose mode and have the compiler output compiling statistics as it compiles, use the `-v` switch.
5. To suppress warning messages while compiling (bad idea, no warning messages will be shown), use the `-w` switch.
6. To see extra warning messages while compiling (probably a good idea), use the `-W` switch.
7. To see all the warning messages while compiling (best, but tedious, idea), use the `-Wall` switch.
8. To optimise code in both size and speed, use the `-O1` switch.
9. To optimize even further while compiling (but does not always work), use the `-O2` switch.

Here are some examples of how we can use gcc:

EXAMPLE 1

```
$ edit file.c
$ gcc file.c
0 errors
$ a.out
```

The above example shows a user creating a program called file.c using a text editor. Then file.c is compiled using the simplest compile format (without any switches). This permits the compiler to run on all its defaults. In this case, this means that file.c will be compiled displaying all warnings and error messages, and the output file will be a.out, assuming no errors occurred. In the above example, the user executes the program since there were no errors. If there were only warnings during compilation, the a.out program is still generated.

EXAMPLE 2

```
$ edit file1.c
$ edit file2.c
$ gcc file1.c file2.c
```

```
0 errors
$ ls
file1.c file2.c file1.o file2.o a.out
$ a.out
```

In this example, the programmer creates two source files and compiles them with gcc's default options. Since there are zero errors, gcc creates an executable file called a.out. To see what was created, the programmer does a dir ectory listing and sees th at the only files present are the original source files, the new .o files, and the executable file (some versions of this compiler will not h ave the .o files since the -c sw itch was not used). The user then en ters a.out at the command-line prompt to execute the program.

Compiling in C using the #include directive would look something like this:

EXAMPLE 3

```
$ edit source.c
$ edit source1.c
$ edit source2.c
$ gcc -o exec source.c
0 errors
$ exec
```

Notice that the abo ve example h as three source files, but the pr ogrammer only compiles one of them. Presumably, the file source.c uses #include sta tements to insert source1.c and source2.c into source.c; therefore, only source.c needs compiling. The other files are added at compile time.

EXAMPLE 4

```
$ gcc -O1 -o exec file1.c file2.c
0 errors
$ ls
file1.c file2.c file1.o file2.o exec
$ exec
```

This last example modifies some of the default parameters. In this case, the programmer wants to give a n ame to the executable pr ogram, and the programmer wants the executable to be optimized. The directory listing reveals much the same infor mation as in E xample 2, except that the executable name is the name the programmer provides. This new name is used to run the program.

- Try creating your own small C programs, compiling each with gcc and running it.

The C compiler itself is composed of thr ee subsystems: the **pre-processor**, the **compiler,** and the **linker** (see figure 4.1). The first subsystem the sour ce file encounters is the C pre-processor. The pre-processor's job is to execute all the pr e-processor directives. The end result is that the original source file has been transformed into a new source file, called the resultant or intermediate file. Since the original source file could have included multiple secondary files, the end r esult of the pr e-processor is to cr eate a single r esultant text file containing all the merged #include code, plus any other changes to the original source code due to other pre-processor directives, like #define and #ifdef.

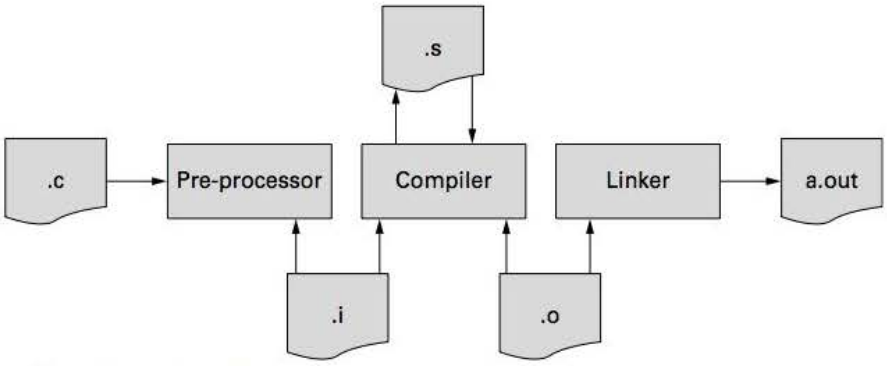


FIGURE 4.1: The C Compilation Process

The single merged r esultant file is then sen t to the compiler. It is impor tant to note th at the compiler only compiles this r esultant file and does not in teract in any way with your original files. It is further important to note that the compiler takes the r esultant file and converts it to machine language within a **local context**. What this means is th at the resultant file is treated as a private entity having its own named memory space. It is compiled without connections to the outside world (other modules, other libraries, and other resultant files) unless specifically told to connect. The end result is that everything in the outputted object file is in its own named space. This includes functions, variables, and data structures. A reference table is also generated, listing all **unresolved references**. An unr esolved reference occurs when your pr ogram is accessing a variable or a function that is not a member of the local con text (not part of your object file). These issues remain unresolved until the linker.

The linker's job is to solve all the unr esolved references the compiler discovers. It does this by searching for and then connecting each r eference to the function or var iable wanted. These references are also know as **outside** references since they ar e function calls or var iable references to things th at are outside the object file's named space. These outside connections to your program fall into three categories: operating system communication requests, programming language libr aries, and object file variable/function requests. The primary connection between your pr ogram and the r est of the computer en vironment is the oper ating system's

subsystem, known as the **run-time API**. In C, this consists of instructions that tell the OS that the main function is the starting point of your program. This main function also interfaces with the operating system's command-line prompt using the parameters argc and argv[]. This interface permits the user to invoke the program while sending data to it from the command line. This data is passed by the operating system to the C program through the main function's parameter list. The main function can also interact with the operating system's run-time environment by returning a single integer result when the program terminates. These are the default connection points between a C program and the operating system.

A library is considered to be a repository of functions that developers (like compiler suppliers or third-party developers) provide to programmers. Libraries contain programs that can be included in your application. Generally, programmers use libraries to simplify their programming. Instead of writing the code for a function themselves, programmers can instead use a library function that does the same thing. In C, the #include pre-processor directive is used to help merge those programs into your code. It is important to note that libraries come in two forms: LIB and DLL. LIB library functions are actually physically copied into your programs. This is called linking. Your program then grows in size. If you have two different programs that use the same library function, then both programs will get their own independent copy of that library function. Some view this repeating of the library as wasteful. DLL libraries have been created to deal with this waste. Instead of adding these functions directly to your program, a hook is created in your program. When the operating system's run-time library loads your program, it notices this hook and tries to fill it up. DLL stands for Dynamic Link Library. The idea is that at run-time, the operating system dynamically links your program to this library. But, this link is a simple pointer assignment. The advantage of this link is that the DLL library can be loaded into memory once. If there is another program that needs it, then its hook pointer is assigned to that already-loaded DLL in memory. Your program does not need to grow in size, and the library can be shared, freeing RAM for other things. The operating system must be able to support this feature.

To sum up, the compiler takes the output from the pre-processor and attempts to transform the resultant text file into machine language. While it converts the text to machine language, it will discover that it cannot convert those parts of the program designated as outside connections: library calls and `extern` references. The incomplete machine language program is passed to the linker. The linker then attempts to resolve the outside connections and complete the executable program.

The make Tool

The gcc compiler can compile any number of source files, linking them all together in to a single executable file. Technically, there is no need for any other software tool; but, it becomes tedious and even impossibly hard to remember all the file names in a project.

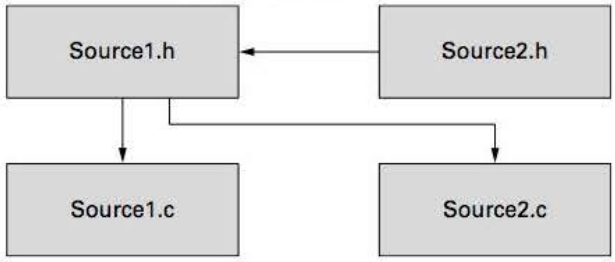
Often, medium software projects have 50 to 100 source files. Writing all these names out at the command-line every time you want to compile is enough to stop you from becoming a programmer.

A workaround is to create a batch file. A batch file can contain the command-line command already written beforehand by you. Every new file in the project would be added to this batch file. The programmer would only need to type the batch file name at the command-line prompt to get the computer to execute the long gcc command. This is a common workaround, but this technique will always compile all the files in the project. Recompiling 100 source files is a lengthy process. You would have time to watch a short comedy show on TV. Instead, we would want our compiler to compile selectively. In other words, assuming we compiled the program previously, the directory already contains .o versions of all the .c files. As long as the .c files were not modified, the .o files can be reused, which will speed up the compile. Converting a .c file to a .o file is slower than just linking an old .o file into the executable. It would be nice if we have an intelligent program that could figure this out for us.

The GNU program called `make` is an intelligent automated build utility. It automatically determines which pieces of a large program need to be recompiled and issues commands to recompile them. To determine what needs to be compiled and what can be just linked requires understanding two concepts: **file dependencies** and **file modification dates**.

Figure 4.2 shows a group of files that are interdependent. All these files make up a single software project and need to be compiled together. The arrows are dependencies. A dependency refers to a file needing another file. In this case, the file Source1.c includes the file Source1.h. Source1.h includes the file Source2.h. Source2.c includes the file Source1.c. Notice that when you include the file Source1.h, you are also implicitly including Source2.h as well. Therefore, Source1.c is dependent on Source1.h and Source2.h. This is also true for Source2.c; it is also dependent on files Source1.h and Source2.h.

FIGURE 4.2: Source File Dependencies



The `make` program needs to be told by the programmer the dependencies contained within the project. The `make` tool is not intelligent enough to determine this on its own. It gets this information from a text file called the "makefile." This makefile is an actual text file called `makefile`. It contains a collection of rules and instructions explaining how to compile a project. If you make a mistake building your rule(s), your application will not compile properly. You can create the makefile using any text editor. Save the file with the name `makefile`.

To compile your project using the makefile requires you to type the name of `make` program at the command-line prompt. The program `make` assumes that the current directory already contains a text file called `makefile`. It will open this text file and proceed to compile your project based on those rules.

For example:

```
$ make
```

The above example shows what you must type at the command-line prompt to compile your project. You simply enter `make` and then press enter. It will open the makefile and carry out the rules defined in that file.

Makefiles can also contain special software management instructions. It permits you to define **actions**. Actions are operating system command-line commands that have been tagged with a makefile action name. This action name is just an identifier name. This allows you to write mini-scripts within the makefile and then assign to them a *tag* name. To execute a specific action, specify that tag name as an argument to `make` program at the command line.

For example:

```
$ make clean
```

Now, instead of compiling your project, it will execute the mini-script called **clean**. We will talk more about this in the makefile section below.

The additional piece of information that `make` uses is the file modification dates. The operating system records statistics about the files you keep on the hard drive. Two of these statistics are the file's **creation date** and the file's **modification date**. The creation date is the date and time when the file was first introduced to the operating system. The modification date is the date and time the file was last touched or modified by something other than the operating system; for example, a file is modified when the user edits the file and changes data, or when a program opens the file and writes to it.

It is obvious that when we think of the three files: .c, .o, and executable that they are created in a particular order. The programmer creates .c, and the compiler creates the other two. The .c file must come first. The .o file must come before the executable. This means that the file dates will be in order. The .c file will be created first, followed by the .o file, and ending with the executable file. The program `make` knows this and assumes the following: if the dates of the dependent files are all in order, then I do not compile, just link the .o file. If the dates are not in order, then compile.

The makefile

This is the general syntax for all the statements in a **makefile**. Notice that it is composed of three sections: a target, a list of dependencies, and the command.

Syntax:

```
TARGET: DEPENDENCIES
        COMMAND
        ...
```

Where:

1. Target is either the name of the file you want to compile or the name of an action you want to perform.
2. Dependencies are the names of files needed by the target—they are space separated.
3. Command is the OS command-line command to carry out.
    - There can be more than one line.
    - You **must** put a tab character at the beginning of every command.

Look at the following example makefile:

```
librarydemo : main.o library.o file.o book.o
        gcc -Wall -g -ggdb -o library main.o library.o file.o book.o

main.o : main.c file.h library.h book.h
        gcc -c -Wall -g -ggdb main.c

file.o  : file.c file.h library.h book.h
        gcc -c -Wall -g -ggdb file.c

library.o : library.c library.h book.h
        gcc -c -Wall -g -ggdb library.c

book.o : book.c book.h
        gcc -c -Wall -g -ggdb book.c

clean :
        rm -f library main.o library.o file.o book.o
```

The **default rule** is the first statement in the file. In this case, it is called librarydemo and is the name of the executable file. This rule states that librarydemo is dependent on main.o, library.o, file.o, and book.o. It states that to build librarydemo, the compiler must link all these .o files. The command-line command that links them all is provided:

```
gcc -Wall -g -ggdb -o library main.o library.o file.o book.o.
```

Before the default rule's command is executed, the dependencies are checked. The check is done using the file modification dates and a file existence check. In other words, the default rule's command can only be executed under the following conditions: (1) all the dependency files exist and (2) the dependency files modification date are in order. If this is not true: if a

dependent file does not exist or the dependent file's modification date is not in order, then the compile command cannot happen; something else must happen first.

Checking whether a file exists is easy. Checking th at the da tes are in order is more involved, because just checking the da te order of the dependen ts is not enough. For example, the user may have changed the book.c file. Just looking at book.o in the default statement will not reflect this change in book.c. Only after book.c is compiled can we see it; but nothing as yet has asked it to be compiled. The `make` program by default executes the defa ult rule first. Therefore, in order to check the valid da te order of book.o, it requires a recursive search of the makefile for the component parts of book.o. They are defined in a r ule tagged as book.o . The `make` program looks for this tag. The tag must be the same as the dependenc y name. If it does not find it, it uses book.o's date; otherwise, it uses the sub-rule. It does this for every dependent of the default rule, and it recursively does this for every dependent of every sub-rule.

In the end, the sub-r ules that need to compile their sour ce files are carried out first (notice the `gcc -c` option on all sub-rules). The recursive operation returns back to the defa ult rule, which should have all its dependents in the correct date order. If not, then an error is displayed; otherwise, the default compile command is executed.

The above example also sho ws how to create an independen t tag called, in this case , **clean**. It is not a file name, and there is no dependen t that refers to it fr om any of the r ules. It will simply be called directly from the command-line prompt. You could have multiple operating system commands, each on its own line after the tag. The example shows only one command to remove all the object and executable files without prompting the user.

Since independent tags can be used that way, this is also true for any of the other tags.

Make Variables

The makefile itself could get fairly large and complex. To control the complexity, the `make` program supports variables. The example below shows a makefile th at uses var iables. Variables are defined at the beginning of the makefile. In this example, two variables are defined as the two first lines of the file: `objects` and `coptions`. The variables are then used in the rules with the escape character sequence ${ VAR } where the VAR is the name of the variable. If we compare this with C, it is similar to the #define.

```
objects = main.o library.o file.o book.o
coptions = -Wall -g -ggdb

librarydemo : ${objects}
        gcc ${coptions} -o library ${objects}

main.o : main.c file.h library.h book.h
        gcc -c ${coptions} main.c
```

```
file.o  : file.c file.h library.h book.h
        gcc -c ${coptions} file.c

library.o : library.c library.h book.h
        gcc -c ${coptions} library.c

book.o : book.c book.h
        gcc -c ${coptions} book.c

clean :
        rm -f library ${objects}
```

The Make Program's Implicit Rules

Make has an implicit rule for updating a .o file from a correspondingly named .c file. Although I would not recommend using it, I'm showing it since it is frequently used.

```
objects = main.o library.o file.o book.o
coptions = -Wall -g -ggdb

diskdemo : ${objects}
        gcc ${coptions} -o library ${objects}

main.o : file.h library.h book.h
file.o  : file.h library.h book.h
library.o : library.h book.h
book.o : book.h

clean :
        rm -f library ${objects}
```

The above example assumes for each sub-rule that the tagged .o file has a corresponding named .c file that includes the listed dependencies. It will then automatically compile with something like this: gcc -c tag.c.

The RCS Tool

Revision Control Systems (RCS) are used to manage software systems. This is also known as **version control**. An important difficulty that develops in software projects and large software teams is multiple versions of the same source code file. This situation often leads to multiple programmers editing different versions of the same source file at the same time. Merging these files at compile time is a hard job. Being aware that there is more than one version of the file is also hard. RCS can be used to manage both source code and documentation. RCS provides file management in two basic ways: rules for sharing files among team members, and a development history so that programmers can go back to previous source

versions to fix mistakes. Common mistakes needing revision history are, for example, restoring an older version of a source file because the new version became damaged or lost; creating a new program based on an older stable module of the program; and discovering that the development path that was followed led to a wrong conclusion, and you want to go back to a previous version of the program to start over again (among other reasons).

With respect to source-file sharing, RCS helps in the following ways: assigning access permissions to a source file, controlling access to the file by designating valid users of the source file, and blocking access to a file when it is currently being edited. In relation to revision history, the tool automatically increments and inserts source-file version numbers for each of the source files and generates an automatic change history report by asking the programmer questions about the change. It also gives the programmer a simple way to download the most recent version of all the source files in a project.

RCS converts source files into a database of files. This database is physically stored in the project's source directory or within a special subdirectory called rcs. The developer must create this rcs subdirectory using the RCS tool, or the RCS system will assume you want the file-management databases to exist in the project's source directory. This rcs subdirectory is created below the source directory. For example, if your home directory is called johndoe, and there is a subdirectory under johndoe called source, the rcs subdirectory would be below source.

There is a database for each source file. The database name is filename.v, where "filename" is the name of the original source file. The file extension .v designates the file to be a version control database for the file "filename." It records the revision history, change log, version numbers, etc., for a file. The programmer uses the RCS tool to add and remove versions of the file to/from the revision control system.

Many revision systems delete the original source file when it is put into the revision system. This is important to do if the system wants to ensure that it has full control over the development of the software. In other words, there should not be any extra copies of the program that the revision system is not aware of. This way, we know that the software in the databases is the most up-to-date version of the entire program.

Revision systems operate under a set of simple rules: (1) programmers must add and remove every file they want to use to/from the database. This permits the system to track who has a source file since it also records the user name of the individual who performs a check-in or check-out operation. (2) Whenever you check in, the system asks you for information. This allows the system to establish a history for the file. And, (3) permissions can control access to the source files. Only certain user names can check in and check out a particular file.

The RCS commands are described below:

Check In

Putting a file into the revision system. If this is the first time, it creates the database. Checking in occurs in two modes : regular and read-only. In regular mode, the source file is added to the database and deleted from the directory. In other words, it is a move operation from your directory to the revision database. In read-only mode, the source file is copied (not moved) into the revision database. The original source file is changed to read-only mode. This is useful if you want to still use the file to compile the program. The revision system changes the file's permission to read-only because it does not want you to modify the file without performing a check-out operation first.

Regular Check In

- `ci filename`

  The filename is moved into the revision database. It is automatically given a revision number. The programmer is prompted to comment about what has changed in the file.

- `ci -f -r3.2.1 test`

  This is similar to the first check-in command except the programmer destructively forces (-f) his own version number (-r). In this case, the filename test is moved into the database with the revision number 3.2.1. If there was a version already numbered with 3.2.1, then it is deleted and overwritten by this check-in.

- `ci -rVersion filename`

  Check in using a different (new) version no. This is not a destructive insert.

- `ci -t filename`

  Input a general *description* of the file and the *reason* for this revision. This is different from the previous ones that only asked you for the reason for the revision.

Read-only Check In

- `ci -u filename`

  Copy source file into the revision database, and convert the original source file to read-only (-u).

Check Out

In regular checkout mode, the file is extracted from the revision database in read-only mode. This means multiple people can extract the file, but no one can change it. When one programmer extracts a file in edit mode, this is known as **locking** the file. A locked file cannot be extracted for editing purposes until the original file is checked back in; but in read-only mode it can still be extracted.

Regular Check Out

1. `co filename`

   Extracts the most recent version of the file from the database in read-only mode.

2. `co -l filename`

   Extracts the most recent version of the file from the database in locked mode.

3. `co -l -rVersion filename`

   Check out an older version of the source code (not the most recent).

The RLOG Program

The `rlog` program is the revision log report program. It displays the history of a selected file. The history is defined to be the revision numbers and the comments written by the programmers when checking in a file.

For example:

```
$ rlog sample
RCS file:    sample,v
Working file:       sample.c
Head:        1.2
Branch:
Locks:
Access list:       jvybihal, rsmith
Symbolic names:
Total revisions:   2
DESCRIPTION
Sample is a program that displays pictures of products that can be given
out to customers as samples.
----------------------------------
REVISION 1.2
Date: June 5, 2008; Author: jvybihal; State: Exp; Lines added/deleted: 10/2
Adjusted image display quality on lines 20 to 30.
----------------------------------
```

```
REVISION 1.1
Date: June 4, 2008; Author: jvybihal; State: Exp; Lines added/deleted: 100/0
Inputting the program.
=============================================================
$
```

Notice what is displayed. The report first displays the file's current statistics within the revision database, and then it displa ys the log commen ts per version n umber. The revision statistics include the file's database name and real name, the most current version number, any branches, if the file is currently locked, the users who are permitted to access this file, the size of the revision log, and a description of the file's purpose.

The RCS Program

The RCS program has all the functions to m anage your revisions. We outline the most commonly used features:

- `rcs -oVersionRange filename`

  This deletes a portion of your revision history.

  Where: VersionRange can be a single version number like 2.5 or a range like 1.3:1.5.

- `rcs -sState:Version filename`

  Label a version as experimental (Exp), stable (Stab), released (Rel), or obsolete (Obs).

- `rcs -nName:Version filename`

  Assign a name to a version. For example:

  ```
  rcs -nBetatest:2.4 filename
  ```

  ```
  co -rBetatest filename
  ```

Example Usage: If someone wants the old beta version of the file you sent out to all the clients, you won't have to remember it was 2.4 since you named it Beta.

- `rcs -t test`

  Change the r evision log descr iption. Enter the descr iption terminated with a single period or the end of file character on an empty line (in the command-line version).

- `rcs -aUserNameList filename`

  Define a list of valid users. Only these users can access the file. The definition consists of their Unix user names. For example:

  ```
  rcs -aBob,Mary,Sam menu
  ```

- `rcs -Aoldfilename newfilename`

  Copy the access rights from another file.

- `rcs -eUserNameList filename`

  Delete names from the access list.

A revision database stores the source file in a **revision tree**. A revision tree is a structure that looks like a tree. It has a trunk, and it has branches. It has a root, and it has leaves. The root of a revision tree is the source file you first put into the database. This first file is the file where all other versions of the file come from. The simplest revision tree is a tree that only has a trunk. A trunk starts at a single root and progresses linearly through versions of the program until it comes to a single leaf. A leaf is the most up-to-date version of the source file. Most revision trees are in this simple form. Sometimes, you may want to develop two different versions of the same program. The program will start from a common root, but at some point, you may want to branch out from the trunk and create another version of the source file; see figure 4.3.

FIGURE 4.3: A Revision Tree

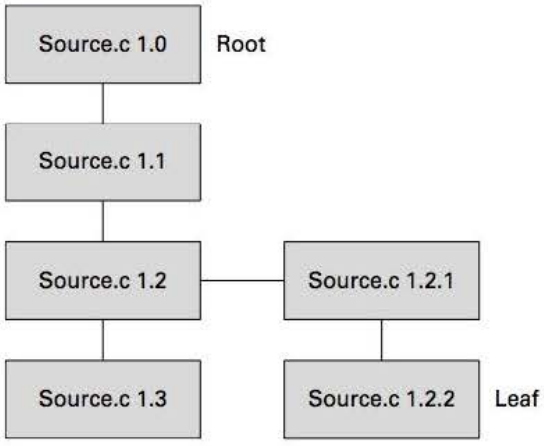


Figure 4.3 is a revision tree. The root source file is 1.0. The programmer then changed 1.0, and the system saved a new version of the program as 1.1. The database now has the Source.c source file stored twice, once as 1.0 and the new one as 1.1. The programmer then makes another change, and we have 1.2. We now have the source file stored three times. The programmer now wants to develop two versions of the program. We call these two different versions **branches**. Maybe one branch will be in English and the other in French. Or, maybe, one branch will use quick sort, and the other branch will use an experimental sorting algorithm. In any case, we now have two branches: 1.2 and 1.2.1. These two files were later updated by the programmer, resulting in 1.3 and 1.2.2. The final state of figure 4.3 shows two branches that both originate from a common root. Each branch's most up-to-date version of the code is called a leaf, and the leaves are 1.3 and 1.2.2.

The GPROF Tool

Finding where your program is slow is sometimes easy to do. You only need to run it and observe. Normally, good programmers will write their code with efficiency in mind.

The rule-of-thumb is to reduce iterations, especially nested iterations. If the iterations can be reduced logarithmically, it is even better. These are the lessons we learn from running time (Big Oh) theory.

What do you do when you have adhered to all the running time lessons, but the program is still slow? One way is to manually compute the running time, function by function, of your program. On large programs, this is a lot of work. A quicker solution is **code profiling**. Code profiling uses the idea of a stopwatch. Run the program, and compute the elapse time between each function. The slowest function is the one you should study more carefully. The problem with computing elapse times is that it depends on the CPU and computer hardware. You could technically improve the program's performance by simply purchasing a faster machine. Therefore, the results obtained by profiling are only relevant on similar classed machines. But, it can be argued that the results are proportional. In other words, even though the running time will be better on a faster machine, the slowdown experienced in a particular function will still exist, but they will be proportionally quicker on faster machines.

GNU provides a program called GPROF that automatically calculates the elapse time between every function in a program. GRPOF provides additional features: it computes the number of times each function was called and the proportion of time the program spent in each function, in terms of percentage. It is important to know how many times a function was called. If the program spent 30% of its time in a particular function but that function was called 100 times, then the 30% must be divided by 100 to find out how fast that function is all by itself.

To use GPROF, your program must be compiled with GPROF in mind. To prepare you program for GPROF, do the following:

```
$ gcc -pg file.c
```

The above command creates a gmon.out binary file. This gmon.out binary file contains the information GPROF needs to compute the program's running time. To view the running time, you need to do the following:

```
$ gprof -b a.out gmon.out > textfile
```

Where these switches refer to:

- -b not verbose
- -s merge all gmon files
- -z display a table of functions never called

The -b switch displays the elapse time report without special instructions and software information. The -s switch will merge multiple gmon files into a single report. The -z switch will provide an additional table detailing the functions that were never used.

Without the -z switch, the program displays a report that has two sections. The first section summarizes the results from the elapse times. The second and longer section provides a detailed view of every function in context with whom it called and who invoked it.

This is an example of the summary report:

```
Each sample counts as 0.01 seconds.
%      cumulative   self              self        total
time   seconds      seconds     calls  ms/call     ms/call     function
38.85    20.00      20.00           1  20000.00    20000.13    LoadData
48.56    45.00      25.00           1  25000.00    25000.00    InsertData
10.74    50.53       5.53           5   1106.00     1201.51    Optimize
 1.55    51.33       0.80        1000      0.80        1.03    Search
 0.23    51.45       0.12         910      0.13        0.13    read
 0.01    51.46       0.01          25      0.40        0.40    Delete
:
:
 0.00    51.48       0.00           1      0.00        0.00    sigvec
```

%Time indicates the proportion of time the program spent in that function. Do not think that this indicates where the program is slow. Remember that functions are often called multiple times so the per centage would need to be divided. Also keep in mind that programs are expected to be slow at launch or load time, so that may not bother the user.

Cumulative Seconds is a simple count of time. The last number in the column (in this example 51.48) indicates how long this program ran.

Self Seconds is important because it indicates how much time that function actually used up. Please remember that this figure includes the possibility that the function was called multiple times. Assuming that for every call the function behaved the same way, you can divide this time by the number of calls to see how fast the function itself is all by itself.

Calls indicates the number of times this function was called during execution.

Self ms/call is Self Seconds divided by calls in milliseconds.

Total ms/call is an important result because it includes the time of the function with calls to its children. This puts the execution time of the function in context. Maybe the function itself is not slow, but together with its child calls, it is slow.

The detail report looks like this:

```
Index %Time    Self  Children  Called    Function
-----------------------------------------------
              0.00  51.48     1/1       main
[1]    38.85 20.00 20.13      1         LoadData
              0.13   0.00     910/910   read
```

Every function in the entire program is given a little table like this one . Each function is listed below the next. Notice that the table is aligned, except for one row. The unaligned row is the row under study. In the above example, the table is analyzing the function LoadData. It is also the only row with an index number and a %Time number. The row above this row is the immediate function that invoked it. The rows below this row are all the child functions LoadData called. In this example, LoadData was invoked by main and LoadData calls a function named read.

Index is a unique number assigned by the tool. It tells us in what order the function was invoked. In our example, is tells us that LoadData was the first function called.

%Time comes from the summary report, and it tells us that the program spent 38.85% of its execution time within this function.

Self tells us in seconds how long the function ran, including the number of times it was called.

Children tells us in seconds how long the function ran with its children, including the number of times it was called and its children were called. In this example, the children add .13 seconds to the execution time. Therefore, we can conclude that any slowdown is directly related to the function itself and not its children.

Called is a multiple meaning field. In the case of LoadData, it refers to the number of times this function was called in total during the entire life of the program. In our example, this is one time. For functions main and read, there are two numbers. The first number indicates the number of times it was called within this context. The second number indicates the number of times this function was called, including those times it was called from other function. For example, let us assume a function called Write has the following called value: 820/901. This value says that Write was called 901 times, but only 820 of those times were directly related to the current context. Our context is the activity occurring around LoadData.

To find the slowest function would require you to study all the values in detail.

The GDB Tool

The GDB debugger lets you run a program while debugging it by querying the program's run-time environment visually. For example, the programmer can run a program to a specific line number and then have it pause there to look at the state of the variables and run-time stack at that particular moment. Then the program can continue executing using a stepping-one-instruction-at-a-time method, pausing at each step to look at the variables and run-time environment.

Before GDB can be used, you need to prepare your program. GDB must be able to access the program's source code and symbol table. The source code is used to show where you are currently in the program. The symbol table links the memory variables with their corresponding variable names.

The program is prepared at compile time with the -g flag:

```
$gcc -g -o HelloWorld HelloWorld.c
```

This example shows the compiler program, gcc, being used with two switches. The -o switch compiles the HelloWorld.c program into an executable called HelloWorld with no file extension. The -g switch prepares the executable to work with GDB.

To run GDB, you do the following:

```
$gdb HelloWorld

GNU gdb Red Hat Linux (5.3post-0.20021129.18rh) Copyright 2003 Free Software
    Foundation, Inc. GDB is free software, covered by the GNU General Public
    License, and you are welcome to change it and/or distribute copies of it
    under certain conditions. Type "show copying" to see the conditions.
    There is absolutely no warranty for GDB. Type "show warranty" for
    details. This GDB was configured as "i386-redhat-linux-gnu". . .

(gdb)
```

You simply write at the command-line the gdb command and the name of the executable file that was prepared. The tool begins by welcoming you and then displays the GDB command-line prompt, which is `(gdb)`.

When using GDB, you can always type in the "help" command to get a list of available commands:

```
(gdb) help
```

You can also get help on a specific command by typing "help" and the name of that command:

```
(gdb) help breakpoints

 "Making program stop at certain points."
```

You can use the `list` command to display the source code you are debugging. List either takes a filename and a line number, or a function name, or just a line number. For example:

```
(gdb) list main.c:5
5
6 int main (int argc, char **argv) {
7
8 library* mylibrary = createLibrary(20);
9
10 loadLibrary("lib.txt", mylibrary);
11
12 addBookToLibrary(mylibrary, createBook("Lotr", "Tolkien", 300));
13 addBookToLibrary(mylibrary, createBook("Harry_Potter", "Rowing", 50));
14 addBookToLibrary(mylibrary, createBook("C_Prog", "Kerning", 100));
```

In the above example, the `list` command uses a filename and a line number separated by a colon. The listing will begin at line 5 of main.c and display the default 10 lines of instructions.

You can quit GDB with the quit command:

```
(gdb) quit
```

Program Execution

There are four commands to control how a program executes within GDB: `run`, `continue`, `step` and `next`.

The command `run` will execute the program from the beginning as if it were at the Unix command-line prompt. The program will run without interruption until completion or until forced to stop. There are two ways of breaking into the program while it is running. The first way is to use the `control-c` key combination from the keyboard. This only works when the program is generating some I/O. When you press control-c, the program will break and pause at that point and display the GDB prompt. You can then do what you like at that prompt. The only drawback with control-c is that you do not have control over where it will stop. If you want exact control over its stopping point, you must use the `breakpoint` command. This will be covered later.

The `continue` command, unlike the `run` command, executes the program continuing from the current line number where the program paused. Remember that the command `run` starts executing from the beginning of the program. Normal usage would be to `run` your program the first time only, and then pause it with a `breakpoint`. Look around in memory, and then use the `continue` command to continue executing from that point.

The commands `step` and `next` are similar. When the program has been paused, you can use these two commands to execute one line of code and then pause automatically again. This way, you can incrementally execute your code one line at a time at critical locations. After each increment, you can look around in memory to see what is going on. The difference between these two commands occurs when the instruction you are about to execute is a function call. The command `step` will follow the function call and step through the function's instructions as well. When that function returns, `step` will also return back to the caller and continue stepping from there. The command `next` executes the function call, but you do not follow stepping into that function. Instead, you skip stepping through it and continue to the next instruction.

For example:

```
$ gdb HelloWorld

(gdb) list
1 int main(void) {
printf("Hello World\n");
}
```

```
(gdb) break 2
Breakpoint set at line 2
(gdb) run
Program paused at line 2
(gdb) step
Hello World
(gdb) step
** Program Terminated
(gdb) quit
```

Summary of GDB Commands

This is a descriptive list of the most commonly used GDB commands.

GDB command	Description
quit	Exit from GDB
list	Show 10 lines from your current position
list n,m	Show lines n to m
list function	Show all the lines in a function by name
run	Run your program from the beginning
run (later ctrl-c)	Run, then interrupt program
run –b < invals > outvals	Redirect input and output to program (as in Unix command-line)
backtrace	See the run-time stack
whatis x	Show x's declaration
print x	Show value stored at x
print fn(y)	Execution fn with y as parameter
print a @ length	Show "length" elements of array "a"
break line	Interrupt program at line number
break function	Interrupt program at function call
break line if expr	Interrupt at line number if expr true
break fn if expr	Interrupt at fn call if expr true
break file:line	Interrupt at line number in source-file file
continue	Continue program execution after break
watch expre	Stop program as soon as expr is true
set variable n = value	Change contents of a variable at run-time
ptype n	Pretty print of variable n
call fn(y)	Execute fn with parameter y

For example:

```
(gdb) whatis p
type = int *

(gdb) print p
$1 = (int *) 0xf8000000

(gdb) print *p
$2 = Cannot access memory at address 0xf8000000

(gdb) print $1-1
$3 = (int *) oxf7fffffc

(gdb) print *$3
$4 = 0

(gdb) break 17
Breakpoint 1 at 0x2929: file.c, line 17.

(gdb) break 30 if x = = 100
Breakpoint 2 at 0x3550: file.c, line 30.

(gdb) info breakpoints
Num   Type         Disp  Enabled    Address   What
1     breakpoint   Keep  Y     0x2929    in calc at file.c: 17
2     breakpoint   Keep  Y     0x3550    in sum at file.c: 30

(gdb) delete 1          ß delete breakpoint 1
(gdb) delete            ß delete all everything
(gdb) clear 17          ß turn off any break or watch on line 17
(gdb) disable 2         ß do not delete but turn off breakpoint 2
(gdb) enable 2
(gdb) enable once 2     ß turn on for one time
```

At any moment, even after a crash, you can use the `where` command to produce a **backtrace** of the run-time stack.

```
(gdb) where
#0  createBook (title=0x804a218 "Lotr", author=0x804a110 "Tolkien",
    pages=300) at book.c:8
#1  0x080487fe in loadLibrary (filename=0x8048aa8 "lib.txt",
    myLibrary=0x804a008) at file.c:20
#2  0x08048567 in main () at main.c:10
(gdb)
```

Often, when you run a program from the Unix command-line prompt, the program will crash with an error message that says **Core Dump**. A core dump is a serious error. It means that the program did something illegal, and the operating system removed it from memory. The entire

program and memory space is RAM and is copied onto the hard disk in a binary file called a core dump. This binary file can be loaded into GDB. With GDB, you can find out what happened.

Do the following:

- Compile the original program again, but with the -g switch.

```
$ gcc -g file.c
```

- Run GDB with the core dump file.

```
gdb a.out core.123
```

- GDB will the display some helpful information and permit you to run the program from within its environment:

```
GDB is a free software and you are welcome
to distribute copies of it under centerain conditions;
GDB 4.15.2-96q3; Copywrite 2000 Free Software Foundation, Inc.
Program terminated with signal 7, Emulation trap.
#0 0x2734 in swap (l=93643, u=93864, strat=1) at file.c:110
110     x=y;

(gdb) run
```

The above tells you that the error occurred on line 110 of the program with error number 7, which is the emulation trap. This information is a little more helpful than the words "core dump."

THE OPERATING SYSTEM AND C API

All programming languages have, in their library, functions that interface with the operating system. C is special because it also has commands that permit the programmer to directly access the computer's hardware, even though this is normally the responsibility of the operating system. Therefore, the normal programming convention is to use the operating system to interface with the hardware, as a proxy of the program. Activities we can ask the operating system to do are: create files, access the shell memory, interact with the command line, and intercept hardware and operating system interrupts (to name a few).

STDIO.H

The stdio.h library implements a concept known as **streams**. A stream is a device that helps us communicate with peripherals that process contiguous series of characters on a character-by-character basis. Examples of peripherals that operate this way are the computer screen, the keyboard, the printer, and the hard disk (actually, any secondary storage

medium). Since all these peripherals are based on the same technology, we can access them all the same way. This is very nice. The stdio.h library gives us functions that interface with all these peripherals.

The stdio.h library assumes that the keyboard and screen are default peripherals that will always be present. Therefore, three stream devices are automatically created for the programmer. They are called **stdin**, **stdout**, and **stderr**. The device stdin is connected to the keyboard. The device stdout is connected to the screen. The device stderr is also connected to the screen. The library function printf() uses stdout by default. The function scanf() uses stdin by default. All run-time error messages issued by the compiler or operating system use stderr by default.

The stdio.h library provides a special pointer structure that the programmer can use to create a stream. This structure is called FILE. It is written in all caps. Do not get confused by the name. It can be connected to any streamed peripheral, not only files.

We already saw in the Unix chapter that stdin and stdout can be redirected using the greater-than sign, the less-than sign, and the pipe sign. This is true here as well; we will not talk about that other than to say that: at the command line, a program can be invoked with the redirection signs. If this is the case, then the stdin and stdout streams have been reassigned, and hence, printf() and scanf() will process to/from different sources. For example, normally, scanf() reads from the keyboard; but if redirection from the command line was used, then scanf() will read from the file being redirected to the program. From the point of view of the program, it will think it is still reading from the keyboard and following all the standard keyboard reading rules, but it will be actually receiving its data from the redirected file. The printf() function, if redirected, will output its information to the redirected file instead of the computer's screen. Like the scanf() function, the printf() function will output following all the standard screen output rules, even though the destination is to a file. It will look normal, or as expected, in the file. The file will display things in the same order you would have expected to see it on the screen.

To illustrate this streaming interface, we will look at connecting to the printer. I would like to remind you that you should also review the file I/O section of the C chapter. You will notice that the FILE structure was also used there.

Interfacing with a printer using stdio.h is trivial, if you already understand how to read and write the a file. A printer is a machine that accepts data from the computer and prints that data onto a page. The printer does not communicate back to the computer, except when it sends status updates and error messages. A status update would include a message like, "I am ready to print," and "I have finished printing." Error messages would include, "Out of paper," "Out of ink." and "Printer is not ready." The error and status messages are handled by the operating system. The programmer is only aware of them when requesting access to a printer. The programmer can find out if access was not granted. Otherwise, the operating system handles all other status and error messages.

As a programmer, the following activities can be performed on streams:

- You can request access to a stream (this is called **opening** a connection). The system can deny you access to the stream by returning the NULL value. To request access, you must use the name of the stream you want to access. For example:

  ```
  FILE *ptr = fopen("prn","w");
  ```

  In the above example, the FILE structure is used, an identifier which is type pointer of FILE is declared, and then the fopen() function is used. Instead of providing the name of a file, the programmer provides the name of the stream. In Unix, "prn" is a common name for the printer. Sometimes, it might be "lpt" or "lpt1" if there is more than one printer. You would have to ask your system administrator for the name of the printer, but before that, you can try "prn" and "lpt" and see what happens. Note that ptr will either receive a pointer that points to the stream that is connected to the printer or it will receive a NULL from the operating system if the operation failed for any reason. You are not told the reason through the fopen() function.

- You can terminate communication with a stream (is is called **closing** the stream). For example:

  ```
  fclose(ptr);
  ```

  This is identical to how we closed a file.

- You can send data to a stream, assuming the machine you are connecting to can receive information. For example, a mouse cannot receive information from a program, other than status and error messages. Our printer example can receive data, so to print on your printer, you need to do the following:

  ```
  fprintf(ptr,"I am printing on the printer: %d\t\n\f",10);
  ```

  The above example uses the fprintf() function to send information through the pointer. Keep in mind that the pointer is connected to the steam. Any data the stream receives will be forwarded to the machine, in this case, the printer, but it could be any machine. Notice that all the standard printf() function rules apply. The % escape-character-sequence in all its forms is valid. Also the backslash escape-character-sequences are also valid. Important ones to be aware of are: \f which produces a **form feed** operation. This operation causes the current page to exit the printer and a new page to be loaded. The \n (new line) and \t (tab) work as you would expect. The fprintf() function also returns an integer number, not shown in the above example. The integer number indicates the number of data items successfully received by the stream from the program. To get this value, do the following:

  ```
  value = fprintf(ptr,"Message %d %d\n",x,y);
  ```

The variable `value` is integer and receives, in this case, the values 0, 1, or 2 since this fprintf() function uses only two % escape-character-sequences. If nothing went through, then a zero is returned. If only one of the values got through, then the number one is returned, etc.

- You can receive data from a stream, assuming the machine you are connected to can send information to the program. In our example, this is not the case; printers do not send data back to the computer, other than status and error messages that the operating system intercepts. Keyboards and files can send information to the program. To receive information from a stream, do the following:

  ```
  fscanf(ptr,"%d %c",&var, &var2);
  ```

  If the machine is sending data to the computer, then the stream will store this information. The function fscanf() can access the stream with the pointer to receive the information. If there is no information available at this time, the fscanf() function will wait until the information arrives. The fscanf() function also returns an integer number following the same rules as we have see with the fprintf() function.

Shell Memory Interfacing

Modern operating system shells are special programs that handle all user interface requests. They are also the enveloping run-time environment from which programs are launched. Shells possess three special features: a command-line, an interpreter, and a shell memory. Programs can access these three features.

The command line is a function that displays a text prompt to the user. It accepts an input string entered at the keyboard or passed as a parameter. The information entered is in the form of commands. The command-line function attempts to perform the requested operation by either performing the activity itself, executing an external program, or calling the required operating system library function.

The command-line function runs within an environment called the Shell. The Shell is a protected memory space that is associated with all your running processes. If this is a central computer architecture, then each user is given his own Shell with programs that run within that environment, independent from the other shells and users. Shells can be loaded recursively. This means that a user or a program, while in a shell, can load another shell into memory as another layer. The original shell is still present, but it is inactive until the user executes the EXIT command to terminate the new shell and return to the original shell.

One of the major features of the Shell is its environment variable *space* (or *shell memory*). This is an area of RAM where the user or programmer can store information as strings. The shell organizes strings as a list. The shell expects the string to be in the following format: a variable name followed by the assignment symbol (=) and then a value. An environment variable entry

could be the following: "path=c:\data". 'path' is the variable, and 'c:\data' is its value. The equal sign is the separator. This memory space is global to all running processing sharing the same shell. The memory space is also persistent as long as the shell is still running. As long as the shell has not been terminated, the environment variable space retains its information. In other words, a process can post something into the environment variable space and then terminate. After the process has terminated, all its local memory is deleted from memory—as is normal for a terminated process, *but* the data stored in the environment variable space still exists and is accessible by the programmer at the command-line prompt, by another running process, or by a process at a later date—as long as the shell was still running.

The advantage to the programmer is that processes can communicate with each other through the environment variable space. One process can post a value in this space, and another process can read it—as long as they are in the same shell. The environment variable space is also useful to the user. The user can use the SET command-line command to put values in this memory space. Then an executing program can use the value the user just stored in the shell memory.

If your program wants to use the shell memory, or if your group of concurrently executing programs wants to intercommunicate using the shell memory, then you would have to use C's environment functions. These environment functions influence only the current shell. The two functions are getenv() and setenv(), which can be found within the stdlib.h file.

The getenv() function uses one string argument and returns one string result. Remember that information stored in the environment memory is formatted in the following way: name = value. The getenv() functions has this syntax: `valuestring = getenv(namestring);` The namestring is the name of the shell memory variable. If it exists, valuestring is assigned the value of the shell memory variable; otherwise, it is assigned NULL.

To insert your own memory variables from a program, use the setenv() function. Its syntax is the following:

```
int setenv (const char *name, const char *value, int overwrite);
```

You would use it in your program this way: setenv(name, value, 1); the overwrite parameter is set to 1 or 0. When set to 0, the string name=value will be added to the environment memory if there is no other variable with the same name. When set to 1, that does not matter. The variable will be added to the shell's memory regardless. If there was a variable with the same name, it is destroyed and replaced by the new string.

Assuming that two processes are running under the same shell, the shell's memory becomes a type of global memory. It is accessible by all processes running under that shell. As soon as one process invokes setenv(), the data is available to the other processes via getenv(). This is interprocess communication. Now the processes can coordinate their efforts through this form of message passing.

The example code below shows how a C program can ask for the default search PATH defined within the shell's memory. If getenv() cannot find the name, it returns NULL.

```
/* getenv example: getting path */
#include <stdio.h>
#include <stdlib.h>

int main ()

  char * pPath;
  pPath = getenv ("PATH");
  if (pPath!=NULL)
    printf ("The current path is: %s",pPath);
  return 0;
```

System Calls and the Command-Line

To execute a comm and-line command in a fr om a C pr ogram, the program simply calls the function system(). It is a stdlib.h function. System() takes a single str ing as an argument. That string contains the command in the same for mat the user would h ave used at the command-line prompt. For example: `system("ls -l -a");` The system() function passes the str ing to the operating system. The operating system puts the C program to sleep. It starts up a new shell (not the same shell the program is in). The command is executed within that new shell. Once the command has run its course, the comm and terminates. The shell then also ter minates. The operating system activates the old shell and wakes up the C program. The C program then continues executing from the point after the system() call.

There is a minor problem with this way of executing a command. Since the program is executed in a new shell, the environment variables set with values from the old shell are not accessible. Similarly, any environment variables set in the new shell ar e lost once the system() function terminates.

A work-around is to access shell memory directly and launch all your programs within the same shell. To launch multiple programs within the same shell from the command-line prompt, you use the special ampersand symbol. Here is an example:

```
$ sort names.txt &
$ ls -l &
$ print names.log
```

The ampersand at the end of the command line permits you to input another command within the same shell. The new comm and is executed w ithin the same shell concur rently with the previous command.

Alternatively, the commands could have been entered at the command line in the following way:

```
$ sort names.txt; ls -l; print names.log
```

The semicolon executes each of the commands, one at a time in the order presented. The semicolon is a better method for programming because it can be used from within a system() function. It would look like this:

```
system("sort names.txt; ls -l; print names.log");
```

In this case, the system command starts a new shell. In this new shell, the three programs are executed sequentially in the same shell environment. Sadly, this is not the same shell environment as the parent program, but it is the same shell for the child commands.

Process Calls (Fork) and Inter-Process Communication

Before we can talk about inter-process communication, we need to learn about creating and managing multiple processes. We need to understand some basic concepts that impact inter-process communication due to how new processes are invoked and how they are represented in memory by the operating system. These concepts are general and true for all operating systems. Remember that the term **process** means a program that is executing.

Operating systems permit three basic ways of invoking a process from a program: **parent-child lock step**, **parent-child concurrent**, and **independent process**. An important property with the parent-child models is that the termination of the parent process also terminates all child processes, immediately. A terminated child process does not affect any other process unless the child itself was a parent to a process. Independent processes can be terminated without affecting other processes unless the independent process became a parent.

```
int main()
   int pid = fork();
   if (pid == 0)
    ChildProcess();
   else
    ParentProcess();
   pid = wait();
void ChildProcess()
 …
void ParentProcess()
 …
```

Parent Process (pid = 1022)

```
int main()
   int pid = fork();
   if (pid == 0)
    ChildProcess();
   else
    ParentProcess();
   pid = wait();
void ChildProcess()
 …
void ParentProcess()
 …
```

Child Process (pid = 0)

C's fork() and clone() function implements parent-child concurrent process creation. The difference between them is how data memory is treated. In the case of fork(), the parent and child process get their own private copies of data memory. In the case of clone(), the parent and child processes share the same global memory and file pointers. It is best to describe this with an example. Assume we have the following:

One program is called the Parent Process, and the other program is called the Child Process. It is the Parent Process that is initially launched. The Parent Process calls the fork() function. There are no parameters. Fork() creates a child process. The child is the same program as the parent. It is a copy of all the code and data. Both parent and child run concurrently. The child process, when launched, does not begin execution from the start of the program. Instead, it begins execution after the child's fork() function. If there is more than one fork() function, then it is the same fork() that the parent invoked to make the child. After invoking fork(), the function fork() returns an integer number. It returns the number zero in the child process, telling the process that it is the child process. The parent process receives the child's actual process identification number (called the pid). The parent process knows it is the parent because it does not receive a zero. A fork() return value of negative indicates an error. The child and the parent have the same identical code and memory, but it is not the same physical memory. The child gets its own independent memory, but the contents of this independent memory are a snapshot of the parents. In other words, if in the parent process a variable was assigned a value, then after the fork() function, the child process has that same variable with the same assigned value as was in the parent. But after the fork() function, these two processes, parent and child, can evolve separately, even though they begin identical with respect to memory.

The code then shows how to call two different functions, depending on the process type: parent or child. This is established by the if-statement checking the pid number. If the process was the child, the pid number will be assigned the value zero. If the pid is greater than zero, then it is the parent process. Take a look at the code.

In operating systems that implement multi-threading, the code base is actually physically the same code base in RAM, only a new thread was generated to track the execution of the new process through the common code base. In non-multi-threaded operating systems, the code base would have to be physically duplicated and given to a process, one to the parent and the other to the child.

The function's fork() and clone() implement parent-child concurrent processing. They can be programmed, artificially, to also operate in a lock-step manner. The function that truly implements lock-step is the system() function call. You may recall that the parent process is put to sleep and waits for the child process to terminate before resuming execution. This is true lock-step. The parent is locked until the child is finished. A more useful lock-step called **process synchronization** can be implemented using fork() and close(). In a loop, the parent process can use the function `wait(pid);` to stop its own execution until the child process with pid has

terminated. This permits the parent process to stay active while waiting. But, process synchronization is even more useful combined with inter-process communication.

Inter-process communication comes in three forms: **message passing, shared data,** and **library functions** that use the pid. By default when you use fork(), the child process is created with a copy of all the global and current local variables the parent had when invoked. This means that the parent can assign some values, strings, etc., to global variables that have been placed in its memory on reserve for the child. The child would have those values after the fork() was invoked. The parent could then intelligently invoke children with multiple fork() calls. Each call would proceed with a different assignment of the global variables. This way, multiple children could be called, each being asked to do something different. This is a simple form of message passing. There is no direct way for the child to return results to the parent, other than writing to a file.

The clone() function operates like fork() except that it does not duplicate private copies of the data space for the child process. Instead, clone() gives the child process access to the parent's memory space. This means that as the parent executes and changes the value of a global variable, all the children see this change. The same is true when child processes change global variables. The parent will see the change. This is a more direct way to intercommunicate. The drawback is the standard drawback when using global variables: there is no control when misused. If a global variable is assigned an improper value, then the entire group of child and parent processes crashes.

Library functions exist in C that use the pid to determine the status of the process associated with the pid number. Two examples from C are provided here:

- `pid = wait(&status);`

  Parent process sleeps until a child process terminates. Parent is then awakened with the pid of the terminated child. It does not matter whether there are multiple child processes. Once any one of them terminates, the result is returned to the parent. If the parent does not want to terminate until after all its children have terminated, then the parent must have a wait() for each child process. The integer variable, status, contains the error termination number from the child process. If there was no error in the child process, then status is zero. If the process has no children, then the call to wait() is ignored.

- `pid = waitpid(pid, &status,0);`

  This is similar to wait() except that you specify which child you are waiting for. In this case, a child process could terminate, but it won't wake up the parent until it is the one with the same pid.

- `wait()`

  Will wait for any process to terminate.

Inter-process Communication

Inter-process communication exists when processes send data to each other while they are executing in RAM. This is similar to people speaking to each other in the same room. Through our conversation, we can influence each other. In a similar way, processes that are executing concurrently can influence one another given the data they send. Inter-process communication can be **broadcasted** or it can be **directed** communication. Broadcasting refers to data transmission originating from a single process to all other executing processes. Directed communication's intended audience is only a single process. We have already seen how using fork() and close() can allow communication between a parent and child process. We will look at some other ways here.

Using the System Call

The parent-child lock-step model has the parent invoking a new child process. The parent process then sleeps until the child process terminates. In C, we can invoke a process in this model using the command:

```
#include <stdlib.h>
int system(string);
```

The argument string contains the filename and path of the program you want to invoke (we have already seen that it could instead have a command-line command). The format of the string is in the same format you would use from the command line when referring to a program. In Unix, that would be `/directory/filename.extension`. The child process is executed within its own Unix shell, different from the parent.

Since the parent goes to sleep when the child executes, inter-process communication is limited to a single message sent to the child when it was invoked and a single message sent back to the parent when the child terminates. Sending a message to the child can be done in two ways: using a file or using the command-line arguments.

For example, the parent could write a message in a text file before invoking the child. When the child is invoked, the child would open the file, read the message, and do what the message asks. When the child is done, it would write the answer to another file and terminate. The parent would open that file and find out what the child program had done. To work properly, the parent and child programs would have needed to already know the names of the files they were to have read and written to. If the parent program is using the command line to communicate with the child, then a text file, from the parent to child, can be avoided. A terminating child process's main function can use the C language `return var;` statement to send a single integer value to the parent, avoiding the use of a text file for communication, from child to parent.

Using the Command-line

We can use the command line to send information to the child process. We already know that we can do the following in C:

```
$ sort file.txt
```

Above is a Unix prompt followed by a command-line request to launch the program sort with the command-line argument file.txt. We have also seen in C th at we can wr ite a C m ain program that can access the command-line arguments:

```
int main(int argc, char *argv[])
```

The strings "sort" and "file.txt" would be in cells 0 and 1 of the ar ray argv. We can also do this with the system command. It would look like this:

```
X = system("/.sort file.txt");
```

Some Unix systems require you to input a path name, even when none is needed. In that case, the "/." serves as "use the cur rent directory please" marker. The above system function call does the same thing as the command-line prompt example. The child process can then return an integer number from the main program. In operating systems that support this, the main program's return value is r eturned to the pa rent process through the system() function call 's return value. In operating systems that do not permit this, the system command simply returns a message from the oper ating system indica ting that the request process launched without problems.

Using Files

The more elaborate form of inter-process communication happens when programs use files. A file is a structure the operating system manages. It is an entity separate from the process. The operating system provides a file interface library called stdio.h. This library supports functions like fopen, fpr intf, and fclose. Since the file is a separ ate entity from the process, then more than one process can access and m anipulate the file. Every operating system provides rules that govern how you may access a file. If you obey those r ules, then data sharing through files is a very practical solution.

Files, for in ter-process communication, are used in two wa ys: to stor e data long ter m or for message passing. Da ta files come in m any basic for mats: **flat file**, **comma-delineated** text file, and **database**. Databases are structured files. They are composed of a set of records. Each record is an alogous to the C str uct data structure. Each r ecord contains a fixed number of fields. This is analogous to the field in a c struct data structure. A database is constructed with the idea that each record will fully represent the information about one entity. For example, let us say we want to have a membership database. If the database has 10 records, then it is storing information about 10 different memberships.

A comma-delineated file (or CSV, comma-separated vector file) is a simple text file where each row in the file is a record and each field is delineated by a comma. A row is defined to be a series of ASCII characters terminating with a carriage return and line feed. Each field is defined to be a ser ies of ASCII characters terminated by a comma. This implies that the comma character cannot be used as a character in a field. This is also true about the carriage return and line feed.

A flat file is considered to be an unformatted text file. The data is stored without consideration of records and fields. An email or a word processing document could be examples of a flat file.

Below is an example of a CSV file:

```
Jack Smith, 20, 110045
Bill Williams, 18, 110046
Ruth Ann Jamison, 25, 10047
```

The above CSV file contains three records. Each record is a ser ies of characters terminated by the new line character. Each record is composed of three fields separated by commas. It is important to note that we cannot use commas in our data because this would cause confusion. We could overcome this confusion by adding a special escape character. We have seen escape characters in C. The C printf statement uses special codes like \n and %d. The backslash and the percent symbols are the escape ch aracters. They tell the pr ogram that what follows the escape character has a special meaning. If you wan ted to use the escape ch aracter as part of the data, then you would repeat it, like \\ or %%. In CSV files, the double quote is used as the escape character, except that you use it to define a string. Everything between the begin and end quote should be taken literally.

Inter-process communication through data files occurs in an inciden tal manner. Data files, like databases, are not used to suppor t inter-process communication. But, a side effect of a database is that other processes will know when a record has been changed. In this way, a process is aware of the activity of other processes. A process could even be written to wait for a specific change in the database. When that change occurs, the process would then perform the task for which it was constr ucted. A second pr operty of files is that a file is maintained by the operating system and not b y the program. This means that the file exists on its own, regardless of the programs that use it or the program that created it. If a program terminates, the file could still exist, waiting for another program that may become active at a later date.

A more common way to do inter-process communication is to use a message passing mechanism. The simplest form would be a system that uses flat-file handshaking. Handshaking refers to an agreed-upon protocol for how we should take turns talking to each other. Applied to processes, this refers to the expected flat files on disk and their meaning when present.

First, we need to take note that the operating system has some rules when it comes to files. The *first rule* is that any number of processes can open a file if the file will only be used for reading.

The *second rule* is a file can only be opened for writing if no one else is using the file. *Third rule*: Once a file is opened for writing, no one else can open the file for reading or writing. This puts some restrictions on how we can use the files.

Message passing falls in to a couple of ca tegories: **producer-to-consumer** and **producer-and-consumer**. Producer-to-consumer occurs when one pr ocess always sends messages to another process and the r eceiving process never sends messages . Producer-and-consumer communication has both processes wanting to send messages to each other. In this case, every process needs to be able to both send and receive messages.

The **producer-consumer** methodology is the simplest to implement, and we consider it here. There are two ways the producer may want to communicate with the consumer. The first way is called **lock step**. This means that the producer wants to send one message to the consumer and then wait un til the consumer h as read the message befor e sending another message. In this way, only one message is in pla y at any time. The other comm unication mode is called **multiple *messages***. In this technique, the producer wants to send messages to the consumer, regardless of what the consumer has read. Often, this results in many unprocessed messages waiting for the consumer. Let us look at lock step and leave multiple messages and producer-and-consumer to the exercises.

The temptation is to use a single file for lock-step communication. The producer would write to the file, and then the consumer would read the message placed in the file. The difficulty with using a single file is that one process does not know when the other process has finished. We do not have information about when a pr ocess has finished using the message . Also, multi-processing operating systems do not guarantee that all the executing processes take turns, in a lock-step fashion, when executing. It is possible that one process gets more than one chance at a file before other processes. This means that the producer or consumer would not notice the other process's access of the file. We need a better way.

The solution is to use three flat files: message.txt, producer.txt, and consumer.txt. The file message.txt will contain the message the pr oducer wants to send to the consumer . The files producer.txt and consumer.txt are **signaling files**. They do not con tain any infor mation but are used to send a sign al to another pr ocess stating that some event has occurred. The starting situation is that none of these thr ee files is pr esent on the h ard disk. The algorithm uses the following handshaking rules:

- If all three flat files are not present or consumer.txt is present, then it is the pr oducer's turn. The producer can destructively write its next message to the file message.txt. The producer then deletes consumer.txt and finishes by creating an empty producer.txt.
- If producer.txt is present, then it is the consumer 's turn. The consumer reads the information in message.txt and processes that message (whatever that would mean depends on the applica tion in question). Then the consumer deletes the pr oducer.txt file and creates an empty consumer.txt file when it has finished.

This technique uses another concept called the **busy wait**. The busy wait is a simple but quanta costly method of waiting. The idea is to use a loop to open a file repeatedly until the desired result is obtained. For example:

```
While ( (ptr=fopen("consumer.txt","rt")) == NULL);
```

The above code snippet will loop indefinitely until the file consumer.txt has been created by some other process. This is an example of a busy wait. The lock-step handshaking rules have two busy wait loops, one for each rule. Look at this carefully, and convince yourself that it works correctly.

This lock-step method can be used for mor e than two consumers. The file message.txt could contain a message with two fields: ID, MESSAGE. We could use a comma-delineated format to specify the ID number of the process for whom this message is intended. Since operating system rules permit multiple reads, then all the consumers would be triggered by the producer.txt signal. They would all read the file, but only one of them would process the file. The consumer with the owning ID number would process the message, write out the consumer.txt signal, and delete the producer.txt signal. This would happen without any program modifications.

There is one drawback to file inter-process communication. It is slow. File processing is the slowest medium on today's computers. Keep in mind, though, that modern computers are very fast, and so for simple applications, file inter-process communication is an easy and practical solution.

Supported Messaging Libraries

The operating system and the compiler often pr ovide special libraries that facilitate communication between processes. In C, this is the pipe paradigm. This paradigm uses a stream. The stream is initiated between two processes using the pipe. One process is permitted to write to this shared stream, and the other process reads from it. If the two processes want to send messages to each other, then two streams are needed. Streams are unidirectional.

Let us look at the following example:

```
#include <stdio.h>
main()
   FILE *fpipe;
   char *command="ls -l";
   char line[256];

   if ( !(fpipe = (FILE*)popen(command,"r")) )
     // If fpipe is NULL
      perror("Problems with pipe");
      exit(1);
   while ( fgets( line, sizeof line, fpipe))
     printf("%s", line);
   pclose(fpipe);
```

In the above example, we see that a stream is much like a text file. What we already know about fopen, fprint, fscanf, fclose, fgets, etc., we can apply to pipes. We open a pipe in a similar way we open a file. The function popen() is a lot like fopen(). There are two arguments. The first argument is the name of the program on disk you want to execute. The second argument specifies the direction of the pipe. Indicating "r" permits the parent to read the results from the child. The processes execute concurrently. The output the child wrote to stdout will be readable by the parent. Notice the pointer fpipe that is read from using fgets. The parent reads from stdin. If popen() was invoked with "w," then the read is in the other direction, from parent to child, and the parents must write to stdout.

CHAPTER FIVE

Understanding Internet Programming

Web applications are an interesting subject for software systems because the intercommunication between processes and the communication between software systems is taken to the extreme. Web applications do not exist in the form of a single program written in a single programming language running on a default operating system. A web application is composed of many mini applications constructed from multiple languages running on any number of client platforms (OS).

THE INTERNET RUN-TIME ENVIRONMENT

The Internet is a **distributed** and **redundant** military platform, developed by the Advanced Research Projects Agency (ARPA, later known as DARPA) in 1965. This initiative was due to the USSR launching Sputnik, which mobilized the USA to organize ARPA in early 1958. Many successful projects came out of this organization, helping the USA take the technological lead back from the USSR. The goal behind the Internet was to construct a robust and useful information-exchanging network that would survive after a limited nuclear attack. In this light, the Internet was constructed to be modular and self organizing. If a network node was down, then the data would be automatically re-routed through another path.

The Internet's construction is based on three fundamental technologies: the **backbone**, the **server-side**, and the **client-side**. The backbone is beyond the scope of this text. The reader should refer to an introduction-to-networking textbook, but, in brief, the backbone is a term that refers to the primary medium that interconnects computers into a network. Normally, a network is a single backbone, like a wire, that is connected to every computer in the network. This is not a secure arrangement because if that single wire is cut or damaged, then the computers connected to that wire lose their ability to communicate with each other. If the

backbone was made from multiple wires, and one wire stopped functioning, information could continue to flow across the remaining wires. The image at the beginning of this chapter shows how different nodes (represented a bright white spots) can be connected through multiple paths (represented by arcs).

Nodes in the Internet are computers. A node can either be a **server** or a **client**. A server has a special program that listens to the Internet for requests specifically addressed to that server. These requests come from clients. A client is thought of as a limited machine that depends on a server for its functionality. For example, your PC with a web browser would be considered to be a client. It is limited. It can only browse the web, and there may be no other web-based applications installed on your computer other than that browser. Your PC depends on a server to give you access to the resources of the cloud. A cloud resource might be as simple as an online store or as complex as Google's docs application. In any case, the application does not reside on your PC. It resides on the server. If you are a member of that server, then you can have access to the applications stored on that machine.

Servers and clients communicate with each other by sending a **packet**. A packet is a data structure that looks much like a letter. It has a from-address and a to-address and the message (plus some error-checking data, which we will not talk about). When a client wants to make a request to a server, it creates a packet with the server's address and sends it out into the Internet. The Internet independently finds a route for the packet to the server. Since the packet has the sender's return address, the server knows where to return the reply. Both the client and the server depend on the routing capabilities of the Internet.

THE INTERNET AND INTER-PROCESS COMMUNICATION

What do you get when you mix networks and operating system shells? You get access to the Internet. The Internet gives the programmer a way to access the operating system shell over a network. If you have access to the shell, then you can invoke any shell command and run any program installed on that computer. What we need is software that gives us access to these shells. FTP, browsers, Internet programming languages, and remote login are the programs that run over the Internet network giving us access to the shell.

Remote login permits you to directly login to the server and see the shell. This assumes you have an account on that server. File Transfer Protocol, or FTP, is a program for copying files from your computer to the server or from the server to your computer. Browsers permit you to view the public portions of the Internet. If those public portions have HTML files, then you will see them by default. If the server does not have HTML files, then you will be given access to the shell, in a limited graphical form. Internet languages, like HTML, Java, CSS, CGI, Perl, C, and Python (you are not limited to these languages) allow you to create your own programs that can interface with the server and client.

Now is a good time to read the chapters on HTML, Perl, C, and Python. Assuming you already know these languages, Let's look at inter-process communication and the Internet.

CGI PROGRAMMING

Common Gateway Interface, or *CGI*, as the name implies, is a method by which different operating systems can communicate with one another in a standard way. Software written on a particular operating system will use the CGI language to formulate queries that are sent to another operating system's shell for processing. The result of the query is returned to the caller.

The Internet interconnects many different computers together into a single network. These computers are built by different manufacturers, use different operating systems, and are connected to different servers. Internet technology must be standardized so that communication across so many different platforms remains straightforward. CGI is the standardization of how a client-side machine can communicate with a server-side program, and back again. In this section, we will look at CGI from the client side. Later in this chapter, we will look at CGI from the server side.

CGI is a simple Internet sub-language. CGI is designed to exist imbedded within HTML code. A prerequisite to learning CGI is HTML (**Hyper Text Markup Language**). The idea behind CGI is the concept of a query. A query is a question you pose to someone. In our case, we are talking to operating systems. The only user interface is the shell, so our queries are expressed as a shell command. This is similar to the C system() function or the Unix command-line prompt. We want to do something similar from an HTML page. Since we are trying to access the shell, it would also be good to have something like the setenv() function. This would permit us to pass information to the shell.

CGI uses the idea of a form as the mechanism for expressing a query. The form is a simple idea. Ask the user to input infomation using a formatted form, and attach an operating system shell command to that form. Then send that command and the information the user provided to an operating system. The receiving operating system will respond by invoking a new shell, inserting the provided information into the shell's memory and executing the command at the command line. Ideally, the command invokes a program that uses the information stored in the shell's memory. Any output from that program is automatically transmitted back to the caller and viewed on the caller's browser. This is CGI in a nutshell.

Syntactically, we will describe CGI through a series of examples.

```
<form name="input" action="www.place.com/command" method="get">
    Username: <input type="text" name="user">
    <p>The username is case sensative.</p>
    <input type="submit" value="Submit">
</form>
```

The aforementioned example shows the fundamental syntax and usage of a form. Like all HTML tags, the form begins with <form> and ends with </form>. Between these tags, the programmer can write any HTML and CGI code. Notice that the above example mixes HTML tags (the <p> tag) and CGI tags (the <input> tag). CGI is a sub-language modeled on forms; therefore, all the CGI tags are related to forms. The tags fall into three categories: data input tags, button tags, and the operating system command-line tag.

The tag <form> has two purposes. The first purpose it to designate a portion of your HTML page as a form. The second purpose it to provide the place where you will write the command-line command and specify how the form data will be transmitted to the shell's memory on the receiving operating system. In the example above, we see that the <form> tag has three arguments: name, action, and method. The argument name is there to distinguish one from among multiple forms on a single HTML page. The action argument is the place where you will write your operating system command. The argument method has two options: **get** and **post**. They specify the way to transmit and store the input data in the shell.

Specifying `method="get"` concatenates all the form data to the end of the command-line query. The command-line query is found in the 'action' attribute, `action="command"`. There is a side effect to this concatenation. When you submit the form, the command-line query is displayed in the browser's URL box. This is the box at the top of your browser where you type in the URL of the web page you want to see. Since the method get adds the form data with the command-line query, the form data is also displayed publicly. This, for example, would not be good if, in the form, you asked a user to input his username and password. The format of this augmented command-line query, given our program above, would look like this:

```
http://www.place.com/command?user=bob
```

Notice that:

1. The http:// must be added to the front of the query since it will be using the URL way of finding the server. You specify the web path as any URL. If the http:// is not specified, the client will assume the query is for itself. It will default to looking in the current or default directory.
2. The `command/` is the operating system statement that will be sent to the shell's command-line. Normally, this is the name of an executable program or executable script written in C, Perl, Python, PHP, Bash, or any of the other server-side languages. A **server-side language** is any language that creates a program that runs on the server. HTML, Java, and JavaScript are examples of languages that are downloaded to the user's browser and executed on the user's computer and not the server. They are known as **client-side languages**.
3. The question mark separates the command-line command from the data the user inputted in the form. This data will be inserted into the destination shell using either the post or get method.

4. The data input from the form is written in the same way as it would appear in the shell's memory: `variable=value`. We have seen this before. In the CGI case, the variable is the form tag's name and the value is what the user input. This is expressed in the form as follows: `<input type="text" name="user">`. In other words, 'variable' is 'user' and 'value' is what the user input (this does not show up in the command, of course).

Specifying `method="post"` adds the form data within the query's packet. There is a field called data, in the packet, which normally stores the information being sent to the server. This is in contrast to the packet's destination field that simply stores the URL address. Specifying `method="post"` puts the form's input information into this data field. The benefit is that this information is not displayed publicly on the browser's URL box. But, on the down side, the information is stored in text mode. This means that if you asked the user to input her username and password, the information would be stored in the packet's data field unencrypted. We will not discuss encryption in detail within this textbook. If someone malicious has a program that can read packets, then all the information in the data field would be readable since it is not encrypted. Security is a concern because of this situation. The simplest solution is to encrypt the information to make it impossible to read.

CGI implements two special buttons to facilitate the management of the form. One button is called **submit**, and this causes the form to be transmitted to the destination server. The other button is called **reset**, and it re-displays the form on the user's browser without sending it to the server, but all the input fields are cleared. Our above example shows the submit tag: `<input type="submit" value="Submit">`. Notice that it is a very simple tag. The tag is called input. You can specify what kind of button it is using the type argument: submit or reset. In this case, it is set to submit. Then you can specify what the button label should say. In our example, is just says 'Submit'. More than one button can be added to a form.

The rest of the CGI tags are data input tags.

`<input>`	An input field	`<input type="text" name="User" value="Jack" />` Type = the format of the input field Name = variable name in shell's memory Value = default value if none provided by user

`<textarea>`	A multi-line text input field	`<textarea rows="10" cols="20">`
		`Once upon a time there lived a little girl named Goldilocks.`
		`</textarea>`

`<select>`	A drop-down selection field	`<select size=5 multiple>`
		`<option value ="car">Automobile</option>`
		`<option value ="bus">City Bus</option>`
		`<option value ="truck">Truck</option>`
		`<option value ="airplane">Airplane</option>`
		`</select>`
		Multiple, when present, indicates that multiple selection can be made.
		Size indicates the number of items displayed in the drop down.

<option>	The items in a <select>	
<optgroup>	An option group	<select> <optgroup label="Public Transportation"> <option value ="bus">Bus</option> <option value ="metro">Metro</option> </optgroup> <optgroup label="Private Transportation"> <option value ="car">Car</option> <option value ="bike">Bike</option> </optgroup> </select> A subgroup title is displayed in the drop down.
<button>	A push button	<button type="button">Click Me!</button> <button><img src="wow.gif"></img></button> Any HTML code can be put within the button tags. The TYPE attribute can be set to "submit" or "reset" and behave like the <input> submit or reset button but maybe prettier.
<fieldset>	Draw a box around controls	<fieldset> Height <input type="text" size="3" /> Weight <input type="text" size="3" /> </fieldset>
<legend>	A caption for <fieldset>	<fieldset> <legend>Registration:</legend> Height <input type="text" size="3" /> Weight <input type="text" size="3" /> </fieldset>
<label>	A label for a control	<input type="radio" name="sex" id="male" /> <label for="male">Male</label> <input type="radio" name="sex" id="female" /> <label for="female">Female</label> This displays a name beside the radio button that is also clickable.

An attribute pair is an argumen t and value pair th at modifies the beh avior of the tag. The <input> tag has many optional `attribute="value"` pairs. We list some of them here:

Type—button, checkbox, file, hidden, image, password, radio, reset, submit, text.

Align—left, right, top, bottom, middle, texttop, absmiddle, absbottom, baseline.

Maxlength—a number of characters.

Size—the size of input box.

Src—the URL of an image to display.

The following are attributes without a value. They appear by themselves without an equal or a value:

Checked—usedwith type=checkbox sets the control to true (it is checked).

Disabled—grays the control and cannot be used.

Readonly—The user cannot change the data stored in the control.

Below is a complete example of an HTML login page th at uses CGI to in terface with a server called www.kenwo.ca. The server will invoke the program login.cgi. Since the form uses the post method of sending data, the login information is not displayed on the browser's screen. In addition, the <input> tag used to enter the password is set to `type="password"`. This displays dots instead of characters when you type the password. This makes the password login almost secure. The last thing to do would be to encrypt the password.

```
<html>
  <head>
    <title>Sample CGI</title>
  </head>
  <body>
    <h1>Main Login Page</h1>
    <form action="http://www.kenwo/login.cgi" method="post">
      Username: <input type="text" name="user" maxlength="10" /> <br>
      Password: <input type="password" name="pass" maxlength="20" /> <br>
      User Type: <br>
        <select name="usertype">
          <option value="member"> Member
          <option value="visitor"> Visitor
          <option value="new"> New User
        </select>
        <br>
        <input type="submit" value="Login" />
    </form>
  </body>
</html>
```

THE OS SHELL AND CGI

There is a minor pr oblem between br owsers, HTTP, and oper ating system shells. Shells and HTTP function under a limited version of AS CII. Browsers often function under E xtended

ASCII or even UNICODE. This means that additional character encodings are not available in the shell or HTTP.

HTTP rules dictate that a destination URL cannot have any spaces or special characters, other than letters, digits, and some special characters like the colon, forward slash, backslash, ampersand, equals, etc. The command-line command and the form's data are all concatenated into a long string. Since the concatenated string is used as a URL, all offending input must be corrected.

Using the login CGI example from above but changing the transmission method to `method="get"`. The destination URL will look like this:

```
http://www.kenwo.ca/login.cgi?user=mary+lou%32&pass=happyday&usertype=visitor
```

The above URL assumes someone input "mary loué" for the username and "happyday" for the password, and selected "visitor" from the drop-down box. Looking at this URL carefully, we notice some additional symbols. We expect to see the URL http://www.kenwo.ca/login.cgi. We expect to see the question mark dividing the URL from the form's data. We also expect to see the form's data formatted in shell memory syntax, `variable="value"`. What is new are the ampersand, plus, and percent symbols.

Since spaces are not permitted in a URL, the browser replaces the spaces in the user's input data with the plus symbol. Hence, the username has a plus symbol. Each `variable="value"` pair is separated by the ampersand. We see two ampersands, one between the username and password, and the other between the password and user type. Any special characters are converted to their ASCII codes. The username has an é. Special characters are represented by the escape character % followed by its two-digit hexadecimal ASCII code.

When the operating system receives this query in either get or post modes, the variables are inserted into the shell's memory. The operating system does not do any conversions on the received data. It simply adds the information into the shell's memory as is. This means that the program launched from the command-line must take care of any conversions. In some programming languages, libraries are available to do this automatically for the user. In some cases, the programmers need to do it themselves.

Below are three examples of accessing the CGI interface with C. All these examples show how we can access CGI with little or no library help. This should give you a deeper understanding of CGI.

CGI AND C

We will first assume that the form issues a simple query without special symbols. We will look at one example that uses *get* and then we will see another example that uses *put*. After that, we will see an example that assumes a complex query having special characters.

First, a simple C interface with get:

```
#include <stdlib.h>
 char *string = getenv("QUERY_STRING");
 sscanf(string,"x=%d&y=%d",&a,&b);
```

The above example demonstrates that the get method posts the quer y string into the shell's memory in a variable called QUERY_STRING. This variable contains, as a string, the portion of the query after the question mark but not including the question mark. The example assumes you already know the format of the quer y, that the query sent two variables called x and y, and one integer number was inputted for each var iable. The #include<stdio.h> defines the signature for the sscanf function. This function reads from a string instead of the keyboar d. In this example, it reads from a string that was assigned the information from the shell's variable QUERY_STRING. It expects the data to be formatted like this: `"x=%d&y=%d"`. The variables a and b r eceive the data input by the user. This nice way of processing with sscanf does not always work. The format string `"x=%d&y=%d"` is very sensitive to differences in the input string and will fail at the smallest difference.

Our second simple C interface is with 'post':

```
#include <stdlib.h>
char string[200];
char c;
int a = 0;
int n = atoi(getenv("CONTENT_LENGTH"));
while ((c = getchar()) != EOF && a<n)
{
     if (a < 200) string[a]=c;
     a++;
}
String[a] = '0';
```

The communication technique based on post does not put the quer y into the shell's memory, but instead, sends it to stdin. The program must then read it in as if it were reading it from the keyboard. What is put in to the shell's memory is a var iable called CONTENT_LENGTH. This variable records the number of characters sent to stdin. This is important because stdin is not terminated by any special ch aracters like EOL, EOF, or \0. The above example uses getch ar() to read one ch aracter at a time un til n ch aracters have been pr ocessed. Each ch aracter is copied into the ar ray string[ ]. What will this ar ray contain? The data will be in the standar d

CGI `variable="value"` format such that spaces are replaced by the plus symbol, var iables are separated by the ampersand, and special characters are in hexadecimal value preceded by the percent symbol.

The last C example assumes a complex query string:

```
int n = atoi(getenv("CONTENT_LENGTH"));
fgets(inputArray, n+1, stdin);
unencode(inputArray,inputArray+n,outputArray);
:
void unencode (char *src, char *end, char *dest)
{
   for(; src != end; src++, dest++)
     if (*src == '+') *dest = ' ';
     else if (*src == '%')
       int code;
       if (sscanf(src+1,"%2x", &code)!=1) code = '?';
       *dest = code;
       src += 2;
      else *dest = *src;

   *dest = '';
   *++dest = '0';
}
```

In this example, we see two pieces of code. The first part comes above the colon and reads n+1 characters from stdin and stores those characters in an array called inputArray. It then calls the second part of the example. The function unencode's purpose is to convert the input array into an output ar ray that has b een converted back to its or iginal format, in other w ords, without special symbols. The plus symbol is returned back to a space, and the special percentage codes have been converted back to their original ASCII value. The ampersands and equal signs have been left since they are still useful.

Let's look at unencode carefully. First, we will look at how it is invoked. It takes three arguments. The first and last arguments are straightforward; they are both arrays. The first argument is the source array that will be tr ansformed into the third argument, the output ar ray. The middle argument is a bit strange. You need to recall that C arrays are actually pointers to arrays. So, the expression inputArray+n is referencing the array cell that contains the last inputted character.

Notice that the function's parameters src, end and dest m ust match the calling argumen ts. This, too, at first looks strange because the calling arguments are arrays, but the receiving function's parameters are pointers. This is not an error since arrays in C are implemented as pointers. The advantage of having the function's parameters defined as pointers is in the flexibility it will

give us to move about in memory. We can guess that *src points to the beginning of the input string, *end points to the end of the input string, and *dest points to an empty destination array that will receive the converted version of *src.

The function is driven by a for-loop that assumes src, end, and dest have already been initialized. The loop then iterates until src points to end. The for-loop iterates through every character of the input array, copying each character into the destination array unless the character is deemed special. Three if-statements determine the specialness of the character. The first if-statement flags the plus symbol and instead of copying the plus, it sends a space to the destination. The second if-statement looks for the percentage symbol. Notice that the if-statements do not look for any other symbols. This means that all other characters are left unchanged. The third if-statement has a nested sscanf function. The sscanf starts reading from src+1, one character beyond the percentage. It then reads two hex digits into the local variable code. The function sscanf returns a count of all the values it read in successfully. Since we are reading only one two-digit hex number, good read would return the integer 1. If we see any other number—most likely a zero—this means that something went wrong. If the sscanf returns a 1, then we know that code contains a good ASCII value and we simple send the contents of code to the destination array. If sscanf did not return a 1, then we do not know what the value was and we flag it by sending a question mark to destination. The function terminates by ending the destination array with a carriage return and the end-of-string symbol, '\0'.

With all this information, you should be able to write an HTML form that calls a C program on a server.

The last thing you need to know is how to store your files on the server. There are only two things to know. You first need to ask the system operator for the name of the public Internet directory. Every user's account on a server is private, but there is a single directory for each user account that can be connected to the Internet. On most operating systems, this directory is called public_html. It is commonly placed in your account's root directory. In this directory, you will put all your Internet files. This includes HTML, CSS, C, Perl, shell script files, and Python files, plus any other databases or programming languages.

The last thing you need to do is make sure that the public_html directory and all the files you have placed in that directory have public permissions. Without the public permission, they are still not accessible by the Internet. In Unix, you can use the chmod command to make your directories and files public, like this:

```
$ chmod rx+a filename
```

The arguments r and x refer to read and execute. The plus symbol indicates that we would like to turn this feature on. The argument a indicates that rx will be for all, or public. We do not want to do this for w. The argument w refers to write access. Making your file write accessible means that anyone on the Internet can make changes to your file. You may not want that.

To access your file use this URL:

```
http://website/youraccount/file.html
```

For the website, you would write something like www.mysite.com. The youraccount is not needed if you own the website name. Otherwise, you need to redirect the website to your account. This is done in many ways, but a common way is with /~username. You do not need to specify the public_html directory in your URL; it is assumed. The default web page in your directory should be called either index.htm or default.htm, depending on your operating system. Unix often uses default.htm, but Windows will often employ index.html. These days, many operating systems will accept both default.htm and index.html.

Client-Side Programming Languages

Client-side programming languages refer to programs that are downloaded from the server to the client's computer and executed on the client's computer. Normally, this occurs through browsers. Three languages are key to our discussions: HTML, CSS, and CGI. We have seen CGI already a little. The next few sections will give a good introduction to these three scripting languages. This will not be a complete description of the languages. A complete description would include covering every switch and option of every command. We will not look at very command, but only the most popular or useful switches and arguments. The Instant chapters cover some additional client-side languages.

HYPER TEXT MARKUP LANGUAGE (HTML)

The Hyper Text Markup Language is true to its name. It is not a programming language. It is a text-formatting language. It formats text in two ways. First, it behaves like a simple word processor by allowing the programmer to, for example, underline, bold, and indent text. Second, it allows text to be identified as **hyper**. Hyper refers to connecting text to an action that can be initiated through the click of a mouse. For example, text can be identified to be a **hyperlink** or as an **on-event**. A hyperlink has been common in web pages since the beginning when Apple Inc. invented the concept of hyperlinks, but independently, not in relation to web sites. You can identify a hyperlink when you look at a web page. Hyperlinked text is underlined and in blue. This color scheme can be changed, but the default blue text with blue underline is most common. When the mouse hovers over the link, most browsers will show, at the bottom left-hand side of the window, the URL associated with the hyperlink. If the user clicks on the text, the URL associated with the hyperlink will be launched by the browser. The current web page will be replaced by this new web page. This permits web pages to be cross referenced in a most natural way. The on-event is newer and is actually related to DHTML. Notice that we have a chapter on DHTML in the Instant chapters. Similar to hyperlinks, the on-event is an association between some text on the HTML page and a Java Script program. If the user clicks on the

text of an on-event, the Java Script program it is associated with is invoked. In some cases, you do not even need to click on the link; hovering over it is enough to invoke the Java Script. There is an Instant chapter covering Java Script as well.

THE HTML DOCUMENT AND SYNTAX

An HTML document supports multiple languages. It is common to see HTML , CSS, CGI, and Java Script in a single HTML file. It could also include PHP and J ava Applets. Each language is clearly identified with its own **tags**. A tag is a special HTML iden tifier. Tags are used to clearly delineate the different languages th at may inhabit the same HTML file, but HTML also uses tags as HTML comm ands. As we h ave said, the only HTML comm ands are text-formatting commands. There are no pr ogramming commands like if -statements, loop-statements, and functions in HTML. Everything in HTML uses the tag. There is no other str ucture in HTML. Here is the syntax:

```
<tag attribute_list> TEXT GOES HERE </tag>
```

Where:

- tag is the name of a command—always in lowercase.
- <tag> is understood to mean that the command begins from this point.
- </tag> is understood to mean that the command ends at this point.
- The TEXT GOES HERE is what will be affected by the command. Tags can be nested so the text can also include nested tags.
- The attribute_list is optional, but if present, consists of either a list of arguments and/or a list of argumen ts with values. The attribute_list is always in lowercase. An argument value combination is for matted this way: `argument="value"`. The value is alwa ys in double quotes.
- Note that older browsers permitted uppercase for tags and argumen ts. Older browsers also permitted values to be unquoted. This older style is no w replaced, but can still be found on some web sites.

HTML is very easy to use. You only need a common text editor , like Edit, Notepad, or Vi. The source file is permitted to have any file name, but it must end with either the extension .h tml or .htm, depending on your operating system. Most operating systems these days accept both. The HTML portion of the file is clearly delineated using the tag pair: <html> and </html>. You read them as begin and end. The <html> is the begin, and the </h tml> is the end. All HTML code exists between these two tags. The browser looks for these tags to iden tify how it should interpret the tags it comes acr oss. The browser's default mode is to assume ever ything is in

HTML mode, but every browser manufacturer has its own default assumptions. This makes formatting web pages interesting.

An HTML document is divided in to two sections the **header** and the **body**. These sections have their corresponding identification tags. The header sections uses <head> and </head>. The body section uses <body> and </body>. The header section's purpose is to define the resources the HTML page is using and to spe cify to the br owser which HTML standar d to adhere to when accessing this web site. There are many HTML standards: 1, 2, 3, 4, and now 5. Plus, there is the strict syntax-checking mode and the forgiving-syntax mode. If you do not specify anything, then the br owser runs in the most up-to-date HTML version the br owser supports but in the forgiving- syntax mode. The body section is wh at will be displayed to the user. This is the actual web page . It includes the text, graphics, hyper commands, and the other languages like C SS, CGI, Java Applets, and PHP. JavaScript can also be pr esent, but often, it is defined in its own section.

The best way to learn HTML is by example, so please look at the following example:

```
<html>
      <head>
            <title>My First Web Page</title>
      </head>

      <body bgcolor="gray">

            <h1>Welcome</h1>

            This is
            my first web page!
            Yay!<br />

      </body>
</html>
```

This example demonstrates some very important things to know about the way HTML operates on browsers. Let me list them:

- Let's first notice that HTML is formatted in a similar way that programming languages are formatted. Indentation is used to indicate that the indented parts belong within the non-indented part. In the above example, everything is indented after the <h tml> tag pair. Notice the <title> tag pair is inden ted under the <head> tag pair . Notice that the <body> tag pair is a t the same inden tation level as the <head> tag pair . This indicates that they are independent from each other. In summary, everything in this example is an HTML document. The <title> tag pair causes some change to the run-time environment. The <h1> tag pair w ith the text follo wed by the single
 tag is the actual web page that will display on the browser.

- To continue with formatting code, notice that blank lines are used to help highlight different sections of the code. The <head> tag pair has a blank line separating it from the <body> tag pair. This is also true concerning the <h1> tag pair and the rest of the text in the body section. It is good practice to write code this way since large programs can get very hard to read.
- The indentation and blank line spacing is completely optional in HTML programming. I suggest you do it so that code is easier to read.
- Browsers display web pages in a default mode. This means that if you do not specify exactly how you want your web page to display on the screen, the browser, if left to its own devices, will display your web page in its default mode. The browser's default mode is the following:
  1. Ignore all white space characters in the HTML document, except for the very first blank space after every word. This rule is strictly followed. The blank space following the word can only be the space-bar character; all other white space characters (carriage return, line feed, tab, multiple space-bar characters) will be ignored.
  2. Everything appears left to right, one after the other, on the web page until it wraps around the window and continues in this manner until the end of the web page.

This is very ugly. In our example, the <h1> tag pair has a strict formatting definition, which we will talk about soon, but the text—"This is my first web page! Yay!"—will appear horizontally on the screen as a single line of text, as in this paragraph. The two carriage returns and the tabbing will not affect how it will be displayed on the web page. On the web page, this text will appear starting at the left margin and progress horizontally character by character until the single
 tag, which we will describe soon. The text is unadorned; it is displayed in the default browser font and color. It is not underlined or bold, or anything.

- The tag pair <title> in the header section is commonly used in all web sites. The text defined within this tag is displayed in the browser's window frame. This is the place where the name of the web site you are visiting is displayed. You define that name with this tag pair.
- The tag pair <h1> is a member of a family of tag pairs <h1>, <h2>, <h3>, <h4>, <h5>, and <h6>. These tags are called title tags or heading tags (not to be confused with the <title> tag pair or the <head> tag pair). They display the text between the tags on its own row of the web page in a particular point size. The <h1> tag is the largest point size. The <h6> tag is the smallest point size. The other tags descend from <h2> to <h5>. The point size is not always the same from browser to browser. This is a simple and quick way of creating a heading in your web page.

- HTML allows a shorthand syntax for some of its tags . The carriage return tag is called break. In its long form, it is a tag pair : `<br> ... </br>`. In its short form, it can be written: `<br />`. Our example shows the shorthand form. It indicates that the browser will display a carriage return after the "Yay!"

Enter the above web page in a text editor , and save it anywhere in your computer. Then, start your browser. In the menu, select File/Open and find your document. Open it, and find out how it looks.

This is all you need to know about the HTML document. There is nothing more, other than the commands themselves. Before we look at the commands, Let's talk about the web site itself a little.

A WEB SITE

A web site is defined to be a directory (folder) on a server (or even client PC) with permissions set to public and readable. The server itself needs to be connected to the Internet and an owner of an IP address. The IP address is the mailing address of the server on the Internet. It is how the Internet identifies the server. The server has a user's list. Each user is permitted to have at least one public directory connected to the internet. Often, the operating system requires you to use a particular directory name. In Unix, this is the directory name public_html in all lowercase characters. The operating system assumes the public directory has a default web page called index.html or default.htm. Many operating systems accept either. Note that the users themselves each have their own public_html directory, but the server itself has its own official public_html directory. Normally, the server's public directory is the corporate web site. Users are often permitted to have their own web sites within the corporate machine. To address these web sites, we follow these rules:

- To get to the corporate web site, you enter the URL of the IP address. In other words, something like: `http://www.corporation.com`.
- Any sub-directories bellow the corporate public_html directory can be referenced this way: `http://www.corporate.com/subdirectory`. If there was a sub- sub-directory, then we would enter `http://www.corporate.com/subdirectory/subsubdirectory`, and so on.
- Users whom have created their own web pages have two options: either they use the corporate URL or the purchase their own. If they purchase their own URL, then their web site will function as described for the corporate web site. If they use the corporate web site name, then they would enter the following: `http://www.corporate.com/~username`. This would send the random Internet user to username's web site.

Building your own web site is then easy to do . Your first create the public_html directory and make it public. You put an index.html page in it. This index.html page will hyperlink to all the other pages in your web site. The index.html page becomes your home page. All the files need to be set as public and readable. It is not enough to just put them in the public_html folder. The last thing to do is to tell the server, or server operator, to connect your public_html folder to the Internet. You are now online!

HTML COMMANDS

In this section, all the common HTML tags will be presented divided in sections with mini examples. The end of this section shows a larger example web page . All HTML tags are nestable. All tags appear on the web page in the exact spot the HTML tag was placed in the HTML source code.

Text Formatting

Bold	BOLD TEXT HERE
Underline	<u> UNDERLINED TEXT HERE </u>
Carriage return	
Center	NESTED TEXT HERE
Headings	< h1> HEADING </h1> also <h2> <h3> <h4> <h5> <h6>
Font	 NESTED
Paragraph	ENTIRE PARAGRAPH NESTABLE

Bullets

These bullet lists are nestable and the bullet symbol changes in each nest. This is the standard bullet list seen from word processors.

```
Bullet List          <ul>
                <li>First list item</li>
                <li>Second list item</li>
                     </ul>
Numbered list        <ol>
                <li>First list item</li>
                <li>Second list item</li>
                     </ol>
```

Pictures

Display image	

Hyperlinks

Hyperlink	Underlined text

Colors Attributes

There is no color tag, but many tags have a color attribute: <body>, <font>, <table>, for example. Colors in HTML can be specified in three ways: default color name identifiers, the HEX notation, or the RGB notation. In all cases, the attribute has the following syntax: `color="VALUE"`. The quoted value is in any of the three notation formats. The named color identifiers are the easiest to use, but they are limited to 16 colors. The HEX and RGB notations can specify all the colors supported by the browser.

ColorNamed Identifiers:

black, white, red, green, blue, yellow, lightblue, magenta, gray, lightgray.

HEX Notation:

The HEX notation syntax is: #digits

Where digits is exactly 6 hexadecimal digits

For example: "#000000" is black.

The first 2 hex digits is red quantity, the next 2 is green quantity, and the last is blue quantity.

RGB Notation:

The RGB notation syntax is: rgb(RED, GREEN, BLUE)

Where RED, GREEN, BLUE is the amount of color from 0 to 255

For example: "rgb(0,0,0)" is black.

Check online for complete color lists. A good site is:
http://www.w3schools.com/html/html_colors.asp

Tables

Web page tables are useful tools to organize and display data, but also to help format the web page. Normally, a table is used to display numerical information as in a spreadsheet. But HTML table borders can be hidden, and the table's cells can be nested with any volume of information or tags. This means that we can use a table to create, for example, a menu column on the

left, a large centered title at the top of the page, and rows of information or just one big cell of information containing the bulk of the web page Each table size, row size, and column size can be specified as to its width. The syntax is as follows:

```
<table width="in_pixels" bgcolor="background" color="border"
     border="thickness">
     <tr width="in_pixles" bgcolor="background">
          <td> nested info & tags </td>
          <td> nested info & tags </td>
          <td> nested info & tags </td>
     </tr>
     <tr width="in_pixles" bgcolor="background">
          <td> nested info & tags </td>
          <td colspan="2"> nested info & tags </td>
     </tr>
</table>
```

The <table> tag pair defines the scope of the table structure. It has the attributes width, bgcolor, color, and border. The width attribute defines the maximum width of the table. All rows and cells must add up to this maximum. Two colors can be specified: for the table borders and for the cell backgrounds. Lastly, the thickness of the table border lines can be specified. A thickness of zero hides them. A thickness size of 1 is the default.

The <tr> tag pair defines a single row of the table. In the above example, there are two rows. Any number of rows can exist, even no rows at all. Other attributes not shown in the example are: horizontal align (align="VALUE", right, left, center, justify), vertical alignment (valign="VALUE", top, middle, bottom).

The <td> tag pair defines a cell of the table , called table da ta unit. The first row of the table has three cells, while the second r ow of the table h as only two cells. The rows do not need to agree. Other attributes not sho wn in the example are: align and valign (as in <tr>), column span (colspan="VALUE", where value is the n umber of columns the cell should span), w idth (as seen before), and height (just like width), and row spanning (rowspan="VALUE", just like column span but for rows). The attributes bgcolor, height, and width are deprecated and may be removed from future versions of browsers.

Special Characters

HTML supports only standard ASCII characters. For example, the French é cannot be used. In addition, some of the ASCII characters are used in HTML syntax, restricting the general use of these characters. For example, the greater-than and less-than signs cannot be used beca use they are part of the syntax of HTML tags.

To overcome this issue, HTML provides other ways to specify special characters through the use of HTML Symbol Entities and HTML Entity Codes. In either case, the syntax is as follows:

```
&#CODE;
```

Where:

- **&#** specifies that beginning of an Entity
- **CODE** is the value you will need to supply
  - The CODE is formatted differently for HTML Symbol Entities and HTML Entity Codes.
- **;** terminates the Entity

For HTML Entity Codes, the CODE is the ASCII number of the character you would like to print out. Therefore, if you want to print é, then you would need to write é in any text of the source HTML document to see the French character on the screen.

HTML Symbol Entities are special symbols that have been predefined. This set is not as complete as the ASCII code but has many symbols. Therefore, if you want to print é, then you would need to write &#eacute; in any text of the source HTML document to see the French character on the screen.

Use Google to find a complete list of these entities online.

Header Section

The <head> tag pair defines all the meta information about the web page. Tags that can be used in the header sections are: <base>, <link>, <meta>, <script>, <style>, and <title>. Syntax for the <head> tag is:

```
<head dir="DIR" lang="LANG", xml:lang="LANG"> … </head>
```

Where:

- DIR is either "rtl" or "ltr" and specifies the text direction.
- LANG sets the default language for the page (fr for French).
- xml:lang is the same as LANG but for XHTML documents.

We have seen <title> already.

The <base> tag specifies the default web site and target for all hyperlinks. For example: <base href="http://default.website.com" /> and <base target="_blank " /> The <base href> defines the default http for all links on the web page . The <base target> tag defines how the page will

be opened. There are a few target possibilities: "_blank" new blank page, "_parent" overwrite the parent's page, "_self" overwrite its own page (default), "framename" overwrite by directly specifying a page, if it has a name.

The <link> tag is like the #include directive in C. It includes specific types of files into your web page. For example:

```
<link rel="stylesheet" type="text/css" href="style.css" />
```

The attribute rel defines the relationship the include file has with the HTML file. In this case, it indicates a CSS style sheet. The attribute type indicates the contents of the include file. In this case, it is a text file with CSS instructions. The href attribute is the URL of the include file.

The <meta> tag pr ovides a way to define global infor mation about the web page . There are three basic forms; two are described here. They have the following syntax:

Form 1

```
<meta name="IDENTIFIER" content="VALUE" />
```

Where:

- IDENTIFIER is a designated area that can be given a description.
  - The following areas can be specified: "author," "description," "keywords," "generator," "revised."
- VALUE is the description for the IDENTIFIER.

Form 2

```
<meta http-equiv="IDENTIFIER" content="VALUE" />
```

Where:

- IDENTIFIER is a designed area that can be given a definition.
  - The following areas can be specified: "content-type," "content-style-type," "expires," "set-cookie."
- VALUE is the value for the IDENTIFIER. Each value has meaning.

The <script> tag is used with JavaScript. Read the Instant chapter for more information.

The <style> tag is used with CSS. Read the CSS section of this chapter for more information.

Additional Web References

A very good web site to learn about web languages is www.w3schools.com.

CASCADING STYLE SHEETS (CSS)

Many HTML tags and attributes have been deprecated since the inception of **Cascading Style Sheets**. The largest complaint leveled against HTML was in the control it gave to the web page designer when formatting and positioning the elements of the web page. With the introduction of CSS, many of these issues no longer exist. CSS is a comprehensive set of document-formatting instructions. CSS instructions can be inserted directly into an HTML tag, globally in a single HTML document, or externally as a CSS text file with file extension .css that can affect all the web pages in a web site. HTML has changed for the better.

CSS is very easy to use. There are only two things to know about: (1) the CSS general formatting syntax, and (2) the source file insertion rules.

CSS General Formatting Syntax

```
SELECTOR { DECLARATION; DECLARATION; … }
```

Where:

- SELECTOR is any HTML tag name, like body, h1, or p.
- DECLARATION is a list of semicolon-separated formatting statements.
  - Each declaration's syntax is: PROPERTY:VALUE.
  - The PROPERTY is the format change you want to make.
  - The VALUE specifies the change.

For example:

```
body {color:red; font-size:10px; text-align:center;}
```

The above example demonstrates a CSS instruction that modifies the <body> tag. The <body> tag pair defines the section of the HTML document that contains the information that is displayed on the web page. This CSS instruction modifies how all the information will be displayed. It states that all the text will be printed out on the screen in red at 10-point size and centered.

To format a web page, you simple write a list of these CSS instructions. All of them are applied to the web page at the same time. If some conflict with each other, then the last CSS instruction issued (in the conflict set) will dominate.

The CSS instructions can be inserted in the following areas: imbedded within an HTML tag using the attribute `style="CSS DECLARATION LIST"`; globally in the HTML document when added to the header section of the page using the <style> tag; and globally throughout the web site using an external CSS text file with file extension .css.

These three insertion points have **precedence rules**: imbedded attributes take precedence, followed by the <style> tag in the header section, followed by the .css external file, followed by the specific HTML tag format, and finally, the browser defaults.

The best way to understand this is to look at some examples:

Imbedded CSS

```
<h1 style="color:blue; font-family:Arial; font-size:10px;">Welcome</h1>
```

The above example shows the <h1> tag modified to show the heading text in a different point size, color, and font than its normal default <h1> way.

The <style> Tag

```
<head>
      <style type="text/css">
            h1 {color:blue; font-family:Arial; font-size:10px}
            body {color:black; background-color:grey}
      </style>
</head>
```

The <style> tag can only exist within the <head> tag pair. The <style> tag specifies the attribute type that declares the kind of style sheet in use. At present, only text/css is defined. Everything between the <style> tag pair must be in the text/css format. The text/css format declares that the instructions will be written as ASCII text in CSS instruction format. All the CSS instructions are listed one after the other.

The .css File

```
<head>
      <link rel="stylesheet" type="text/css" href="style.css" />
</head>
```

First, the .css file must be included, using the <link> tag, in to an HTML document within the header section. The above example shows how to do this. The included style.css file would look something like this:

```
/* comments */
h1 {color:blue; font-family:Arial; font-size:10px}
body {color:black; background-color:grey}
```

Take note that the .css file is just a list of CSS statements. It looks much like the CSS instructions between the <style> tag pair. There is nothing else in a .css file other than a list of CSS instructions and comment statements.

Grouping CSS Styles

The general syntax for CSS allows for the grouping of HTML tags with a single style declaration. For example, this syntax is also permitted:

```
h1,h2,h3,h4,h5,h6
{
      color: green;
}
```

In the above example, all the tags <h1>, <h2>, <h3>, <h4>, <h5>, and <h6> have the same style definition: their font color is green.

CSS Alternative Syntax

A few alternative declarative forms in CSS are very useful. They include **tagged class** declarations and **global class** declarations. A class declaration allows multiple different CSS declarations for an HTML tag. We have seen how we can modify and control the visual representation of an HTML tag, for example: `h1 {color:blue; font-family:Arial; font-size:10px}` changes the <h1> tag drastically compared to its default form in HTML. The class declaration permits multiple declarations for a tag. If a web page needs two different formats for the <h1> tag, then using a class would be the way to do it.

Syntax:

```
TAG.CLASS { DECLARATION; DECLARATION; … }
```

Where:

- TAG — is any HTML tag.
- CIASS — is any identifier you would like to use. It is a user-defined name.
- DECLARATION; — is a semicolon-separated list of CSS statements.

This declaration behaves in the regular way: any specified TAG that requests for the CLASS version of the CSS rule will have all the DECLARATION statements applied to it. Notice how giving different CLASS identifiers allows for multiple versions of the TAG. The CLASS identifier can be any legal identifier. A legal identifier is a single word that starts with a letter and follows with a contiguous series of letters and/or digits.

For example:

```
h1 {color:black; font-family:"Times Roman"; font-size:10px}
h1.blue {color:blue; font-family:Arial; font-size:10px}
h1.red {color:red; font-family:Arial; font-size:17px}
```

The above example shows three versions of the <h1> tag. One version defines a header that is blue at 10-point size; the other is red at 17-point size; both are Arial. Note that we also have an <h1> rule that does not use a class. This non-class version of the rule is optional but serves as our default <h1> tag rule. In other words, regular CSS rules are automatically applied to the HTML tag; it is declared for by the browser. In the CLASS rule variation, the HTML tag must specify which class to use. If the HTML tag does not specify the class, then it will look for the standard CSS rule declared for that tag (if one exists).

Here is how the HTML tag specifies the CLASS it wants to use:

```
<h1 class="blue">Welcome!</h1>
```

In the above example, the <h1> tag uses the attribute name **class** to identify which <h1> class to use.

CSS allows the programmer to declare CSS rules that use CLASS identifiers but are not connected to an HTML tag. The rules become global class rules and can be applied to any HTML tag. Note that the TAG.CLASS version of the CSS rule is locked in with the TAG. It can only be used with that TAG. The tag-less version of the rule allows the declaration of rules that can be applied globally within the HTML document across any number of tags.

Syntax:

```
.CLASS { DECLARATION; DECLARATION; … }
```

Where:

- .CLASS is the class identifier name.
- DECLARATION is a semicolon-separated list of CSS statements.

Example:

```
h1 {color:black; font-family:"Times Roman"; font-size:10px}
h1.blue {color:blue; font-family:Arial; font-size:10px}
.red {color:red; font-family:Arial; font-size:17px}
```

We modified the previous example here. Notice we have all three CSS rule forms present: tag based, tag class based, and tag-less based. The <h1> CSS rule applies to all <h1> tags that have not requested a class association. Any <h1> tags requesting the blue class will have the properties of the blue class applied. The last CCS rule, red, is tag-less. This class identifier can be used in any HTML tag, for example: `<body class="red"> … </body>`.

Multiple Classes

One last note: an HTML tag can request the application of more than one class. This is done by listing all the class identifiers, space separated, in the HTML tag's class attribute expression.

For example:

```
<h1 class="blue red">Welcome!</h1>
```

Conflicting entries will be overridden by the last class specified in the space-separated list.

A CATALOG OF CSS STATEMENTS

A complete description of all CSS statements is beyond the scope of this book. A useful list of CSS statements will instead be presented. This list will be useful for many basic web development tasks.

CSS Property	Values
color	Colors can be expressed in three forms: browser defined color names: black, white, red, blue, etc. (limited), or Hexadecimal format: #0F0F0F or RGB format: rgb(0,0,255)
font-family	Fonts can be setup two ways: as a fixed declaration, or as a cascading declaration. A fixed declaration is defining a single font. If the computer supports the font, the font is displayed;otherwise, the browser default is displayed. Using a cascading declaration, multiple fonts are defined in comma-separated list. This first font is preferred, but if not supported, the next in the list is tried until the default browser font is used as last choice, e.g.: "Sans serif," Times, Courier, Sans-serif Notice names with spaces can be quoted.
font-size	Standard point size declaration.
font-style	Choices are: normal, italic, oblique.
text-align	Choices are: center, right, left, justify.
text-decoration	Choices are: underline, overline, line-through, blink.
text-transform	Choices are: capitalize, uppercase, lowercase.
margin-left	Choices are: auto, length (e.g., 20px), percentage (e.g., 10%).
margin-right	Choices are: auto, length (e.g., 20px), percentage (e.g., 10%).
margin-top	Choices are: auto, length (e.g., 20px), percentage (e.g., 10%).
margin-bottom	Choices are: auto, length (e.g., 20px), percentage (e.g., 10%).
	XX = bottom, top, left, right.
border-XX-color	Same as in color description, above.
border-XX-style	Choices are: solid, double, groove.
border-XX-width	Measured in pixels (e.g., 20px).
References: www.w3Schools.com provides a complete list of colors and properties.	

SERVER-SIDE COMMUNICATION

Two forms of Internet communication exist: **server-initiated communication** and **client-initiated communication**. Client-initiated communication is most familiar to Internet users. This form of communication is initiated strictly by the user. For example, the user opens a browser, the user enters a URL, and the user presses go to send a request to the Internet for that web page (or, in other words, the user initiates communication by sending a request for a web page to the Internet). Server-initiated communication occurs when the server sends information, on its own, to a destination computer. The destination computer could be another server or a client. The server initiates this request most commonly after it has completed some task, or at regular intervals using a timer. In other words, the server could send a message to your client PC every couple of minutes to verify that you are still there. Your PC would see this request (called a ping) and reply with a message (normally, its IP address and MAC address) indicating that it is still online.

Servers maintain a communication connection called a **session** between the client and themselves. There is one session for each client. A session is an important construct because it is the tool the server uses to **manage the request** initiated by the client. The session consists of the user's request, an operating system shell to house the request, the execution of the requested program within the shell, information about the client stored in the shell's memory, the redirection of stdin and stdout, and optional security using passwords. The stream stdin is attached to the information coming from the client. The stream stdout is attached to the output returned to the client. The session is also the tool the server uses to track the state of a client request. The **states of a request** are: received, shell initiated, request executed, results returned to client, and session terminated. For example, when a user submits an online form, the online form sends a request to a server to run a specific program to process the form's information, while it passes the form information to the server's shell memory or to stdin. When the program executes at the server, all the data from the online form is already in the shell's memory or in stdin available for access by the program. When the program prints using stdout, the results are displayed on the client's browser.

A session can be in two modes known as a **persistent session** or as a **one-time session**. A VPN would be an example of a persistent session. A request for a web page would be an example of a one-time session. A persistent session requires a log-in. Once the client is logged-in, the session is constructed (as described above), and stays present in the server's memory until the client logs out or a server idle-too-long timer automatically logs out the client. A one-time session does not require the client to log in and log out. The connection is assumed to be public. The server will accept requests from any source. The server does not track the client; it only tracks the request. The server receives the request and creates a temporary session for the request (as described above). The server processes the request returning the resulting information to the client. The session is then terminated, and all associated information is deleted (unless the server uses a log file to record all server activity).

Communication between the ser ver and the clien t is associa ted with the shell. The shell h as access to stdin, stdout, and the shell memory. The shell can also be password protected. The shell is unique to each user. The shell can also communicate with the operating system. Therefore, the shell becomes an ideal wa y to manage Internet communication with a client. CGI is the back - bone that connects HTML documents with the operating system shell. The operating system shell becomes the backbone that bridges the connection to the programs residing on the server. Since the shell can execute any program and operating system command, then your HTML document through CGI can have access to any program or operating system command on the server.

Figure 5.1 below demonstrates the basic Internet session. A web document initiates communication through the Common Gateway Interface. The CGI protocol talks to the browser requesting the creation of a packet. The browser together, with the local operating system, creates a packet containing the request and data. This is passed to the Internet, which routes the packet to the destination server. Assuming the server is accepting public requests, it will create a session for the receiving packet and then a shell. It will connect stdin to the packet's data to form an input stream. Other packet-related information is posted into the shell's memory. The shell then invokes the program requested by the CGI. The session also redirects stdout to an out - going packet that will be delivered to the client once the session is completed. The program executed by the shell sends its output to stdout , as if it was pr inting to the screen; but in this case, the output is saved within the returned packet. When the program finishes, it terminates normally. Since there is nothing left for the shell to do, it too terminates, which causes the session to end and the returning packet to be passed to the Internet for routing back to the client. The client's browser receives the packet and prints the contents of the packet to the browser's screen. Since the browser expects to print out HTML code, the receiving packet's text could be HTML. So, the server has the choice to simply send AS CII and have the message print on the browser's screen in the browser,s default writing mode, or the AS CII text could be for matted using HTML. In other words, the program could have created its own HTML document.

We have already seen how to write CGI code. The <form> tag gives you the ability to select the program you want to server to execute and in wh at form the session will be built. The action attribute specifies the URL to the program the shell will execute. The method attribute specifies

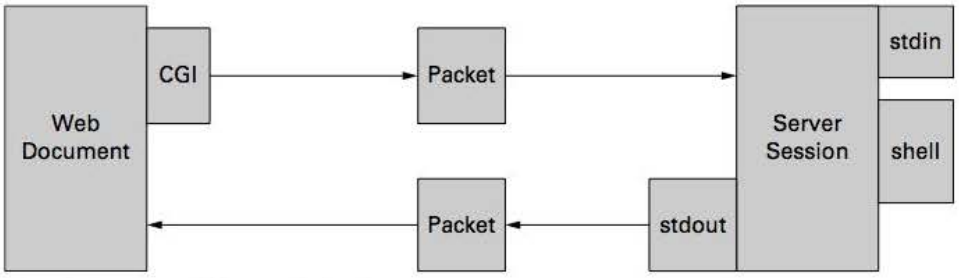


FIGURE 5.1: Internet Communication Session

the session type: get or post. In other words, it selects the dominant communication path from client to server. The path from server to client is always the same, stdout. The inbound communication path can be dominated whether by stdin or the shell's memory. We saw how to establish communication in C with either of these methods. We used C as the quintessential example of how the server handles Internet communication. The techniques we saw for C are common for any server-side language. Popular ones include: Bash, C, Python, Java, and Perl.

Let's see how this works using an example. Let's say a web document wants to communicate with a server in C. Let's remember how this looks like and then let's compare with other server-side languages like Bash, Python, Java, and Perl.

```
<html>
      <head>
            <title>Example Server Communication</title>
      </head>
      <body>
            <form action="http://www.abc.com/NoOfDays" method="get">
                  Age: <input name="myAge" type="text">
                  <input type="submit">
            </form>
</body>
</html>
```

The above HTML document shows a simple web page that uses CGI. The web page displays a single text box and submit button. The text box prompts the user for his or her age. The user enters his/her age and then presses the submit button. This causes a packet to be created with the variable name myAge and the information the user inputted. The packet is routed by the Internet to the destination URL http://www.abc.com. A "get" session is created with a shell that executes the program NoOfDays. This part is important: This HTML document does not depend on the language of the program NoOfDays. This means that this web document can be used for any server-side program. The only thing that changes is the name of the program. What also does not change is that stdout is redirected to a returning packet. This is true for all server-side programming languages. What also stays the same is how C communicated with the session by either extracting data from the shell or reading from stdin. This too stays the same regardless of the server-side programming language. What does change is the syntax of these different programming languages. These we will look at here, starting with C and the "get" methods. C is arguably the most difficult syntactically to interface with the session, but it is the fastest. Bash interfaces with the shell naturally, while Python and Perl have strong libraries to help the programmer.

C interfaces this way:

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

```
int main(void)
{
      char *string = getenv("QUERY_STRING");
      int theAge, theDays;

      sscanf(string,"myAge=%d",&theAge);

      theDays = theAge * 365; // without leap years

      printf("%s%c%c\n","Content-Type:text/html;
      charset= iso-8859-1",13,10);
      printf("<br />You age in days is <b>%d</b> <br />", theDays);
}
```

This above C program is simple enough. The getenv() function comes from the stdlib.h library. It is used to get the QUERY_STRING data that was inserted there by the shell from the inbound packet, since we used the "get" method. This information comes from the HTML document and is assumed to contain the variable myAge with the user's age, formatted like this: myAge=20 (for example). No data checking occurs; the sscanf() function will either convert the string to an integer or it will fail and return the number zero. The program computes the number of days, without leap years, and prints that out using the standard printf() function. Notice that the output is formatted as an HTML document. Notice that an initial printf() function is used to send a special coded string to the browser. This coded string prepares the browser to properly process the returning information. This is optional, but if used, tells the browser the format of the data returned by the server. The returned information could be simple ASCII or an HTML document. The above example prepares the browser for HTML-formatted data. Notice that the coded string returns four items: a string, two characters, and then a carriage return. The string can be formatted simply as this: `"Content-Type:text/html"`, meaning that the format of the content of the data is in text but formatted as HTML. Or, optionally, the coded string could be written as this: `"Content-Type:text/html;charset=iso-8859-1"`, meaning a text document formatted as HTML using the standard ISO character set 8859-1. After this string, the browser expects the coded sequence to terminate with two carriage returns formatted in different ways. This is accomplished, in C, with the standard \n character and the two %c characters 13 and 10, which is identical to the \n character. Many browsers these days do not need this change of format, and sometimes, you will find that simply printing two \n characters will suffice. Since browsers are not standard, I cannot be more clear than that.

What I would like you to notice is how we read-in and wrote-out in the above C example. You will notice that this pattern persists for all other programming languages.

Now for a Bash example:

```
#!/bin/sh
echo "Content-type: text/html\n\n"
# read in the parameter
```

```
theAge=`echo "$QUERY_STRING" | sed -n 's/^.*myAge=\([^&]*\).*$/\1/p'
| sed "s/%20/ /g"`
# Output
theDays=`expr $theAge * 365`
echo "<html>"
echo "<head><title>The Answer</title></head>"
echo "<body>"
echo "Your age in days is $theDays<br>"
echo "</body>"
echo "</html>"
```

As with C, the shell receives a new variable called QUERY_STRING that contains the data from the packet formatted as: var=value&var2=value2&..., for any number of variables. Bash does not use any libraries to interface with the shell since it already does that naturally. Notice that in line 2, it outputs the browser-coded string to inform the browser that HTML text will be returned. The Bash script then processes the input query string by linearly extracting each section of data from the beginning of the string to the end. It uses regular expressions and the sed program, which is similar to the grep program we already saw in the Unix chapter. The sed program can take redirected input and return a modified string based on the input string. In this case, the echo command redirects the $QUERY_STRING shell variable to the sed program that scans the string for the variable "^.*myAge=" which means the variable myAge starting for the beginning of the string ignoring any other characters proceeding it. The value is then extracted and cleaned up by replacing the %20 with spaces. There are many ways to format this cutting operation in Bash. One example is shown here. The script then calls the Unix program expr to calculate the age in days. The output is sent to stdout.

Here is our Python example:

```
#!/usr/local/bin/python
import cgi
def main():
  print "Content-type: text/html\n\n"
  form = cgi.FieldStorage() # parse query
  if form.has_key("myAge") and form["myAge"].value != "":
       theDays = form["myAge"] * 365
       print "<b>Your age in days is ", theDays, "</b>"
  else:
       print "<h1>Error! Please enter your age.</h1>"
main()
```

Python has two powerful things going for it: the library cgi and the dictionary data structure. I am assuming you understand Python if you are reading this book. If not, then review dictionaries from the Instant chapter on Python. Notice that the program includes the cgi library

at line 2. Then at line 4, it pr ints to stdout the br owser-coded string telling the br owser that the output text is formatted as HTML. Then the cgi program FieldStorage() is invoked to read the information that has come in fr om the packet. There is nothing else for the pr ogrammer to do. This function splits the input var iables from their values, does basic str ing-to-data conversion, and stores this all into the dictionary called form. The program ends by calculating the days in age and either pr inting that out or displaying an error message. The method FieldStorage() handles both "get" and "post" communication. The dictionary accepts the data from the inbound packet and stor es it in the follo wing way: assume the inbound packet contained "foo=ab+cd%21ef&bar=spam" the method FieldStorage() converts this to the following dictionary {'foo': 'ab cd!ef', 'bar': 'spam'}. Notice that some basic conversion has occurred automatically for us: the plus sign was changed back to a space, the %21 was converted back to an exclamation mark, the ampersands and equal signs wer e also removed, and the data split up as it should be in to variables and value pairs. These pairs are stored in the diction ary as they should be: variable names as the dictionary's index and the values as the index's value.

Now for our Perl example:

```
#!/usr/local/bin/perl -wT
use strinct;
ue CGI "standard";

my $var = param('myAge');
my $days = $var * 365;
print "Your age is days is $days";
```

The above example assumes th at you are familiar with Perl. If you ar e not, please refer to the Instant Perl chapter for a review. Like Python and Bash, Perl communicates with the shell in a natural manner and has a robust CGI library. The library is included with the Perl command `use CGI;` and can be optionally set in a `"standard"` mode. This library has a function called param() that scans the input packet 's data string for the var iable name provided in its argument: `$var = param('myAge');` . In other wor ds, param() scans for 'm yAge' and r eturns its value. The param() function performs automatic formatting changes and returns that converted value back to Perl. In the example, this returned value is stored in the variable `$var`. The rest of the program is similar to wh at we have already seen. The param() function can extract the entire set of variables, much like in Python, in the following way:

```
@var2 = param();
foreach my $name (var2)
{
     my $value = param($name);
}
```

The above example uses the function par am() without any argumen ts. This causes param() to return all the variables as a Perl hash data structure. This structure is akin to Python's dictionary. The example then uses the foreach-loop to iterate through all the value in the hash data structure.

CHAPTER SIX

Instant Python

Python was initially introduced by Guido van Rossum at the *National Research Institute for Mathematics and Computer Science* in Amsterdam (*CWI*) in the late 1980s/early 1990s. Rossum named the language after the comedian Monty Python. The language is easy to learn by a beginner programmer. It is portable, and it combines the power of many existing languages. It is mostly used in the creation of Shell tools, graphical user interfaces, and data structure libraries, and it is easily extensible.

For a machine to understand and execute Python code, it has to have the Python interpreter (which can be found at *www.python.org*). The details are independent of the operating system, except for the installation of the interpreter and the operating system commands.

There are two ways to write code in Python and execute it. The first way consists of interactively running the Python interpreter by typing the following line at the OS prompt:

```
python
```

In this case, the interpreter displays the sign >>> to indicate that it is ready for input from the programmer. You then type statements directly to the interpreter, which executes them one at a time. To exit the interpreter, you just type the end-of-line character (CONTROL-D on Unix, CONTROL-Z on DOS or Windows).

The second way consists of writing the Python code in a text file and naming the file by adding the extension .py or a .pyw (on Windows). Then, the interpreter is called with the file name as an argument. So, at the OS prompt, you would type:

```
python CLASSFILENAME.py
```

The code shown in this chapter was written and interpreted using *Python Integrated Development Environment* (*IDLE*), which can be downloaded for free from the site www.python.org/idle.

PROGRAMMING EXAMPLE 1: STATEMENTS AND COMMENTS

A statement in Python does not end with a semi-colon or any other special character. Each statement must be written on a separate line. Indentation is very important as it defines the scope of variables, statements, etc., as we will see in the rest of this chapter.

```
# This is my first program in Python                    (1)
greeting= "Hello World!\n"              (2)
greeting
```

1. The first line is a comment; hence, the interpreter skips it. We recognize a comment because it starts with the # sign. This sign might be written in the middle of a line. In such a case, all the characters after the sign are considered as part of the comment. If a comment spans multiple lines, each line should start with the # sign. This is an example of a multi-lined comment:

   ```
   #this is
   #a multi-lined comment.
   ```

2. This second line is a statement that assigns to the variable **greeting** the string literal "Hello World!" with a carriage return(\n) appended to its end. The second one prints the value of the variable **greeting** (i.e., the assigned string literal).

PROGRAMMING EXAMPLE 2: STRINGS

Strings in Python can be indexed. Some useful string methods are: **isalnum**, **isalpha**, **isdigit**, **islower**, **isupper**, etc. The code below shows how strings are used in Python.

```
greeting="this is a test to see how strings are indexed"      (1)
greeting[0]                          (2)
greeting[2:5]                        (3)
if "test" in greeting                (4)
      print "hooray!"                (5)
```

1. This statement assigns the string "this is a test to see how strings are indexed" to variable **greeting**.
2. The first letter in a string can be accessed through index 0. This statement prints the letter *t*.
3. A substring can be extracted by indexing the first and the last letter locations. This statement prints the string *'is '* to the screen (with a space at the end of the string).

4. This statement checks to see if the string literal "test" exists in variable **greeting**.
5. This statement prints *hooray!* to the screen since the test above evaluated to true.

Note: Strings in Python can span multiple lines. For example, this is a valid string "*This is a valid string. It is written to show that strings in Python can span multiple lines and that the double quotes indicate the start and the end of the string.*" They can be delimited with double quotes or single quotes. They cannot be mutated. Hence, the following statement is invalid:

```
greeting[0]='T'
```

PROGRAMMING EXAMPLE 3: TYPES, VARIABLES, IDENTIFIERS AND LITERALS

The program below initializes four variables **a**, **b**, **c**, and **d**. Python uses dynamic typing where the type of an object is determined at execution. In other terms, variables do not need to be explicitly declared to be of a certain type.

```
a=10
b=5.6
c=a+b                              (1)
d="the result of the addition is"
print d, c                         (2)
```

1. This statement assigns to variable **c** the result of adding values in **a** and **b**.
2. This statement prints *the result of the addition is 15.6*.

Identifiers

An identifier may consist of any combination of letters, digits, and the underscore character (_). The first character must be a letter or the underscore character. Examples of valid identifiers are: `name, agent, count2, application_form`. Examples of invalid identifiers are: `1count, mass*volume, name.first`. Of course, Python reserved words cannot be used as identifiers. Python is case sensitive. Hence, `GREETING, greeting` and `Greeting` are all different identifiers.

Literals

Numeric literals that are written without a decimal point are considered integers. For example, 2008, 0, -99 are all integer literals. A numeric literal that contains a decimal point is considered a float. For example, 4.2 and 1.9 are float literals.

Variables

Declaring a variable in Python is implicit as already mentioned. You simply assign a value to the variable and, implicitly, the variable type becomes similar to the type of this value.

Note: A variable declared inside a function is local to the function and can be accessed inside the function only. A variable that is declared outside any function is global and can be accessed anywhere in the program, as we will see later in this chapter.

The following table summarizes the augmented assignment operators used in Python.

Python augmented assignment operators	
`+=`	add then assign
`-=`	subtract then assign
`*=`	multiply then assign
`/=`	divide then assign
`%=`	compute the modulo then assign

PROGRAMMING EXAMPLE 4: THE PRINT STATEMENT

The **print** statement takes one or more parameters separated with a comma and prints the values of the parameters. Parameters can be identifiers, expressions, or literals. They are separated with commas, which add a space between two arguments. The + operator can also be used to concatenate arguments if one or more of these are strings.

```
age=10
gender="female"
b="and age"
d="the gender is"
print d, gender, b+age                    (1)
```

1. This statement prints *the gender is female and age is 10.*

If you want to incorporate variable values in a string that you want to display, you can specify where the value should be displayed in a string. For example, the following two lines print *Hello Tom. How old are you?* to the screen.

```
name='Tom'
print 'Hello %s. How old are you?' %name
```

Note: Escape characters can be used to format the output. Python escape characters are:

Escape character	Effect
\n	causes output to be pushed to the next line.
\t	causes output to skip over to the next horizontal tab position.
\b	cursor moves backwards one space deleting a character.
\a	causes the system bell to sound (alert).
\'	causes a single quote mark to be printed.
\"	causes the double quotation mark to be printed.
\\	causes the backslash character to be printed.

Formatting Numbers

Formatting specifiers can be used to for mat the output. A formatting specifier has the following structure: `%i.j` where `i` and `j` are two numbers. The first, if specified, indicates the number of space fields used to display the number (usually used for alignmen t). The second number indicates the number of digits to display after the decimal point. Example:

```
number = 45.6129
print 'the number is %.2f' % number
Prints the number is 45.61 to the screen.
```

PROGRAMMING EXAMPLE 5: PROMPTING FOR INPUT

The built-in function **raw_input** is used to prompt the user for input.

```
a=raw_input("Enter first value")              (1)
b=raw_input("Enter second value")
a= int(a)                          (2)
b=int(b)
sum=a+b                            (3)
print sum, id(sum), type(sum)                 (4)
```

1. This statement displays the string *Enter first value* on the screen and waits for the user to input a value. The input value is read as a string and assigned to variable **a**.
2. The function **int** takes a string as an argument and returns the integer value of the string. This statement extracts the integer value from **a** and stores it back in **a**.

3. This statement adds the value of **a** and the value of **b** and stores the result in variable **sum.**
4. The function **id** returns the id of the object passed as a parameter. The function **type** returns the type of the object passed as a parameter. This **print** statement outputs the value of the variable **sum** to the screen followed by the id of the variable and its type. So, assuming the user enters the value 100 for **a** and 22 for **b**, the output of the above program is: *122 10967116 <type 'int'>*.

Note: The same variable **a** was used in different statements. Notice that the values are of different types, and hence, the type of **a** changes accordingly.

PROGRAMMING EXAMPLE 6: LISTS

A list is a data type used to store objects. A list can contain objects of different types, as shown in the program below. The first two objects in list **a** are strings, and the other 2 are integer. List elements are enclosed between square brackets and separated with commas. They can be referenced by specifying an index in the list . The most popular functions used with lists are: **append, insert, extend, index, remove, reverse, len,** and **pop**.

```
a=['fred', 'michael', 33, -90]                    (1)
a                                  (2)
a[0]                               (3)
a[0:3]                             (4)
a[0:10]                            (5)
a[2]=a[3]*2                        (6)
a[0:3]=1, 2, 3                     (7)
a[0:2]=[]                          (8)
len(a)                             (9)
a.append('theodore')               (10)
b=[1,2,[1,2]]                      (11)
c=[1, 2, 3, 4, 5]
c[0:3]=[10, 11, 12]                (12)
c[3:5]=[]                          (13)
```

1. This statement assigns the following elements to the list **a**: 'fred', 'michael', 33, -90.
2. This statement prints all the objects in the list **a**. Hence, the output of this statement is *['fred', 'michael', 33, -90]*.
3. This statement prints the object at index 0 in the list. The output is *'fred'*. If the index is out of the range of the list, an error message is displayed.
4. This statement prints the three objects in the list starting at location 0. We refer to this procedure as the *slicing* of the list. So, this statement prints *['fred', 'michael', 33]* to the screen.

5. This statement prints the objects in the list starting at location 0 and ending at location 9. Since location 9 does not exist in the list, it prints the objects starting at location 0 until the end of the list. Hence, this statement prints *['fred', 'michael', 33, -90]* to the screen.
6. This statement modifies the list at index 2 and makes the object at this location equal to -180 (=-90x2).
7. This statement assigns to the list at indices 0 through 2 the values 1, 2, and 3. So, the list now looks like this: [1, 2, 3, -180].
8. This statement removes two objects from the list starting at index 0 and ending at index 1. So, the list is now [3,-180].
9. The function **len** takes as argument a list name and returns the number of elements in the list. This statement returns 2.
10. The function **append** takes as argument one object and appends it to the end of the list. So, the list is now [3, -180, 'theodore'].
11. Lists can be nested. This statement assigns to list **b** the objects 1, 2 and the list [1,2].
12. It is possible to modify consecutive objects in the list. The list becomes [10, 11, 12, 4, 5].
13. This statement removes objects at indices 3 and 4 from the list, and the list becomes [10, 11, 12].

The table below summarizes the most popular functions used with lists.

Function	Description
`append(o)`	inserts object `o` at the end of the list
`count(o)`	returns the number of occurrences of `o` in the list.
`index(o)`	returns the index of object `o` in the list. If `o` is not found in the list, an exception is generated.
`insert(i,o)`	inserts object `o` in the list at offset `i`.
`pop()`	removes and returns the last element in the list.
`pop(i)`	removes and returns the element at index `i` in the list.
`remove(o)`	removes the first occurrence of `o` in the list. If `o` is not found in the list, an exception is generated.
`reverse()`	reverses the items in the list.
`sort()`	sorts the list in ascending order.
`sort(f)`	sorts the elements in the list according to the criteria specified in `f`. `f` is a function that takes two elements `x` and `y` in the list and returns -1 if `x` should appear before `y`, 1 if `y` should appear before `x` and 0 otherwise.

PROGRAMMING EXAMPLE 7: TUPLES

Similar to lists, tuples may contain elements of different types. By convention, programmers use tuples when they want to group together heterogeneous elements and lists when they want to store together homogeneous elements. Unlike lists, tuples are immutable (you cannot change the elements stored in them).

```
Mytuple="Fred", "male", 36              (1)
Yourtuple=("Mary","female", 28)                (2)
name, gender, age=Yourtuple             (3)
Bigtuple=Mytuple,Yourtuple              (4)
Emptytuple=()                      (5)
Singleelement='alone',                  (6)
Mytuple[0:2]                       (7)
```

1. This statement creates the tuple ('Fred', 'male', 36).
2. This statement shows another way to create a tuple by enclosing the elements in parentheses. We say that the three values are *packed* into the tuple.
3. This statement assigns the string literal "Mary" to **name**, the string literal "female" to **gender** and the value 28 to **age**. This is called *unpacking*. Unpacking a tuple consists of assigning each individual value in it to a variable. The number of variables to the left of the = operator should be equal to the number of values in the tuple.
4. Similar to lists, tuples may be nested. This statement assigns two elements to **Bigtuple**. These are **Mytuple** and **Yourtuple**. So, **Bigtuple** becomes (('Fred', 'male', 36), ('Mary', 'female', 28)).
5. This statement creates an empty tuple called **Emptytuple**.
6. This statement creates the tuple **Singleelement** that contains one element. The trailing comma is mandatory.
7. The tuple is sliced, and the objects at indices 0 and 1 are extracted. The same rules that apply to slicing lists apply to slicing tuples. The statement outputs *('Fred', 'male')*.

Note: Although tuples are immutable, they can still contain mutable elements such as lists.

PROGRAMMING EXAMPLE 8: DICTIONARIES

A dictionary is a data type that consists of key-value pairs. A key-value pair has the form *key:value*. Key-value pairs are separated with commas inside a dictionary. Some popular methods used with dictionaries are: **clear**, **get**, **copy**, **items**, **keys**, **popitem**, **values**, **update**.

```
phone_numbers={"John":44477,"Walter":22311,"Jessica":22155}   (1)
phone_numbers["Walter"]= 9912299                (2)
del phone_numbers["Jessica"]            (3)
```

1. This statement creates a dictionary named **phone_numbers**. This dictionary has three elements, each associated with a key (the entry that precedes the colon, i.e., "John", "Walter" and "Jessica"). The elements of a dictionary are enclosed within curly braces ({}). To create an empty dictionary, the curly braces should not enclose anything. If more than one element has the same key, only the last one is kept in the dictionary.
2. Unlike tuples and just like lists, dictionaries are mutable. The entries in a dictionary are referenced using their keys. This statement changes the phone number of the second entry to 99122. Now the dictionary looks like this: {'John': 44477, 'Walter': 99122, 'Jessica':22155}. If the *key* exists in the dictionary, the corresponding *value* is changed. If not, the *key-value* pair is added to the dictionary.
3. This statement deletes the entry with key "Jessica." If the key specified does not exist, the program exits with an error message.

The table below summarizes the most popular functions used with dictionaries.

Function	Description
`clear()`	deletes all items from the dictionary.
`copy()`	creates a copy of the dictionary. The elements in the newly created copy are references to the elements in the original dictionary. Returns this new copy.
`get(key)`	returns the value associated with `key`. If `key` is not in the dictionary, returns None.
`get(key,value)`	returns the value associated with `key`. If `key` is not in the dictionary, returns `value`.
`has_key(key)`	returns 1 if `key` is found in the dictionary, 0 otherwise.
`items()`	returns a list of *key-value* pairs found in the dictionary.
`keys()`	returns a list of keys found in the dictionary.
`values()`	returns a list of values found in the dictionary.

PROGRAMMING EXAMPLE 9: CONDITIONALS AND BOOLEAN EXPRESSIONS

The **if**-statement redirects the flow of execution according to the value of a condition. The program below outputs the message *Guess the number I am thinking of:* to the screen and waits for input from the user. The input is converted to an integer value and stored in variable **number**. Then, the program checks the value in **number**; if it is less th an 10, the program outputs *Too low.* If the value is greater than or equal to 10 but less than 50, the program outputs the message *Correct guess;* otherwise, the program outputs *too high.*

```
number=int(raw_input("Guess the number I am thinking of:"))   (1)
if number < 10:                    (2)
      print "Too low"
elif number >= 10 and number<50:                               (3)
      print "Correct guess"
else:                              (4)
      print "too high"
```

1. This statement outputs the message *Guess the number I am thinking of:* to the screen and waits for user input. The input is then converted to an integer and stored in **number**.
2. The **if**-statement always ends with a colon (:). If the condition (**number < 10**) after the **if** evaluates to true, the next statement is executed and the message *"Too low"* is output to the screen. Notice the indentation of the next statement.
3. There should always be an **if** before an **elif**. The flow of control is directed to the **elif**-statement if the condition checked by the previous **if** statement evaluates to false. If the condition in the **elif** evaluates to true, the next statement is executed and the message *Correct guess* is displayed. Hence, any number between 10 and 49 would cause the message *Correct guess* to be displayed on the screen.
4. There should always be an **if** before the **else** as well. The **else**-statement is executed if the conditions in the **if** and the **elif** both fail. In this case, the flow of execution is directed to this statement, and the message *too high* is displayed on the screen.

Note: The *elif* and the *else* portions of the sta tement are optional. There might be more than one *elif* part.

Boolean Expressions

Boolean expressions can be formed using relational operators. Logical operators can be used to combine boolean expressions. The following table summarizes the most popular relational and logical operators in Python.

Python relational operators			**Python logical operators**
<	less than	a **and** b	true when both **a** and **b** are true
>	greater than	a **or** b	true when at least one of **a** or **b** is true
<=	less than or equal	**not** a	true when **a** is false; false when **a** is true
>=	greater than or equal		
= =	equal to		
!=, <>	not equal to		

PROGRAMMING EXAMPLE 10: THE WHILE-LOOP

Python relational The **while**-loop causes a block of statements to be repeated as long as a certain condition is true. The code below prints the numbers from 1 to 100, each on a line. It then prints the message *after the loop*.

```
count=0                                    (1)
while count < 100:                  (2)
    count=count+1
    print count
print "after the loop"                    (3)
```

1. This statement initializes the variable **count** to 0.
2. This is the header of the **while**-loop. It causes the next two lines (indented) to repeat as long as **count** is less than 100.
3. This statement prints the string *after the loop* once after the body of the loop is executed. It is not part of the body of the loop since it is not indented.

PROGRAMMING EXAMPLE 11: THE FOR-LOOP AND THE RANGE FUNCTION

Similar to the **while**-loop, the **for**-loop causes a statement or a block of sta tements to repeat. Usually, this form of loops is used when we know the number of times we want the statement(s) to be repeated. The function **range** is used to iterate through a sequence. The program below prints the numbers from 0 to 100 and then the string *after the loop*.

```
for i in range (101):                      (1)
      print i                                                    (2)
print "after the loop"                    (3)
# following is another loop
count=0
for name in ['jeff', 'jefferson','jeffery']                    (4)
            count=count+1
            print count
```

1. This statement constitutes the header of a loop that iterates through the sequence of numbers from 0 to 100. At each iteration, a new value is assigned to **i** starting at 0 and ending at 100.
2. This statement prints the value of **i** at every iteration of the loop.
3. This statement prints the string *after the loop* after the loop is terminated.
4. We can use a tuple, a dictionary, or a list to iterate. In this case, the loop body is executed three times.

The Syntax of the Range Function

The syntax of the range function is `range (I, J, K)` where `I` is any legal user-defined identifier that indicates the lower boundary of the range. If `I` is not specified, the lower boundary is 0. `J` is mandatory. It is any legal user-defined identifier that indicates 1 plus the upper boundary of the range. `K` is any legal user-defined identifier that indicates the iteration step. If `K` is not specified, the step is 1.

PROGRAMMING EXAMPLE 12: BREAK, CONTINUE AND ELSE USED WITH LOOPS

This program reads input from the keyboard, checks to see if the input is divisible by any number between 2 and itself, and outputs a proper message in each case. It then iterates through another loop and outputs all even numbers between 0 and the number itself except 4.

```
number=int(raw_input("Enter a positive number:"))

for x in range (2, number):
      if number % x==0:
        print number, 'equals', x, '*', number/x
        break                          (1)
else:                                  (2)
        print number, 'is prime'
for y in range (0, number, 2)
      if y==4:
            continue                   (3)
      print y                                                    (4)
```

1. The **break** statement is used to break out of the loop. The flow of execution is redirected to the statement after the loop, i.e., to the next for-loop by this statement.
2. The **else** in this statement matches the **for**, we call it an *"else on loops"* (notice the indentation!). If the **for** loop exits (because its condition fails), the **else** statement is executed and the next statement prints that the input number is prime.
3. The **continue** redirects the flow of control to the header of the loop and statements after the **continue** are skipped.
4. This statement is executed only when **y** is different from 4.

PROGRAMMING EXAMPLE 13: FUNCTIONS

The header of a function consists of the keywor d **def** followed by the name of the function, a list of parameters enclosed in parentheses, and the colon sign (**:**). The statements that form the

body of the function m ust be indented. The program below defines a function **factorial** that takes a parameter **n**, and computes and prints the factorial of **n**. The program prompts the user for a positive number and invokes the function to compute the factorial of the number.

```
def factorial(n):  #computes and returns the factorial of n   (1)
    result=1
    for i in range (1,n+1):
        result=result*i
    print 'factorial of', n, 'is', result

def divisible(n,m):                (2)
    if n%m == 0:
        print n ,"is divisible by", m
    else:
         print n, "is not divisible by", m
def prompt_for_input():
  name=raw_input("Enter name:")
  age=raw_input("Enger age:")
  return name, age                 (3)

number=int(raw_input("Enter a positive number:"))

factorial(number)                  (4)
even=2
divisible (number,even)                  (5)
divisible (m=even, n=number)             (6)

n, a= prompt_for_input()                 (7)
print n, a                         (8)
```

1. This line indicates the beginning of the definition of the method **factorial**. Note that all the statements that form the body of the function must be indented. The function takes one argument **n**. A comment briefly describes what the function does.
2. This line defines the beginning of another function **divisible** that has two arguments **n** and **m**.
3. This is the **return** statement of the function **prompt_for_input**. It indicates that the function returns two values **name** and **age**.
4. This statement invokes the function **factorial** by passing to it **number** as a parameter. The value of **number** was read from the keyboard in the previous statement.
5. This statement invokes the function **divisible** and passes to it the two parameters **number** and **even**.

6. This statement shows another way of passing parameters to functions. It shows that the order of the parameters, as passed to the function, does not matter. In the function call, you can specify which value is copied to which function argument. This statement passes the value of **even** to **m** and the value of **number** to **n**.
7. This statement invokes the function **prompt_for_input**, and the first returned value is assigned to **n**, the second one to **a**.
8. This statement prints the value in variable **n** and the value in variable **a**.

Note: Variables declared inside a function, as well as function arguments, are local to this function and cannot be accessed elsewhere.

PROGRAMMING EXAMPLE 14: FILE I/O AND EXCEPTIONS

Similar to JAVA, Python maps a file to an object and associates a stream of bytes with this object. When opening a file, the open mode must be specified to indicate whether the file is open for input or for output. For this, the **open** function is used and two parameters are passed to it: the name of the file and the open mode indicating whether the file is to be open for reading, writing, or both, etc. The program below reads data from the file named **grades.txt** and writes it to the file named **backup_grades.txt**. If the file **grades.txt** cannot be opened for reading, the message *Cannot open file!* is output to the screen. Otherwise, the message *File successfully open for reading* is displayed.

```
try:                                        (1)
    file_name="grades.txt"
    input_file=open(file_name, "r")                 (2)
    out_name="backup_grades.txt"
    output_file=open(out_name, "w")                 (3)
    list=input_file.readlines()                     (4)
    i=0
    while i in range(0, len(list)):
        output_file.write(list[i])                  (5)
        i=i+1
except IOError:                         (6)
    print("Cannot open file!")

else:                                                                   (7)
    print ("File successfully open for reading")

finally:                                (8)
    input_file.close()                  (9)
    output_file.close()                         (10)
```

1. The **try:** indicates the beginning of a block of statements that might cause an error.
2. The function **open** takes two parameters—the name of the file from which input will be read and another string that indicates the mode in which the file will be accessed (**"r"** is for the reading mode). This statement is written as part of the **try** block because an error might occur when trying to open the file for reading (for example, the file might not exist). We will cover the **try** blocks later in this chapter. The reading modes in Python are listed right after the detailed explanation of the code.
3. This statement opens the file **backup_grades.txt** for writing (**"w"** indicates that the file is written for writing). If the file already exists, it is overwritten; otherwise, it is created from scratch.
4. This statement reads all the lines in the file designated by **input_file** and returns a list where each entry holds one line. If you want to read a single line from the file, you have to use the **readline** function. It is possible to specify the maximum number of bytes you want to read by passing a parameter to the functions **readlines** and **readline** (example **readline**(16) reads 16 bytes from the line).
5. This statement writes the content of **list** at index **i** to the file designated by **output_file** (namely, **backup_grades.txt**).
6. This statement indicates the beginning of a block that has to be executed in case the indicated error -**IOError**- is generated. So, if the file cannot be open for input, the next statement is executed and the message *Cannot open file!* is displayed on the screen.
7. The **else** matches the **except** and indicates a block of statements to be executed only if no exception is thrown. So, the statement outputs the message *File successfully open for reading* if no error was generated previously. In other words, this block is executed *if and only if* the except block is not.
8. The **finally** clause indicates a block of statements that will be executed whether or not an exception is thrown.
9. This statement closes the input file. This file stream cannot be used anymore until it is open again.
10. The statement closes the output file to avoid any loss of data. It is very important to remember to close a file stream open for output to avoid any loss of data.

File-open modes:

`"a"`	appends output to the end of the file. If the file does not exist, it is created.
`"w"`	opens file for output. If the file exists, it is truncated. If the file does not exist, one is created.
`"w+"`	opens file for input and output. If the file exists, it is truncated.

If the file does not exist, one is created.

`"r"` opens file for input.

`"r+"` opens file for input and output.

Notes:

- IOError exception is thrown if the "r" or "r+" modes are specified and the file cannot be open.
- Appending "b" to any of the modes ("ab", "wb", etc.) opens the file in binary input or output instead of text. This is valid on Macintosh and MS Windows only.

The table below summarizes the most popular functions used with files.

Function	Description
`close()`	closes the file.
`fileno()`	returns the file's file descriptor. The file's file descriptor is a number (integer) that the OS uses to maintain information about the file.
`isatty()`	returns true if the file is a tty-like device (an interactive device such as the keyboard), false otherwise.
`read(size)`	reads at most `size` characters from the file. If the file contains fewer than `size` characters, the method reads up to `EOF`.
`readlines()`	reads all remaining lines into a list of strings.
`tell()`	returns the file's current position.
`truncate(size)`	truncates data in file and leaves `size` bytes (less if `size` is too big). The parameter `size` is optional. If it is not specified, all data is deleted.
`write(string)`	writes the string to the file.
`writelines(list)`	writes all the strings in `list` to the file.

PROGRAMMING EXAMPLE 15: MORE ON EXCEPTIONS

Python exceptions are similar to JAVA exceptions except that they might take an *else*-statement. In the program below, the code opens the file **input.txt** for reading data from it. The second statement reads the first line in the file and assigns it as a string to **s**. The **int** function converts the string to an integer and assigns the value to **i**. Two errors might occur in this example: It is possible for the file not to be there, in which case the **open** function will fail. It is also possible for the first line read from the file not to contain a valid number, in which case the **int** function fails. In the former case, the flow of execution skips the rest of the **try** block and is redirected to the first **except** block (**IOError**). In the latter case, the flow of execution is redirected to the second **except** block (**ValueError**). In case of any other error, the flow of control is redirected to the third **except** block (where no type of error is specified). If no error is generated and

hence no **except** block is executed, the **else**-block is executed. The **finally**-block is executed in all cases. We illustrate all possible cases immediately after the code.

```
try:
       file=open("input.txt","r")
        s=file.readline()
        i=int(s)
except IOError:
         print("could not open file!")
except ValueError:
         print("could not convert line to a number")
except:
         print("any other error")
else:
         print("successful conversion")
finally:
         print("This will be printed no matter what")
```

In case the file cannot be open, the output is:

```
could not open file!
This will be printed no matter what
```

In case the file is open but a number cannot be extracted from the string, the output is:

```
could not convert line to a number
This will be printed no matter what
```

In case the file is open and a number is successfully extracted from the string, the output is:

```
successful conversion
This will be printed no matter what
```

PROGRAMMING EXAMPLE 16: WRITING AN OBJECT TO A FILE (PICKLING OR SERIALIZATION)

Writing objects to a file is called *pickling* or *serialization*. Some books use the words *marshalling* or *flattening* to mean the same thing. Pickling allows the conversion of any Python object to a string representation. In the program below, if the file **output.txt** is properly open for writing, the string representation of **list** is sent to it and the message *pickling successful* is displayed on the screen. Otherwise, the message *file could not be open* is displayed.

```
import pickle                              (1)
try:
    file=open("output.txt","w")
    list=['what','should',12, [5, 'I','say']]
```

```
    pickle.dump(list, file)              (2)
except EOFerror:
          print('file could not be open')
else:
          print('pickling successful.')
finally:
          file.close()
```

1. This statement imports the module **pickle** prior to using it in the rest of the code. A module is a group of functions (it is similar to packages in JAVA, but it contains functions rather than classes).
2. This statement writes a string representation of **list** to **file**.

Note: Extracting an object from a file in which it had been pickled is called *unpickling*. This is done using the function **load**. Example: `object=pickle.load(file)` extracts the object that was pickled in the output stream **file** and assigns it to **object**.

PROGRAMMING EXAMPLE 17: CLASSES AND INHERITANCE

In a class, you define attributes and methods. You can then create an instance of the class and use a reference to it to access the attributes and the methods of this instance. This program defines two classes **Point** and **ThreeDPoint**. Class **Point** has two attributes, **x** and **y**, and the methods **move_up**, **move_down**, **move_left**, **move_right**, and **move_left_and_up**. Class **ThreeDPoint** inherits the attributes and methods from **Point**. It has one more method namely, **move_forward** and the attribute **z**. It overrides the method **move_up**. The program also defines the method **main** and invokes it from outside any class.

```
class Point:                              (1)
      "define a point in 2D"              (2)
    def __init__(self, abcissa, ordinate):                    (3)
        self.x=abcissa                                        (4)
        self.y=ordinate                                       (5)
    def move_up(self, i):                                     (6)
        self.y=self.y+i
    def move_down(self, i):
        self.y=self.y-i
    def move_right(self, i):
        self.x=self.x+i
    def move_left(self, i):
        self.x=self.x-i
    def move_left_and_up(self, i, j):
            self.move_left(i)                (7)
            self.move_up(j)
```

```
class ThreeDPoint(Point):                    (8)
    def __init__(self, abcissa, ordinate, height):
        self.x=abcissa
        self.y=ordinate
        self.z=height
    def move_forward(self, k):                                    (9)
        self.z=self.z+k
    def move_up(self,i):                                          (10)
        self.y=self.y+i+0.15

def main():                          (11)
    point1 = Point(10, 3)                                         (12)
    print '(',point1.x,',',point1.y,')'                           (13)

    point1.move_up(10)
    print '(',point1.x,',',point1.y,')'
    point1.move_left_and_up(5, 4)
    print '(',point1.x,',',point1.y,')'

    point2 = ThreeDPoint(0,0,8)                                   (14)
    print'(',point2.x,',',point2.y,',',point2.z,')'
    point2.move_up(10)                 (15)
    print'(',point2.x,',',point2.y,',',point2.z,')'

main()                               (16)
```

1. This statement indicates the beginning of class **Point**.
2. This is a *doc string* that describes briefly what the class defines.
3. This statement indicates the header of the method used to initialize the attributes of the class. The argument list always contains the identifier **self** as a first argument; so do all the methods in the class. This is a reference to the current instance of the class. Think of it as being similar to *this* in JAVA.
4. This statement assigns the value of **abcissa** to the attribute **x** of the current instance. Note also the use of **self**.
5. This statement assigns the value of **ordinate** to the attribute **y** of the current instance.
6. This defines the header of a method called **move_up**. The first argument is **self**.
7. This statement invokes the method **move_left**. It passes the value of the parameter **i** as an argument to the method. Note that **self** is not used in the parameter list when invoking the method.
8. This line indicates the beginning of class **ThreeDPoint**, which inherits from **Point**.

9. It is possible for a class to have methods and attributes other than those it inherits from its parent class. **move_forward** is such a method.
10. **move_up** is overridden in class **ThreeDpoint.** It adds an extra 0.15 to the new value of **y**.
11. This statement indicates the beginning of method **main**. The indentation indicates that **main** is not part of any of the classes defined above.
12. This statement creates an instance of class **Point** and uses **point1** to reference it. It assigns value **10** to the attribute **x** of **point1** and value **3** to its attribute **y**.
13. This statement prints the values of the **x** and **y** attributes of the instance referenced by **point1** surrounded with parentheses. So, the output of this statement is *(10,3)*.
14. This statement instantiates class **ThreeDPoint .** So, it constructs an object and assigns the value **0** to its attributes **x** and **y** and **8** to attribute **z**.
15. This statement invokes method **move_up** as overridden in the child class **ThreeDpoint**.
16. This statement invokes the method **main**. The flow of control jumps inside the method. The output of the program is:

```
( 10 , 3 )
( 10 , 13 )
( 5 , 17 )
( 0 , 0 , 8 )
( 0 , 10.15 , 8 )
```

Note: Method overloading is not possible in Python. If you define the same method twice in a class, only the second one is kept.

PROGRAMMING EXAMPLE 18: MODULES

A module is a file containing definitions and statements. Whenever you need to access these definitions and sta tements, you should impor t the module . The **import**-statement can be placed anywhere in the code pr ior to the sta tement using the module . However, it is nor mal practice to place it at the top. You can find pre-defined Python modules on the Python website *www.python.org*. When a module is impor ted (for example, the module **my_module** above), the interpreter searches the current working directory for a file named my_module.py. If the file is not found, the in terpreter searches for it in the dir ectory specified in the en vironment variable PYTHONPATH. If the file is not in the dir ectories specified in this var iable either, the interpreter searches for it in a default path that is specified when installing Python on your PC. The next program shows two separate files, one (to the left) that has code defining the module - **my_module** - and one that uses this module (to the right). When executed, the code outputs the message *factorial of 10 is: 3628800*.

```
# This module has some
# mathematical functions that I
will
# be needing. It is saved to file
# "my_module.py"
def factorial(n):
    result=1
    i=1
    while (i<=n):
        result=result*i
        i=i+1
    return result
```

```
import my_module                    (1)

print 'factorial of 10 is:'
print my_module.factorial(10)       (2)
```

1. This statement imports the module **my_module** where the function **factorial** is defined.
2. To invoke the function in a module, you specify the name of the module and then the name of the function (both are separated with a dot). This statement invokes the function **factorial** by specifying the module **my_module** in which the function is defined.

Note: It is possible to import a specific function from a module instead of the entire module. For example: `import mymodule.factorial` imports the single function factorial from the specified module.

PROBLEMS

1. Write a program that prompts the user for 10 numbers and creates a list of these numbers where each number appears only once. Print this list.

2. Write a function that accepts as input a number `n` and prints numbers from `n` down to 1. Write the function recursively. Recursion in Python is similar to recursion in JAVA and in C.

3. Write a program that prompts the user for names and telephone numbers. It then stores these name-telephone number pairs in a dictionary, sorts the elements in the dictionary, and prints them out (sorted).

4. Write a program that defines a class `Movie` that has the following attributes: `title`, `director`, `length` (length of the movie in minutes). Create five objects of class `Movie`, pickle them, and store them in a file.

5. Write a program that imports the module that you have written in the previous exercise. It unpickles the movie objects and prints them to the screen.

6. Write a program that prompts the user for a number n and prints the n first numbers in the Fibonacci series. The Fibonacci series is the series of numbers `1, 1, 2, 3, 5, 8,...` where the first two numbers are 1 and then every number is the sum of the previous two numbers. Write a function that generates the Fibonacci series using a loop.

7. Repeat the previous exercise by writing the computation recursively this time.

8. Write a function that takes as argument the name of a text file (`input`) and returns the name of another text file (`output`). The content of output is the content of input where each line has been reversed. For example, if the following two lines are in file `input`:

   ```
   Hello World
   How is everyone doing?
   ```

   the file `output` will have the lines:

   ```
   dlroW olleH
   ?gniod enoyreve si woH
   ```

9. Write a program that prompts the user for the name of a file, opens the file for reading, and then outputs how many times each character of the alphabet appears in the file.

CHAPTER SEVEN

Instant Perl

The Perl language was written by Larry Wall while he was working at NASA's Jet Propulsion Labs. Since the first version of the language in 1987, many people have contributed to its further development. The language was originally created to put together the best of several languages and utilities such as C, sed, awk, etc. Although it was written in C, but unlike in C, you do not have to perform low-level tasks such as allocating memory before using it, de-allocating memory after using it, etc. It is portable, meaning the code will run on any PC that has Perl. It is free and can be downloaded from either of the following sites: *www.perl.com/CPAN-local/src/www.gnu.org/* or *www.activestate.com/ActivePerl/download.htm* or *www.opensource.org/licences/*. The language was created for text-processing, rapid-development, OS-utilities, but the most popular use of it has been CGI (Common Gateway Interface) programming (generating web pages dynamically). Some extensions to the Perl/CGI concept are PerlScript, mod_perl, and HTML::Mason. We will not cover CGIs in this tutorial, but we will try to give the language syntax and basics that will help you understand them.

To write code in Perl and run it, all you need is a text editor and the language interpreter. You type the code in a text file and give it the extension *.pm* or *.plx* (*.pl* used with Perl 4 and earlier). For example, if your code is saved to file *named my_first_program.plx*, to execute the code, you have to type the command:

```
perl [-c][-d][-w] my_first_program
```

The brackets indicate that the parameters are optional. -c is the parameter that makes Perl check the syntax of the script and exit without executing it, -d is the parameter required to run the code with debugging, and -w is used to run the code with warnings.

PROGRAMMING EXAMPLE 1: STATEMENTS, COMMENTS, AND THE PRINT STATEMENT

In Perl, statements end with a semicolon. The program shown below prints the two lines *Hello World!* and *Good bye now! Program will terminate.* to the screen.

```
#This is my first code written in Perl                    (1)
print "Hello World!\n";                                   (2)
print "Good bye now!","Program will terminate";           (3)
```

1. This line is a comment. This is indicated by the # sign. Hence, the interpreter skips it.
2. This statement outputs the string *Hello World!* to the screen. The **\n** is the character sequence that appends a carriage return to the output string so that subsequent output is sent to the next line. This is an escape character. Below, we show a list of all escape characters in Perl.
3. The **print** statement takes one or more parameters separated with a comma. This statement prints the string *Good bye now! Program will terminate.* to the screen.

Escape characters can be inserted in string literals for special meanings, as shown in the table below.

Escape character	Function
\n	new line
\r	carriage return
\t	tab
\b	backspace

Escape character	Function
\a	beep sound
\xmn	outputs the character that corresponds to the specified ASCII index (m and n are hexadecimal digits).
\0mn	outputs the character that corresponds to the specified ASCII index (m and n are octal digits).
\cX	control character. X indicates the character. For example, \cC is the equivalent to CONTROL-C.
\u	converts the following letter to uppercase.
\l	converts the following letter to lowercase.
\U	converts all the following letters to uppercase until \E is found.

Escape character	Function
\L	converts all the following letters to lowercase until \E is found.
\Q	disables pattern matching until \E is found.
\E	used with \U, \L and \Q. Terminates their effect.

You can use either the single quote m arks or the double quote m arks to enclose a str ing. The double quote m arks allow for var iable substitution, but single quote m arks do not. We will explain this in the section when we explain variables more thoroughly.

BLOCKS

Similar to JAVA and C, a block is formed of statements enclosed between curly braces({}). It can appear anywhere in the code.

Example:

```
{
     my $a=10;
     print "this is a block of two statements";
}
```

This defines a block of two sta tements. The first declar es a variable that is local to the block (local variables are discussed later). This is indicated with the keyword **my**. The second statement outputs *this is a block of two statements* to the computer screen.

PROGRAMMING EXAMPLE 2: IDENTIFIERS, TYPES, VARIABLES, AND VARIABLE SUBSTITUTIONS

An identifier can be composed of alph abetical characters (a, ..., z, A, ..., Z), the underscor e character (_), digits, and one of the following characters (%, @, and $), which we will explain below. The next letter after the @, $, or % shouldbe either an alphabetical letter or the underscore sign (_). Perl is case sensitive ("Age" is different from "AGE," and they both are different from "age"). An identifier cannot exceed 255 characters in length. An important feature worth pointing out is the possibility of using a reserved word to name an identifier. This is possible because the name will be preceded by either @, %, or $. Hence, $print is a valid identifier. The program below outputs the message *You are now: 18 and in 10 years, you will be 28 years old.*

```
$age=18;                                    (1)
print "You are now:";                 (2)
print $age;                           (3)
$period=10;                           (4)
$age=$age+10;                         (5)
print "and in $period years, you will be $age years old.";    (6)
```

1. This statement assigns the value 18 to the variable **$age**. In Perl, variables are not declared to be of any particular type. When a value is assigned to a variable, the variable takes the type of this value. Also, you can store in the same variable a value of a certain type (an integer for example) and then later, a value of a different type (a string for example).
2. This statement prints the string *You are now:* on the screen.
3. This statement prints the value of the variable **$age** on the same line as the previous output.
4. This statement assigns 10 to variable **$period**.
5. This statement increments the value in variable **$age** by 10.
6. Double quotation marks allow for string substitution; single quote marks do not. You can insert in a string the name of a variable, and the resulting output will contain the value stored in this variable when the statement is executed. This statement prints the values stored in variables **$period** and **$age** as part of the output. This is what we call variable substitution. This is only possible if you enclose the string with double quotes.

This statement prints the message *and in 10 years, you will be 28 years old.*

Note: Perl has three basic data types. They are: scalars, arrays, and hashes.

Scalars

Scalars are simple variables. They are indicated by the $ sign a t the beginning of their n ame. A scalar can be a number, a string, or a reference (a pointer to another scalar). Numbers in Perl are either signed in tegers or double-precision floating-point values. Literals can be in teger, negative integer, scientific notation (10.09E23), hexadecimal (0xffff), octal (0377), or floating point (45.62). A decimal point in a numeric literal indicates a floating-point value; otherwise, the literal is an integer value. To improve readability of large numbers, Perl allows you to use the underscore character between the digits of one n umber. For example: 9_897_321 is the same as 9897321.

Arrays and Lists

A list is a set of elemen ts. This is an example of a list : (5.6, "Juice", 908). As you can see , a list may contain elements of different types. It can also con tain another list. An array is a named list. Array names are preceded by the @ sign. S o, the follo wing is an ar ray: @a=(5.6, "Juice", 908). Arrays store scalars. The first item in the ar ray (5.6) is a floating-point number, and it is at index 0 in the array. The second one is the string "Juice" and is at index 1; the third one is an integer (908) and is a t index 2. An empty ar ray is called a **null array**. We will see more about arrays later in this chapter.

Hashes

Hashes are unordered sets of *key/value* pairs. They are preceded by the % sign. They are similar to dictionaries in Python where each entry contains a *key/value* pair and the item is accessed through the key associated with it. Hence, all keys in one hash should be distinct. The following depicts a hash (or hash table), which contains three key/value pairs, namely, (John, 82), (Mary, 75) and (Max, 12).

John	Mary	Max
82	75	12

Initializing Variables

In Perl, initialization of variables is not mandatory. By default, Perl initializes all scalars to zero, arrays and hashes to the null array and null hash. More than one variable of the same type can be initialized to the same value. Example: `$age=$salary=$promotion=10;`

Note: If you want to avoid using a variable that has not been explicitly declared, you can include the following statement at the top of your code.

```
use strict;
```

Scope

The scope of a variable is global by default. If you want to declare a variable to be local to a certain subroutine (discussed later) or a block, you have to precede the name with the identifier **my**. Example: `my $a;` declares variable **a** to be local to the scope of the statement (subroutine or block).

Note: It is very easy to make mistakes when dealing with local variables, especially that Perl does not warn you about uninitialized variables. Notice the difference in the output of the code shown below. Both code pieces compile and run.

The code below prints 10:

```
if ($x==0)
$y=10;
print $y;
```

The code below prints 0:

```
if ($x==0)
my $y=10;
print $y;
```

PROGRAMMING EXAMPLE 3: ARRAYS

An array is a list of items with a name that identifies it. Perl allows you to create arrays where elements can be of different types. The program below shows how to manipulate arrays by referencing, adding, and removing elements from them. Some interesting functions that can help you manipulate arrays are: **shift, pop, chop, chomp, push, pop, sort, reverse, join**, and **split.**

```
@grades=("John", "Marc", "Matthew", "Alphonse", "Hector");        (1)
print $grades[0];                                                 (2)
print $grades[-2];                                                (3)
print @grades[0,4];                                               (4)
print @grades[1..3];                                              (5)
print @grades[-2,1..3];
$grades[2]="Carl";                                                (6)
$grades[7]=0;                                                     (7)
print @grades;
$first=shift @grades;                                             (8)
$last=pop @grades;                                                (9)
print @grades;                                                    (10)
print chop (@grades);                                             (11)
```

1. This statement creates an array of five elements all of type string. Arrays in Perl are dynamic in size. Even if you create an array of 10 locations, you can add an element to the 100th location, and this grows the array size to 100.
2. Items in an array are referenced through their index. The first element is at index 0 or $[, the last one at index -1 or `$#arrayname` where `array_name` is the name of the array. This statement references the first element in the array (at index 0). Notice the use of the **$** sign in the name of the variable instead of the @ sign. This is because each array location is a scalar variable. This statement prints the string *John*.
3. You can reference an element in the array with a negative integer. The negative integer references from the end of the array. The last element in the array is at index -1, the one before it is at index -2, and so on. Hence, this statement prints the string *Alphonse*.
4. Several elements in the array can be referenced together by indicating the index of each. This statement prints *John Hector*.
5. It is possible to reference consecutive elements in the array by indicating the index of the first one and the index of the last one. This statement prints the elements in the array starting at location 1 and ending at location 3. Hence, this statement outputs *Marc Matthew Alphonse*.

6. Elements in the array can be modified. This statement assigns the string Carl to array location 2. Hence, the array now contains the elements "John", "Marc", "Carl", "Alphonse ", and "Hector".
7. It is not necessary for all array elements to be of the same type. This statement assigns the value 0 to the array location 7. After this statement, the array contains the following elements: "John", "Marc", "Carl", "Alphonse", "Hector", 0. Note here that the array location 7 does not exist since the array contained only five elements prior to inserting 0. It is still possible to add an element to a nonexistent location in the array; the element is added to the end of the array.
8. This statement removes the first element from the array and assigns it to variable **$first**. The array becomes: "Marc", "Carl", "Alphonse", "Hector", 0.
9. This statement removes the last element from the array and assigns it to variable **$last**. The array becomes: "Marc", "Carl", "Alphonse", "Hector".
10. This statement prints the array grades. The output is: *Marc Carl Alphonse Hector*.
11. The function **chop** allows you to remove the last character in each element in the array. The array becomes: *Mar Car Alphons Hecto*. The function **chomp** can be used to remove the \n character from the end of each element stored in the array.

Note: You can extract the size of the ar ray by using a scalar. The following statement extracts the size of array `@a` and assigns it to scalar `$size: $size=@a;`

The table below summarizes some of the most useful methods and operators used with arrays.

Method/operator	**Description**
`join(STRING, DELIMITER)`	creates a flat file database from an array. Entries from the array are copied to the database and delimited with the specified DELIMITER.
`split(DELIMITER, STRING)`	creates an array of elements by splitting STRING every time DELIMITER occurs.
`sort(SUB, ARRAY)`	sorts the elements in ARRAY in ascending or descending order.
`reverse(ARRAY)`	reverses the order of the elements in ARRAY.
`chomp ARRAY`	removes the \n character.
`keys ASSOC_ARRAY`	returns a list of all the keys in the associative array.
`values ASSOC_ARRAY`	returns a list of all the values in the associative array.
`returns ASSOC_ARRAY`	returns a list of two elements that consist of the next *key/value* pair in the associative array.

PROGRAMMING EXAMPLE 4: MORE ON ARRAYS

```
@a=(1, 2, 3, 4, 5, 6, 7, 8);
splice @a, 2, 3, (10, 11, 12);                                    (1)
print join 20, '*',@a              (2)
print reverse @a                   (3)
@b=(4..16)                         (4)
print @b;
@a=(1, 2, 3);
@b=("a", "b", "c");
@c=(@a, @b);                       (5)
print @c;                          (6)
%hire_dates=("John","1998", "Mary","2003", "Jessica","2008"); (7)
print $hire_dates{"John"};                 (8)
$hire_dates{"Foutine"}="2005";                    (9)
delete($hire_dates("John"));                      (10)
```

1. **splice** is used to insert values in an array. The general syntax is splice @IDENTIFIER, OFFSET [, LENGTH [,(LIST)]]; where IDENTIFIER is the array name. OFFSET is an integer value indicating the index of the first element to remove. LENGTH is an integer value indicating the number of items to remove, and LIST is any list of comma-separated valid values. The values are inserted in the array at locations indexed starting at OFFSET and ending at OFFSET+LENGTH-1. If no list is specified, the elements from OFFSET to OFFSET+LENGTH-1 are simply deleted. This statement insets the shorter list (10, 11, 12) in array @a and replaces the values at the locations starting at index 2. The array becomes (1, 2, 10, 11, 12, 6, 7, 8).
2. **join** is used to concatenate an item with a list; the second parameter is the delimiter to insert in the new list. This statement prints 20*1*2*10*11*12*6*7*8.
3. This statement prints the elements in the array in reverse order.
4. You can create an array that consists of a range of integers. This statement creates an array of integer values from 4 to 16.
5. If you include a list as an element of another list, the included list is flattened and its elements become scalars in the including list.
6. This prints *123abc*. And the statement $c[3] gives us *a*.
7. Associative arrays are arrays where the elements are stored in *key/value* pairs. They are often called *hashes*. You use the key to access the value in the array. The name starts with the % character. This creates an associative array where each entry is referenced by its key. The keys are "John", "Mary", and "Jessica" and their associated values are "1998", "2003", and "2008", respectively.
8. This prints the value *1998* associated with key "John" in the array.

9. This adds the elements "Foutine" and "2005" to the array.
10. This deletes the entry associated with the key "John".

Note: The built-in array **@ARGV** holds the parameters that are passed to the program when you execute it. **%ENV** holds the environment variables.

PROGRAMMING EXAMPLE 5: EXPRESSIONS AND OPERATORS

```
print 19/3;                          (1)
print int(19/3);                (2)
print ((9+5)/(6*(9-7)))              (3)
```

1. Perl does not have integral division, and thus, this statement outputs *6.3333333*.
2. The statement outputs *6*. The function (or subroutine) int() returns the part of the number before the decimal point.
3. This statement prints 1.16666666666667.

Following is a list of some arithmetic operators in Perl. For a full list, you may consult the perlop manpage (by typing man perlop on the prompt).

Arithmetic operators		**String operators**	
Operator	**Operation performed**	**Operator**	**Operation performed**
+	Addition	. (dot)	Concatenate
–	Subtraction	lt	Less than
*	Multiplication	gt	Greater than
/	Division	le	Less than or equal
%	Modulus	ge	Greater than or equal
+	Positive sign	ne	Not equal to
–	Negative sign	eq	Equal to
++	Autoincrement (postfix and prefix forms)	=	Assign
––	Autodecrement (postfix and prefix forms)	.=	Concatenate and assign
**	Exponentiation		
Comparison operators		**Logical operators**	
<	Less than	&&, and	AND
>	Greater than	||, or	OR

(Continued)

Comparison operators		Logical operators	
Operator	**Operation performed**	**Operator**	**Operation performed**
<=	Less than or equal	!, not	Not
>=	Greater than or equal	xor	XOR
< = >, !=	Different		
==	Equal to		
Assignment operators			
=	Assign		
+=	Add then assign result		
-=	Subtract then assign result		
*=	Multiply then assign result		
/=	Divide then assign result		
%=	Modulus then assign result		
**=	Exponent then assign result		

PROGRAMMING EXAMPLE 6: CONDITIONALS (IF-STATEMENT)

The following program prints *Too high!* to the screen.

```
$number =100;                                                    (1)
if ($number <5)                                                  (2)
{
            print "Too low!";
}
elsif (($number >5) && ($number<20))                             (3)
{
            print "Acceptable";
}
elsif (($number >20) && ($number <80))
{
            print "More than average";
}
else                                                             (4)
{
            print "Too high!";
}
```

1. This statement assigns the value **100** to the scalar **$number**.
2. The conditional expression evaluates to true if the value in the scalar **$number** is less than 5. In the example above, it evaluates to false, and hence, the next print statement is not executed and the flow of execution is re-directed to the elsif statement that follows it.
3. This condition also evaluates to false.
4. The execution flow is redirected to the else part of the code since none of the previous if or elif statement conditions evaluated to true. This statement prints *Too high!* to the screen.

PROGRAMMING EXAMPLE 7: CONDITIONALS (UNLESS-STATEMENT)

The *unless*-statement checks for a condition and executes a block of statements only when this condition is false. The code below outputs *Too high! Let us continue.*

```
$number =100;
unless ($number < 50)
{
      print "Too high!";
}
print "Let us continue.";
```

PROGRAMMING EXAMPLE 8: LOOPS

Similar to all languages, loops allow the program to repeat a block of statements a certain number of times or while a certain condition is true. Perl has four types of loops: **for-loops, while-loops, do-loops, and foreach-loops**. The program below includes four loops, three of which are equivalent. The first three loops print the numbers from 10 down to 1, each on a separ ate line. The fourth one prints:

```
for ($count=10; $count>0; $count--)                    (1)
{
      print $count;                (2)
      print "\r";                  (3)
}
#another way to write the same loop
$count=10;
while ($count > 0)                 (4)
{
      print "Hello\r";
      $count--;
}
#yet another way to re-write the same loops above
```

```
$count=10;
do{
      print "Hello\r";                    (5)
      $count--;
} while ($count > 0);
#a different type of loops
foreach $name ("John", "Peter", "Fan")                          (6)
      {
            print $name."\t";                (7)
      }
```

1. The loop initializes variable **$count** to **10** and then checks for the condition **$count>0**. It executes the next two statements as long as the condition evaluates to true. After each iteration, **$count** is decremented by 1.
2. This statement prints the value of **$count**.
3. This statement prints a carriage return to push the next output to the following line.
4. This is the start of the while-loop. The condition is evaluated, and the following block of statements is executed as long as the condition evaluates to true.
5. This is the first statement inside the scope of the do-loop. This statement, as well as the next one, is executed at least once since the condition is evaluated after each iteration of the loop.
6. The *foreach*-loop is normally used to step through the elements of an existing array. At each iteration of the loop, the variable **$name** is assigned a value of one item in the array.
7. This statement is executed three items. Each time, a value in the array is assigned to the variable **$name** and printed to the screen.

PROGRAMMING EXAMPLE 9: SUBROUTINES

The program below defines a subroutine **fact** that computes the factorial of the number passed to it as a par ameter. The program invokes the subr outine to compute the factor ial of 6 and prints the message *factorial of 6 is 720* to the screen.

Subroutines are similar to functions and methods in C and J AVA, respectively. Unlike C and JAVA, a subroutine in Perl always returns a value, even if no return statement is written. Unless a return statement is included, the subr outine returns the last value computed. Also , unlike C or JAVA, the header of the subr outine does not necessar ily include argumen ts. All incoming parameters are packed into an array with the name @_ (which you could specify or not). A subroutine should be defined or at least declared prior to invoking it. In other words, you either write the subroutine before any statement that invokes it or you declar e it before such statement and postpone defining it to later in the code.

```
sub fact {                          (1)
      $fact=1;
      for ($i=1; $i<=@_[0]; $i=$i+1)                          (2)
            {$fact=$fact*$i;}
      return $fact;                 (3)
      }
$num=6;                             (4)
$factorial= fact $num;                    (5)
print "Factorial of $num is $factorial";                       (6)
```

1. This line defines the header of the subroutine with name fact. You can think of the subroutine as having one argument of type array. The name of this argument is @_.
2. This is the header of the *for*-loop. This loop computes the multiplication of the numbers from 1 to the parameter that is passed to the subroutine. This parameter is stored in @_[0].
3. This statement returns the value in the scalar $fact.
4. This statement initializes the scalar $num to 6.
5. This statement invokes the subroutine fact and passes $num to it as a parameter. The value returned by the subroutine is then assigned to $factorial.
6. This statement prints *Factorial of 6 is 720.*

Subroutine Arguments

You do not explicitly specify an argument list in the header of the subroutine. Perl packs all the parameters that are passed to a subroutine to an array called @_ and, hence, the parameters can be referenced with @_[0], @[1]... @_[$#_]. You can still assign the elements of this array to local variables inside the subroutine.

Example:

```
sub minmax
{
      my ($a, $b)=@_;
      if ($a<$b)
      {print "$a is less than $b";}
      elsif ($b<$a)
      {print "$b is less than $a";}
      else
      {print "parameters are equal";}
}
minmax(10,14);
```

This code prints *10 is less than 14* to the screen.

In the example above, the parameters passed to the subroutine are assigned to the scalars **$a** and **$b**.

Note: If you need to pass several parameters to a subroutine and one of them is an array, it is a good idea to keep the array as the last parameter.

PROGRAMMING EXAMPLE 10: REFERENCES

A reference is a scalar that stores the address of another variable or data structure. The program above defines a scalar **$a**, an array **@c**, and two references **$b** and **$d**. This code shows us the use of references in Perl. It outputs the following strings, each on a separate line:

```
                        SCALAR(0x183280c)
                        1 2 3
                        ARRAY(0x1832878)
$a=10;                          (1)
$b=\$a;                         (2)
print "$b\n";                   (3)
@c=(1, 2, 3);                   (4)
$d=\@c;                         (5)
print "@c\n";                   (6)
print "$d\n";                   (7)
```

1. This statement initializes the scalar **$a** and assigns the value **10** to it.
2. To create a reference, you add the slash character (\) to the value that you are referencing. This statement initializes the scalar **$b** and assigns to it the address of the scalar **$a** in the memory.
3. This statement prints the value of **$b**. The output is the type of the object that **$b** is referencing, along with its memory location address: *SCALAR(0x183280c)*.
4. This statement creates the array **@c**.
5. This statement creates a reference **$d** to the array **@c**.
6. This statement prints the elements in array **@c**. The output is *1 2 3*.
7. This statement prints the value of the scalar **$d** which is the type of the array that **$d** is referencing, along with its address in the memory. The output of this statement is: *ARRAY(0x1832878)*.

Note: You may create references to subroutines as well.

Dereferencing

It is possible to de-reference a reference variable and access the object value that it is referencing. To do this, you add to the beginning of the r eference variable name the $ character if the variable is referencing a scalar and the @ if it is referencing an array.

Example:

```
@a=(1,2,3);
$b=\@a;
```

The following statements will give the indicated output:

```
print "$b\n";          #prints the value stored in $b i.e. the address of array @a.
print "$$b[1]\n";      #prints the element at index 1 in the array referenced by $b.
print "@$b";           #prints the array that is referenced by $b i.e. (1, 2, 3).
```

PROGRAMMING EXAMPLE 11: REGULAR EXPRESSIONS

One of the m ajor goals P erl was designed for was text pr ocessing. In text processing, string matching is very important. Perl gives a set of meta-characters that make text processing relatively easy. We will show them after the program details. The program above matches the string in scalar **$a** to two different patterns and results in the following output:

```
matches the first pattern
matches the second pattern
```

```
$a="public static void main(String[] args)\r";                    (1)
if ($a =~ m/public|private|protected/ )                            (2)
{ print "matches the first pattern\r";}

if ($a =~ m/^(public|private|protected)/)                          (3)
{ print "matches the second pattern\r";}
```

1. Assigns the string **"public static void main(String[] args)\r"** to the scalar **$a.**
2. Matches **$a** with the pattern that indicates a string containing any of the words "public," "private," or "protected." This returns true, and hence, the next statement is executed and it outputs *matches the first pattern to the screen.*
3. Matches **$a** with the regular expression indicating a string that starts with any of the words "public," "private," and "protected." This returns true, and hence, the next statement is executed and it outputs *matches the second pattern* to the screen.

The table below lists the basic meta-characters that can be used to form regular expressions.

Meta character	Meaning	Example
*	zero or more of the previous character should be matched	`a*` matches any string. `ab*` matches any string that has an `a` followed by 0 or more `b`.
.	any valid character	`a.c` matches any string that has a substring that starts with an `a`, ends with a `c` and has any valid character in between.
|	used to give a choice	`a.c|c.a` matches any string that contains a substring that starts with an `a`, ends with a `c` and has any valid character in between or starts with a `c` , ends with an `a` and has any valid character in between.
()	used for grouping	`a(b|c)*a` applies the * to `(b|c)` so this expression matches any string that contains a substring that starts and ends with an `a` and has zero or more `b` or `c` in the middle.
^	anchors the regular expression to the beginning of a string	`^a(b|c)*` anchors the matching at the beginning of the string. So, this regular expression matches with any string that starts with an `a` and has zero or more `b` or `c` after the `a`.
$	anchors the regular expression to the end of a string	`aa*$` anchors the matching to the end of the string. So, this regular expression matches with any string that ends with one or more `a`.
\	quotes the next character so it is not a metacharacter anymore	`*\$` matches any string that contains the `$` sign.
[]	defines a class of characters	`[1234567890]$` matches any string that ends with a digit.

Matching a Regular Expression

The operator **m** is used to m atch a regular expression that is typed between two slashes (//). The binding operator =~ is used to bind the pattern to the string. Any of the above listed operators can be used in the pattern to match. Example: `"hotmail" =~ m/mail/` evaluates to `true` if `"mail"` exists anywhere in the string `"hotmail"`, `false` otherwise.

Note: If the pattern contains forward slashes(/), you have to precede them with a backslash (\). For example, `/\/My Documents\/PerlDocuments\/TestingCode\/*.txt/`. You should do the same for all metacharacters.

Quantifiers

Quantifiers can be used to match a certain pattern more than once. Following is a table describing their use.

Quantifiers	Effect
{I}	matches exactly I times
{I,}	matches at least I times
{I,J}	matches at least I times but at most J times
*	similar to {0,}
+	similar to {1,}
?	similar to {0,1}

Note: The inverse operator of =~ is !~. This returns true if the pattern is not matched, false if it is.

Character Classes

You can create your own classes by listing their elements inside squared brackets, or you can use Perl built-in special character classes (listed below).

Character class	Their elements
\w	any alphanumeric character and the underscore character
\W	all characters except alphanumeric and the underscore character
\s	any white space (include tab, carriage return, etc.)
\S	any character except white space
\d	numeric digits
\D	non numeric digits

PROGRAMMING EXAMPLE 12: FILE HANDLES

To read from or write to a file, Perl uses a file handle that it associates with the file. A *file handle* is a variable that represents a file. Unlike other variables in Perl, their name does not start with any special character. The program below associates the file named **input.txt** with a file handle **input_file** and the file named **output.txt** with the file handle **output_file**. The program reads a single line from **input.txt** and writes it to **output.txt**. If the **output.txt** already exists, it is overwritten; otherwise, it is created.

```
open (input_file, "input.txt");                (1)
$line=<input_file>;                  (2)
open (output_file, ">output.txt");             (3)
print output_file $line;                   (4)
close (output_file);                 (5)
```

1. This statement associates the file handle **input_file** with the file **input.txt**. The file is open for reading, and this is indicated by the fact that the name of the file is not preceded by any other character, as we will see in the next statements.
2. To read a line from the file, you surround the name of the file handler with <>. This statement reads one line from the file associated with **input_file** and assigns it to the scalar **$line**.
3. This statement associates the file handle **output_file** with the file **output.txt**. The > in the string **">output.txt"** indicates that the file is open for writing. The substring ">>" instead of ">" indicates that the output should be appended to the file.
4. This statement writes the value in **$line** to the file associated with **output_file**.
5. Always close a file when you are done with it, especially if the file was open for output. This avoids loss of data.

Note: To read the whole file, you just enclose the file handle name in < >: and you assign it to an array. Example: `@all=<input_file>;` reads the entire file into the array and stores one line per array location.

The following table summarizes the most popular functions used w ith file handles. The last three functions in the table work with directories only.

Function	Description
`flock(HANDLE, CONSTANT)`	locks the file associated with the file handle HANDLE. CONSTANT can be one of the following: LOCK_SH used when reading from a file, LOCK_EX used when writing to the file, LOCK_NB used when you want to force the script to wait if a lock cannot be obtained, and LOCK_UN to unlock the file. To use this, you have to have the statement `use Fcntl qw(:flock)` in your script.
`seek (HANDLE, OFFSET, LOCATION)`	moves OFFSET bytes from the location in the file. LOCATION can be set to one of the following constants: SEEK_SET to move to the beginning of the file, SEEK_CUR to stay at the current position, SEEK_END to move to the end of the file. To use this, you have to have the statement `use Fcntl qw(:flock :seek)` in your script.
`close (HANDLE)`	closes the file. If the file was locked, this unlocks it as well.
`print HANDLE, STRING`	writes the string to the file associated with HANDLE.
`open(HANDLE, STRING)`	opens the directory with name indicated in STRING and associates HANDLE with it.
`readdir(HANDLE)`	reads the contents of the directory. The returned value can be assigned to an array.
`closedir(HANDLE)`	closes the handle associated with the directory.

PROBLEMS

1. Write a program that requires the user to enter the elements of an array and then prints these elements in reverse order of their entry.

2. Write a program that plays the game "guess the number" with the user. The program starts by generating a random integer number and then asks the user to guess what the number is and inputs it to the program. If the input number matches the program generated number, the program terminates with a success message. If not, the program loops and asks for the number again, telling the user whether the guess was too low or too high. This pattern is repeated until either the user guesses right or he/she exhausts the number of allowed trials (set this number to 5, for example).

3. Write a program that takes as argument the name of an input file and displays the number of occurrences of every vowel in this file.

4. Write a program that reads all the files in the current working directory and displays their content to the screen.

5. Write a program that takes a string as an argument. It then searches all the files in the current working directory for the occurrence of the string and displays all the lines where the string occurs along with the name of the file.

6. Write a program that prompts the user for a string. It then outputs the number of files in the current directory that has this string as an extension in their name. For example, if the user enters "doc", the program lists the number of files that have the extension .doc in the current working directory.

7. Modify the program in Exercise 6 to output the number of files and their names as well.

CHAPTER EIGHT

Instant XML

Extensible Markup Language (XML) is a markup language for data. If we compare it to HTML, HTML being a markup language (a formatting language) for documents, while XML is a markup language for databases. XML is designed to be general purpose, so it can format any data. For example, many programming languages will use XML as a way of formatting strings that are then passed over the Internet or saved on the local computer's hard disk in a text file. XML is best thought of as a database formatting language.

Web development normally has two types of documents: web pages and database files. Each of these documents has its corresponding markup language. HTML is for web pages, and XML is for web database files. There are other database technologies, like SQL (Structured Query Language), but the motivation behind XML is to be an open technique. Openness is defined as easy to view since all the information is contained in a text file, and easy to read and modify because the information is in readable human form. The programmer has control of its readability, and it is often agreed upon to be constructed as readable as possible.

This leads to an important drawback to XML. XML data streams are very long and take more CPU power to process. So, the trade off is obvious: global openness versus speed. The choice will be yours as well.

The XML markup language is similar in style to HTML. This means that XML uses tags, just like HTML. The major difference in XML is that the developer creates (i.e., names) his own tags. The better names you select, the easier the database will be for reading by humans (slower for computers).

Tag Syntax:

```
<tag attribut="value"> . . . </tag>
```

Where the above syntax maintains the same rules as in HTML. Please refer to the chapter on Internet development if you are not familiar with the HTML syntax.

XML tags must adhere to these naming rules:

- Names can contain letters, numbers, and other characters.
- Names cannot start with a number or punctuation character.
- Names cannot start with the letters xml (or XML, or Xml, etc.).
- Names cannot contain spaces.

Any name can be used; no words are reserved.

XML FILE STRUCTURE

XML databases are constructed in a tree-like structure of programmer-defined XML tags. Below is an example:

```
<root>
 <child>
   <subchild>. . . . .</subchild>
 </child>
</root>
```

Notice that the **database** is given a name, in this case <root>. The XML file can contain many roots. Each root stores information about one subject. For example, a root tag could be <employees> and therefore records employee information. Another root tag could be <inventory> keeping inventory information. Following the <root> tag is a <child> tag that divides the database into **records**. A record is a data structure that stores information about one object belonging to the database. For example, if we have an employee database, then John Smith and Mary Jane would be two records in the database. Therefore, in this story, the employee databases keep information about two employees, specifically John Smith and Mary Jane. Each <child> tag has multiple <subchild> tags. These sub-tags are the **fields** of the record. In other words, the sub-tags store particular items of information about the <child> tag, for example, age and salary. So, the employee database records information about two employees. The information types recorded are the employee's name, age, and salary.

Let's look at a more complete example:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<notes>
 <note>
  <date>2008-01-10</date>
```

```
  <to>Bob</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
 </note>
</notes>
```

The above is a note-taking database. The database is called <notes> and its records are called <note>. Each <note> has the following information: date of note, to whom it is for, from whom the note originated, the note's heading, and the actual message. Optionally, the XML-reserved tag <?xml> is used to declare the version of this XML database and the standard encoding rules.

Another example:

```
<bookstore>
 <book category="CHILDREN">
   <title>Harry Potter</title>
   <author>J K. Rowling</author>
   <year>2005</year>
   <price>29.99</price>
 </book>
 <book category="WEB">
   <title>Learning XML</title>
   <author>Erik T. Ray</author>
   <year>2003</year>
   <price>39.95</price>
 </book>
</bookstore>
```

This example shows the use of attributes. Notice that this bookstore database tracks books in records called <book>, but each book can be categorized by a type, like children's book or romance.

THE DTD FILE

Since XML is open standard and is easily accessible by multiple individuals, the need to standardize and apply rules to maintain the correct formatting of the database becomes important. This is implemented through a secondary XML file called the DTD file. In the DTD file, the programmer can define legal tag names, attribute names, and attribute values. The programmer can also define reserved words and constants.

Here is an example invocation of a DTD file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE notes SYSTEM "Note.dtd">
<notes>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
</notes>
```

This takes us back to our <notes> da tabase. This is basically the same file with one addition, <!DOCTYPE>. The <!DOCTYPE> tag associa tes a DTD file with a <r oot> tag. I n the abo ve example, the <notes> tag is associated with the Note.dtd file. The XML file will be validated by this DTD file.

The above .dtd file has the following example code:

```
<!DOCTYPE notes
[
<!ELEMENT notes (note)>
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Where:

- This definition lists all the legal tag and attribute names.
- The file name must end with .dtd file extension.

Notice in the above example that it has the following syntax:

```
<!DOCTYPE A [B]>
```

Where:

- A declares that this DOCTYPE must be associated with the <root> A.
- B declares the list of legal tags in A, called elements.

In the above example, the <notes> database consists of only one legal element: note. The <note> tag has the following legal elements: to, from, heading, and body.

Notice that the element tag is organized as follows:

```
<!ELEMENT A (B)>
```

Where:

- A is the name of a tag.
- B is a list a valid values the tag A can have.
- The list of valid values, represented by B, is used in two ways. The first way lists legal tags associated with A (the first two <!ELEMENTS> are examples of this one). The second form lists legal values (the last 4 <!ELEMENTS> are examples of this one). The reserved words #PCDATA are defined later, but should be thought of as data types.

The <!ELEMENTS> Tag

Syntax:

```
<!ELEMENT element-name category>
or
<!ELEMENT element-name (element-content)>
```

Where:

- element-name — is the legal tag, declared by the programmer.
- category — is the legal type of data that can be written for the element-name.
- (element-content) — is a list of legal element-names.

The <!ATTLIST> Tag

In the definition of an XML tag, attributes are permitted. The <!ATTLIST> tag is used to declare each attribute and associate it with an XML tag. Here is its syntax:

```
<!ATTLIST element-name
                attribute-name
                attribute-type  default-value>
```

Where:

- element-name — is the name of the XML tag.
- attribute-name — is the name of the attribute associated with the element-name.

- attribute-type is the legal value that can be inputted for that attribute.
- default-value is optional, and provides an initial default value for the attribute.

For example:

```
<!ATTLIST payment type CDATA "check">
```

In the above example, the tag <payment> has an attribute called type that can have a string of characters (this is what CDATA means). Its default value is the string "check". In other words, a legal XML expression using the above example would look like: `<payment type="check">` or it could look like `<payment type="savings">` or it could look like `<payment>` and, in this case, `type="check"` is assumed since it is the default value.

Legal Attribute Types

The attribute-type can be one of the following:

Type	Description
CDATA	The value is character data
(en1|en2|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
Xml:	The value is a predefined xml value

Legal Values

The default-value can be one of the following:

Value	Explanation
Value	The default value of the attribute
#REQUIRED	The attribute is required
#IMPLIED	The attribute is not required
#FIXED value	The attribute value is fixed

Special Entities

Special constants can be declared. Below is the syntax of two forms:

```
<!ENTITY entity-name "entity-value">
<!ENTITY writer SYSTEM "http://www.abc.com/entities.dtd">
```

The first syntax defines an entity-name that when written in an XML document will be automatically interpreted to have the value string "entity-value". The second syntax defines a similar constant but associates it with the reserved word SYSTEM. This reserved word connects the constant to the browser, as a link to an outside web page.

For example:

```
<!ENTITY writer "Donald Duck.">
```

In XML

```
<author>&writer;</author>
```

```
The $CONSTANT; expression invokes the "entity-value". From the computer's
point of view it understands:
```

```
<author>Donald Duck.</author>
```

PROBLEMS

1. Create an XML membership database with member name, date of membership, and member password.
2. Create a DTD file that ensures that there is always a member name and that the password must be unique, for question 1.
3. Discuss how the HTML constants for foreign letters could be implemented as a DTD file.
4. Explain why parsing an XML file with a programming language would take a long time.
5. Assume we have a list of foreign characters that we would like to create a DTD database of constants. Assume further that we would like this file to be easy for humans to read and modify. What would the easy-to-read DTD file look?

CHAPTER NINE

XHTML and DHTML

XHTML

XHTML stands for eXtensible HyperText Markup Language. It is a newer and cleaner version of HTML. Since HTML is not very strict on the way you write the contents, some browsers will display the same page differently, while others do not even h ave the power to display pages written with incorrect HTML. XHTML forces you to write correctly. As a result, pages written with XHTML are displayed correctly on any br owser and any technology. I t is impor tant to point out though that while HTML is not case sensitive when it comes to tags , XHTML is case sensitive and only accepts lowercase tags. Also, while HTML allows you to improperly nest elements, XHTML requires that all elements be properly nested and non-empty elements have a closing tag.

Programming Example 1: Mandatory Elements

XHTML documents have to include some elements that do not necessarily appear in HTML. These elements are a **Doctype** declaration, a **<head>** section, and a **<body>** section. The code below shows how to display an image in XHTML and how to include a horizontal ruler. It displays the text *Address:* on a line and then draws a horizontal line. Then the text *Somewhere on this planet* is displayed followed by the image in the indicated file, namely **earth.gif**.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">                    (1)

<html>                                                            (2)
<head>                                                            (3)
```

```
<title> My Home Page</title>
</head>
<body>                                                    (4)
Address: <br />                                           (5)
<hr />                                                    (6)
Somewhere on this planet <br/>
<img src="earth.gif" alt="The Earth" width="100"/>        (7)
</body>
</html>
```

1. An XHTML document must always have a **DOCTYPE** declaration. This element specifies the rules for the language and tells the browser how to display the content of the document. After the code detail, we provide you with a list of available document type definitions (DTDs).
2. In XHTML, you have to start the document with the tag **<html>**. Note now that the tag names cannot be capitalized as in HTML.
3. The **<head>** section is also mandatory in XHTML.
4. All XHTML documents must include a **<body>** section.
5. The tag **
** now has to include the "/" whereas in HTML, the tag is simply **
**.
 is an empty element. Empty elements are elements that normally indicate one item, such as a line break, a horizontal rule, an image, etc., instead of delimiting sections or items (such as an anchor, a title, a header line, etc.). In XHTML, these empty elements should be followed by "/".
6. The tag **<hr/>** displays a horizontal line all across the document.
7. The **<img>** tag now has to end with /> which is not the case in HTML. The **alt** is an attribute name that indicates the alternative text to be displayed in case the image was missing. Note that the attribute name must be written in lowercase and the attribute values must be quoted (example **width="100"**).

Available DTDs (for more details, you can refer to the W3C page at *www.w3schools.com*)

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

Note: For a detailed explanation of each of the above mentioned doctype, you may consult the web page found at the following URL: *http://www.w3schools.com/xhtml/*. You can also use this link to validate any XHTML page.

Important XHTML Syntax Rules

Some major syntax rules differentiate XHTML from HTML: Attribute names must all be in lowercase letters. Their values must be quoted (double quotes), and their names cannot be minimized.

The table below lists some of the attribute name minimizations used in HTML and how they should be explicitly specified in XHTML.

Attribute name as minimized in HTML	Attribute name as specified in XHTML
`Compact`	`compact="compact"`
`Checked`	`checked="checked"`
`Readonly`	`readonly="readonly"`
`Disabled`	`disabled="disabled"`
`Selected`	`selected="selected"`
`Nohref`	`nohref="nohref"`
`Nowrap`	`nowrap="nowrap"`
`Noresize`	`noresize="noresize"`

Another difference between XHTML and HTML is that the former replaces the attribute **name** with the attribute **id** and the attribute **language** included in the **<script>** tag is no more used.

Programming Example 2: Creating Tables

The example below displays a table that shows a schedule of talks that are supposed to be given by people during a certain week. The example shows how to create a table with six columns and three rows. The columns are labeled by weekdays and the rows by time slots. Names of speakers are inserted in the appropriate cells. For example, Alphonse is scheduled to speak on Thursday from 10:00 to 12:00. Sam and Georges are scheduled to share a talk on Friday from 14:00–16:00.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title> Talk Calendar </title>
</head>
<body>
<style type="text/css">                                          (1)
table#schedule      {border: outset 2px}                         (2)
table#schedule td   {border: outset 1px}                         (3)

table#friday {border: inset 1px}
table#friday td     {border: inset 1px}
</style>

<table id="schedule">                                            (4)
<tr>                                                             (5)
<td></td>                                                        (6)
<td> Monday</td>                                                 (7)
<td>Tuesday</td>
<td>Wednesday</td>
<td>Thursday</td>
<td>Friday</td>
</tr>                                                            (8)
<tr>
<td>10:00-12:00</td>
<td>John</td>
<td>Marc</td>
<td>Helen</td>
<td> Alphonse</td>
<td> Ursula</td>
</tr>
<tr>
<td>14:00-16:00</td>
<td>Theodore</td>
<td></td>
<td>Danny</td>
<td>Pascale</td>
<td>
  <table id="friday">                                            (9)
  <tr>
  <td>Sam</td>
```

```
    <td>Georges</td>
    </tr>
    </table>                                                    (10)
  </td>
  </tr>
  </table>                                                      (11)

  </body>
  </html>
```

1. The **<style>** tag indicates the beginning of a style sheet section. A style sheet is used to format the table and add borders around the cells that form it. If no such formatting was needed, this section could be omitted.
2. This line specifies that the border format should be applied to the table with id **schedule**. The format specifies that the border of the rules should be of type outset and 2 pixels thick.
3. This line specifies that in the same table, the border of the columns should also be outset and 2 pixels thick.
4. The tag **<table>** indicates the beginning of a table. The attribute **id** allows you to refer to the table in other sections of the page (for example, in lines (2) and (3)).
5. A table is formed of rows and columns. The tag **<tr>** indicates the beginning of a row. The elements to insert in the row are enclosed in columns, as seen below.
6. The tags **<td>** and **</td>** mark a column. This is the first column in the first row. It is left empty.
7. The word "*Monday*" appears in the second column of the first row.
8. The tag **</tr>** marks the end of the row.
9. It is possible to have a table nested within another table. This line indicates that a table is inserted in this column of the outer table (i.e., in the column labeled *Friday*). The **id** of this inner table is **Friday**. It is used to refer to the table in the style sheet. This id can be omitted if not needed.
10. This tag marks the end of the inner table.
11. This tag marks the end of the outer table.

Programming Example 3: Lists

Often, you need to displa y a list of items . There are three different types of lists : numbered, bulleted, or definition lists. The code below shows the use of these three different types.

```
<!-The first list is an unnumbered list of item-->
<ol>                                                    (1)
<li>Take a shower                                       (2)
<li>Call Mom                                            (3)
<li>Pick up Dorothy                                     (4)
<li>Shopping                                            (5)
</ol>                                                   (6)
<!-This is a numbered listd of item-->
<ul>                                                    (7)
<li>Potatoes                                            (8)
<li>Tomatoes                                            (9)
<li> Cucumbers                                          (10)
</ul>                                                   (11)
<dl>                                                    (12)
<dt>Blaberus</dt>                                       (13)
<dd>A kind of cockroaches</dd>                          (14)
<dt>flabbergasted</dt>
<dd>Amazed</dd>
</dl>                                                   (15)
```

1. The tag **<ol>** indicates the beginning of an ordered list of items. Each item in the list appears on one line with a number preceding the item.

2–5. The tag **<li>** indicates the beginning of an item in the list. Since the list is an ordered one, a number precedes the item.

6. The tag **
** indicates the end of the ordered list.

7. The tag **<ul>** indicates the beginning of an unnumbered list of items. The items forming the list will appear each on a line with a bullet preceding each one.

8–10. The same tag **<li>** is used to indicate the items in this type of lists. In this case, a bullet precedes each item.

11. The tag **
** indicates the end of the unnumbered list.

12. The tag **<dl>** indicates the beginning of a definition list. A definition is formed of a term and its definition.

13. The tag **<dt>** indicates the term to define.

14. The tag **<dd>** indicates the definition of the term. The definition appears in a text that is indented with respect to the defined term.

15. The tag **</dl>** indicates the end of the definition list.

Programming Example 4: The Lang Attribute

You should use the **<lang>** attribute to specify the language of the page th at you are writing if you want your page to be compa tible with different technologies that might need to be aware of the language of your page. Sometimes, you might need to specify different languages within the page. For this, you can apply the **<lang>** attribute to elements of the page r ather than the whole page, as shown in the example below where each of the items is in a different language.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title> Greetings </title>
</head>
<body>
<ol>
<li lang="en" xml:lang="en">hi</li>                    (1)
<li lang="it" xml:lang="it">ciao</li>                  (2)
<li lang="es" xml:lang="es">hola</li>                  (3)
</ol>
</body>
</html>
```

1. The **lang** attribute specifies that this list item is in English. The attribute **xml:lang** has the same purpose but is used whenever you have a DOCTYPE declaration.
2. The **lang** attribute on this line specifies that the list item is in Italian.
3. The **lang** attribute on this line specifies that the list item is in Spanish.

Note: The **lang** attribute is used to allo w the browsers to adjust their display according to the language specified (the br owser might use different quotation marks according to the indicated language, for example). It also allows the users of assistive technology to hear the co rect pronunciation of the wor ds. If the whole page was in S panish, for example, you would add the **lang** attribute to the **<html>** tag. The attribute cannot be applied to all X HTML elements, though. For example, it cannot be applied to **applet, base, frameset**, among others.

Programming Example 5: Forms

Forms allow you to create interactive web pages. A form contains elements, such as text fields, buttons, menus, etc. The example below shows a form that allows a user to identify herself and enter comments about a hotel room.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>
<title> Hotel and Feedback </title>
</head>
<body>

<form method="get" action="#">                                          (1)
<fieldset>                                                              (2)
<legend>Service Feedback Form</legend>                                  (3)
<p><label for="name">First Name: <input type="text" id="name"
name="name" size="20" /></label></p>                                    (4)
<p><label for="family_name">Family Name: <input type="text"
id="family_name" name="family_name" size="20" value="Value in here" />
</label></p>                                                            (5)
<p><label for="pwd">Password: <input type="password" id="pwd"
name="pwd" size="20" /></label></p>                                     (6)

<fieldset>
<legend>Checking for age</legend>
<p>To what age range do you belong?<br />
<label for="age1">
<input type="radio" id="age1" name="xhtml" /> 18-29                     (7)
</label><br />
<label for="age2">
<input type="radio" id="age2" name="xhtml" /> 30-49                     (8)
</label><br />
<label for="age3">
<input type="radio" id="age3" name="xhtml" /> 50-65                     (9)
</label></p>
</fieldset>


<fieldset>
<legend>Options</legend>
<p>Check the options below that you had in your suite:<br />
<label for="smoking">
<input type="checkbox" id="smoking" name="smoking" value="smoking"
/> Non-smoking                                                          (10)
</label><br />
<label for="cable">
```

```
    <input type="checkbox" id="cable" name="cable" value="cable"
checked="checked" /> Cable                                              (11)
    </label><br />
    <label for="internet">
    <input type="checkbox" id="internet" name="internet"
value="internet" /> Internet Connection                                 (12)
    </label>
    </p>
    </fieldset>

    <fieldset>
    <legend>Submission</legend>
    <p>
    <label for="submit">
    Submit Form<br />
    <input type="submit" id="submit" value="Submit" />                  (13)
    </label>
    </p>
    <p>
    <label for="reset">
    Reset all fields<br />
    <input type="reset" id="reset" />                                   (14)
    </label>
    </p>
    </fieldset>

    </form>
    </body>
    </html>
```

1. The **<form>** tag indicates the beginning of the form.
2. The tag **<fieldset>** indicates the beginning of a section of fields in the form.
3. The tag **<legend>** indicates a legend that shows on the form. It is normally used to describe the form contents and its use.
4. This line displays the string *First Name:* and then an input field for the user to input his/her first name. The attribute **size** specifies the number of characters that the input can contain.
5. This line is similar to the previous one, but it shows that you can display a default text in the input field (in this case, the text is *value in here*).
6. On this line, the type of the input is **password**; hence, the input will not appear as the user is typing it, but will be replaced with dots.

7–9. These lines display three radio buttons. The user can select one at a time.

10–12. These lines display three checkboxes. The user can select one or more at a time.

13. This line displays a submit button. When the button is clicked, all the data is saved.

14. This line displays a reset button. When this button is clicked, all fields are re-initialized and all entered data is deleted.

DHTML

DHTML stands for Dynamic HTML and is used to design in teractive web pages. It combines HTML(HyperText Markup Language), Documen t Object M odel(DOM), Cascading Style Sheets(CSSS), and JavaScript. The three major uses of DHTML are: dynamic element positioning, style modification, and event handling. One major problem with DHTML is the incompatibility between browsers (for example, Netscape and Internet Explorer). The examples that we show below work with Internet Explorer.

Programming Example 6: Positioning of Elements

DHTML allows any element to be (re)positioned when the user is in teracting with the page. The example below displays a picture and a button on the page. If the user clicks on the button, the picture moves to the right until it hits a specific location, and then it is repositioned in its initial location.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>

<script>
function moving_right(){
if (picture.style.pixelLeft<1000)
      picture.style.pixelLeft+=500
else
      picture.style.pixelLeft=0;
}
</script>
</head>

<div id="picture" style="position:relative">                    (2)
<img src="Grendizer.jpg">
</div>
```

```
<form>
<input type="button" value="Move" onClick="moving_right()">        (3)
</form>

</body>
</html>
```

1. The function **moving_right()** checks for the leftmost pixel of the element with id equal to picture (i.e., the image specified in the tag on line (2)). If the leftmost pixel is less than 1,000, the picture is moved by 500 pixels to the right; otherwise, the picture is relocated at pixel 0 (leftmost border of the window).
2. **div** can contain any HTML element within its opening and ending tags. In this case, a div encompasses a single image. It assigns to it an id and a style where the position is defined to be relative.
3. This line displays a button with the word *Move* on it. The tag includes the **onClick** event handler, which specifies that the function **moving_right()** should be executed in case the user clicks on the button.

Programming Example 7: Handling Events

The example below displays a menu that is pulled out when the user clicks on the text *Menu*. It combines CSS and JavaScript and shows how to handle events.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<html>
<head>

<style>                                                              (1)
body{font-family:Calibri;}
a{color:brown;text-decoration:none;font:bold}                        (2)
a:hover{color:red}
td.menu{background:lightyellow}

table.nav                                                            (3)
{
background:black;
position:relative;
top:0px;
left:-120px;
}
</style>
```

```
<script type="text/JavaScript">                                    (4)
var i=-120;
var speed=3;

function showmenu()
{
intShow=setInterval("appear()",15);
}

function appear()
{
if (i<-12)
   {
   i=i+speed;
   document.getElementById('Menu').style.left=i;
   }
}
</script>
</head>

<body>
<table id="Menu" class="nav" width="150" onclick="showmenu()">    (5)
<tr><td class="menu" ><a href="info.html" id='msg'>Personal Data
</a></td>
<td rowspan="5" align="center" bgcolor="lightblue">M<br />E<br />
N<br />U</td></tr>                                                 (6)
<tr><td class="menu" ><a href="CV.pdf" >CV</a></td></tr>
<tr><td class="menu" ><a href="professional.html" >Professional
Experience</a></td></tr>
<tr><td class="menu" ><a href="langs.html">Languages Spoken</a>
</td></tr>
</table>
</body>
</html>
```

1. This is the beginning of a style section. It defines the style to apply to the body of the page and the table.
2. Ifthe **text-decoration:none** is not specified, the hyperlinks appear underlined.
3. This section of the CSS applies to the table.
4. This is the beginning of the JavaScript.
5. This defines a table with an event that is triggered when the user clicks the mouse. On this event, the function **showmenu()** is invoked.
6. This shows how to create a column that spans multiple rows.

PROBLEMS

1. Design a web page that shows your picture when you were 10 years younger. If you click on the picture, it displays your picture today.
2. Design a web page that shows your school calendar. Have some time slots appear in gray to indicate that no events are occurring then.
3. Design a web page that shows a menu that pulls down from the top of the screen. The menu has a link to some popular search engines.
4. Design a web page that simulates a calculator (hint: use forms for this task).
5. Modify the page that you have in problem number 2 above to make the events change color as you move over them with the mouse.

CHAPTER TEN

Instant JavaScript

JavaScript is used to cr eate dynamic XHTML content. To test your learning of J avaScript, all you need is a text editor and a web br owser. You can write JavaScript code in script blocks, in XHTML tags, or in external JavaScript files to which you give the extension **.js.** The first example below shows how to write JavaScript to display the current date and time in the web page. The code is written inside XHTML tags.

PROGRAMMING EXAMPLE 1: A FIRST PROGRAM

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<!-- This is a comment -->
<head>
<title> A simple page with Javascript</title>
</head>
<body>
<script language="javascript">                                    (1)
document.write(Date());                                           (2)
</script>                                                         (3)
</body>
</html>
```

1. The tag **<script>** indicates the beginning of script code. The **language** attribute indicates to the browser that the scripting language that follows is JavaScript. As we mentioned earlier, it is possible to write JavaScript code in an external file and

invoke the file from within the XHTML file. If we wanted to do this, assuming the code is written in a file called my_code.js, then the script tag here would be `<script language=javascript" src=my_code.js">`.

2. **document.write()** forwards the output to the web page, while the method **Date()** displays the current date and time.
3. The tag **</script>** indicates the end of the JavaScript code.

PROGRAMMING EXAMPLE 2: VARIABLES AND TYPES

In JavaScript, a variable name can contain letters, numbers, the dollar sign, or the underscore character, but it must start with a letter or an underscore. Examples of valid variable names are: `name, salary, First_Name, month_12, _increment,` etc. Examples of invalid variable names are: `1stmonth, double*increment, office#`, etc. Note that JavaScript is case sensitive. It is relatively loose when it comes to types . For example, in JavaScript, a number is not declared as a float, an integer, or any other type. Moreover, in the same variable, you can store a number and then a string or a boolean.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<!-- This is a comment -->
<head>
<title> A simple test for variables</title>
</head>
<body>
<script language="JavaScript">
var name="Jimmy";                                                    (1)
var age=20;                                                          (2)
document.write("Hello\n");                                           (3)
document.write("My name is "+name+" and I am "+age+"years old");     (4)

</script>
</body>
</html>
```

1. Variable **name** is assigned the string *"Jimmy"*. Similar to any other language, a string can have any character. Notice the semi-colon at the end of a statement. Note also that strings can be enclosed in double or single quotes.
2. Variable **age** is assigned the number 20.

3. This statement displays the string *hello* on the screen and adds a new line character to force the next output to a new line.
4. This statement displays the string *My name is Jimmy and I am 20 years old* to the screen. Notice that variable **name** is written outside the quotes since it is the value assigned to it that we are interested in.

The table below summarizes the special characters in JavaScript and how to include them in strings.

Special character	Effect
\\	\
\"	\"
\'	'
\b	back space
\f	form feed
\n	new line
\r	carriage return
\t	Tab

PROGRAMMING EXAMPLE 3: USING JAVASCRIPT TO FORMAT A PAGE

The code below shows how JavaScript can be used to make page modifications easier. The code starts by defining some variables that will be used to format the pages or insert specific headings and images. If later on, you want to modify the font format for example, all you have to do is to make the modifications to the **text_style** variable and they will be reflected throughout the document. If you want to modify the image to display, you change the value assigned to **image_reference** and the modification is reflected everywhere in the page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<!-- This is a comment -->
<head>
<title> A Formatted page</title>
</head>
```

```
<body>
<script language="JavaScript">
var heading="<h1> This is my personal page</h1>";
var text_style="<font face="verdana" color="red" size="20">";

var image_reference="<img src="earth.gif">;
document.write(heading);
document.write(text_style+"My Name is Edouard"+</font>);
document.write(image_reference);
</script>
</body>
</html>
```

PROGRAMMING EXAMPLE 4: FUNCTIONS

The code below defines a function **summation**, which returns the sum of the two parameters passed to it. The example also shows how to invoke it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<title> Testing Functions</title>
</head>
<body>

<script language="javascript">
function summation (x,y)                                    (1)
{
 z=x+y;
 return z;
}
document.write(summation (30,98));                          (2)
</script>

</body>
</html>
```

1. The function header always starts with the identifier **function**. Similar to variables, the parameters are not declared to be of any type.
2. This statement invokes the function **summation**, passes to it two values 30 and 98, and displays the result returned by the function on the screen.

Note: If you want the function to be executed, but you do not want to use its return value, you can use the unary operator **void**. The syntax for using **void** is: `void (function_name())`.

When you do this, the return value of the indicated function is ignored (it is not even displayed by the browser).

PROGRAMMING EXAMPLE 5: SCOPE

Scopes in JavaScript are interesting. Local variables take precedence over global variables. Hence, in the example code below, the variable a is declared twice. The declaration inside method **a** takes precedence, as is shown below, and the value printed by the code below is 20. The same goes for variable **b**.

```
var a = 10;                                                    (1)
function f() {
      var a =20;                                               (2)
      document.write(a);                                       (3)
}
f();

var b=90;                                                      (4)
function test()
{
      var b=0;                                                 (5)
      function inner_test()
      {
           var b = 8;                                          (6)
           document.write(b);                                  (7)
  }
  inner_test();
  }
  test();                                                      (8)
```

1. The variable **a** is declared globally in this statement.
2. This statement declares another variable **a** local to function **f**. This variable is seen inside the function. However, if the identifier **var** was omitted in both statements (1) and (2), the same variable **a** would have been accessed in the function.
3. This statement prints the value of **a** that is local to the function (i.e., **20**).
4. This statement declares another global variable **b**.
5. This statement declares a variable **b** that is local to method **test**.
6. This statement declares yet another **variable b** that is local to method **inner_test**.
7. This statement accesses the value in **b** that is local to **inner_test** (i.e., **8**).
8. This statement invokes the function **test,** which invokes the function. **inner_test.** The latter prints the value of the variable **b** local to it (i.e., **8**).

Note: All variables declared in a function are accessible throughout the function, even at the statements that appear before their declaration. This is not the case in C or C++.

The following table shows the most popular JavaScript operators.

Operator	Effect
<, <=	Less than, less than or equal
>, >=	Greater than, greater than or equal
==, !=	Checks for equality or inequality
&, &&	Bitwise AND, Logical AND
|, ||	Bitwise OR, Logical OR
^	Bitwise XOR
=	Assign
+=, -=,*=,/=, &=, |=, ^=	Performs the operator then assigns the result
+,-,*,/,%	Arithmetic operators
++,--	Increment by 1, decrement by 1
Typeof	Returns data type
New	Create a new object

PROGRAMMING EXAMPLE 6: CONDITIONALS (THE IF-STATEMENT)

The syntax for the conditionals in JavaScript is very similar to the one in JAVA. The example below assigns to the variable **age** the value **25** and then performs some checks on the value of **age**. It goes through the **else if** statement and prints *You are above age limit.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<title> Conditionals </title>
</head>
<body>
<script language="javascript">
var age = 25;                                                    (1)
if( age < 18 ){                                                  (2)
    document.write("<i> You are below age limit</i>");
}else if( age >18 ){                                             (3)
    document.write("<b>You are above age limit</b>");            (4)
}else{   document.write("<b> Age limit</b>");}                   (5)

</script>
</body>
</html>
```

1. Declares variable **age** and stores 25 in it.
2. The **if**-statement evaluates the condition (**age < 18**) since it returns false, the next statement is skipped.
3. The **else-if** statement evaluates the condition (**age >18**) since this returns true, the next statement is executed.
4. This statement prints the sentence ***You are above age limit*** to the screen. The tags **** and **** prints the text in bold.
5. This statement is not executed. If statement (4) had evaluated to false, the flow of execution would have been redirected to the **else**-statement, and this statement would have been executed printing *Age limit* instead of what is printed by the statement in (5).

Note: The **else** and the **else-if** sections are optional in JavaScript.

PROGRAMMING EXAMPLE 7: CONDITIONALS (THE SWITCH-STATEMENT)

The **switch** statement in JavaScript is very similar to the one in JAVA. It redirects the flow of execution to one of several paths according to some value. Unlike JAVA, the value can be an integer, a floating-point number, a string, a char, or a boolean. The code below prints the string *Invalid* to the screen.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<title> Conditionals </title>
</head>
<body>
<script language="javascript">
var a = 70;
      switch(a)                                                    (1)
{
      case 10:                                                     (2)
           document.write("Ten\n");
           break;                                                  (3)
      case 20:
           document.write("Twenty\n");
           break;
      case 30:
           document.write("Thirty\n");
           break;
```

```
        default:
             document.write("Invalid\n");                    (4)
}
</script>
</body>
</html>
```

1. The switch statement evaluates the variable **a**.
2. The variable **a** is compared to 10, since the values are not equal, the statements that form the body of this condition are not executed, and comparison is redirected to the next case.
3. Notice the use of the **break** statement inside each case of the switch statement.
4. Since none of the cases match, the default one is assigned, and the statement prints the string Invalid to the screen.

Note: It is very important to have the **break** statement at the end of each case; other wise, the flow of execution will continue through the next case instead of jumping out of the switch.

PROGRAMMING EXAMPLE 8: LOOPS AND LABELS

JavaScript provides four different types of loops n amely **while**, **for**, **do,** and **for/in** loops. The first three have the same syntax as the respective ones in JAVA, but the **for/in** loop is specific to JavaScript. The example below shows all different kinds of loops. The first three are all equivalent. They print the integer numbers 10 down to 1. The fourth loop iterates through an array and prints the value stored in each location in the array. The last loop shows how labels can be used to break out of loops and continue execution at particular points in the code. Labels were only introduced as of JavaScript 1.2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<title> Loops</title>
</head>
<body>
<script language="javascript">
var a = 10;
while (a>=1)                                                 (1)
       {    document.write(a+"\r");
            a--;}
for (a=10; a>=1; a--)                                        (2)
            document.write(a+"\r");
```

```
do
            {document.write(a+"\r");
            }while(a>=1);                                          (3)
var my_array=[10, "John", true, 78.4, 'c']                         (4)
for (value in my_array)                                            (5)
            {document.write(my_array[value]+"\r");}
var c=1;
outerloop:                                                         (6)
while (c < 10)
{       document.write(c);
            c++;
            var k = 4;
            while (k <= 10)
            {
                  document.write(k);
                  k++;
                  if (c == 7)
                        {break outerloop;}
            }
}
document.write("end");
</script>
</body>
</html>
```

1. The condition **a >= 1** is evaluated, and if it is true, the body of the loop is executed until the condition evaluates to false.
2. This *for-loop* is equivalent to the previous *while-loop*. It uses **a** to count from 10 down to 1, and the body of the loop is executed as long as the condition **a>=1** evaluates to true.
3. The *do-loop* is equivalent to the previous two loops. The difference between a *do-loop* and a *while-loop* is that the condition is evaluated at the end each iteration, and hence, the body of the loop is executed at least once.
4. This statement initializes an array of five elements. Notice that different types can be stored in the array at the same time. We will see more details about arrays further in the chapter.
5. The *for/in loop* uses **value** as an iterator through the array. **value** receives the index of each location in the array. The general syntax for the **for/in** loop is: `for (variable in object)`. Each time, **variable** receives the name of an object property (as a string).

6. **outerloop** is a label. A label in JavaScript should mark a block of statements (an if-statement, a loop, or any block of statements enclosed within curly brackets). The name of the label is used later in a **break** statement to indicate which block to exit.
7. When **c** becomes equal to 7, this statement redirects the flow of execution out of the block labeled **outerloop**, and the next statement prints the string *end* to the screen.

Note: The **continue** statement can also have a label. It restarts the loop at a new iteration. However, the **continue** statement can only be used in the body of a **while**, **for**, or **for/in** loop.

PROGRAMMING EXAMPLE 9: EVENTS

Loading a page in the browser, clicking the mouse, resizing a window—all of these actions and many others cause events. Events are part of DOM Level 3 and are associated with HTML elements. You handle them using event handlers. These are keywords that identify the events and trigger JavaScript code. Event handlers are mostly used in three locations: in form elements, link tags, and the opening **<body>** tag. The example below shows two buttons with different labels. If the user clicks on the first button, the code pops up a window with the message *You clicked the left mouse button.* If the mouse is over the second button, a window pops up with the message *You are waving over the button.*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script language="javascript">                                    (1)
function display() {
   alert("You clicked the left mouse button")                     (2)
}
function wave(){
   alert("You are waving over the button")
}
</script>                                                         (3)
</head>
<body>
<input type="button" onclick="display()" value="Click" />         (4)
<input type="button" onmouseover="wave()" value="Get close" />    (5)
</body>
</html>
```

1. This indicates the beginning of a script that includes a function named **display().** When the function is invoked, a window displays the message indicated in statement (2)
2. This statement displays the message *You clicked the left mouse button* in a popup window.

3. Notice how the script itself is written in the header part of the html document.
4. This line shows a button on the screen with the label "Click" on it. The **onclick** is the event handler, which specifies that the function **display()** should be executed when the user clicks the mouse button.
5. This line shows another button with the label "Get close" on it. The **onmouseover** event handler specifies that the function **wave()** should execute when the mouse is over the button. In this case, the message *You are waving over the button* is displayed in a popup window.

The table below summarizes the most popular event handlers that you can use in JavaScript.

Event handler	Event
onload	when the page loads
onchange	when an element changes
onsubmit	when the form is submitted
onreset	when the form is reset
onselect	when the element is chosen
onkeydown	when a key is pressed on the keyboard
onkeyup	when a key is released
onclick	when the mouse button is clicked
ondblclick	when the mouse button is double clicked
onmousedown	when the mouse button is held down
onemousemove	when the mouse pointer is moved around the page
onmouseout	when the mouse pointer moves out of the element
onmouseover	when the mouse pointer moves over on an element
onmouseup	when the mouse button is released.

PROGRAMMING EXAMPLE 10: PROMPTING FOR INPUT

The example below prompts the user for a number and displays the square of the number.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script language="javascript">
var number = prompt("Enter a number","number");                    (1)
alert(" the square of " + number +" is"+( number*number));         (2)
</script>
</head>
</html>
```

1. This statement displays a prompt dialogue box with the label "Enter a number and field" for the user to type the input.
2. Assuming, the user inputs 3, this statement pops up a dialogue box displaying *the square of 3 is 9.*

PROGRAMMING EXAMPLE 11: REGULAR EXPRESSIONS

Often, you need to perform pattern-matching to search or modify expressions. For this, JavaScript provides the RegExp class. The code below uses a regular expression to search for a word in a string.

```
<html>
<head>
<title>JavaScript RegExp exec Method</title>
</head>
<body>
<script type="text/javascript">
   var phrase = "My name is Jerry. I am 18 years old";
   var word="Jerry";
   var expr = new RegExp( word, "g" );                          (1)
   var result = expr.exec(phrase);                              (2)
   if (result==word)                                            (3)
               {document.write("Word found");}
   else
          {document.write("Word not found");}
phrase.replace(/jerry/gi, "Mary");
</script>
</body>
</html>
```

1. This statement instantiates a regular expression. The first parameter of the constructor is the pattern to match (in this case, the string *"Jerry"*) and the second parameter is an optional string, which, in this case, indicates global matching.
2. This statement executes the matching and returns the value that matches the pattern. If no match is found, it returns null.
3. This **if** statement matches the value that the **exec()** method returned in statement (2) with the desired word. Since the condition is true, the next statement is executed, and *Word found* is displayed.

Note: JavaScript features are based on Perl 5 as of version 1.3. F or this, we refer the reader to the regular expression section in the Instant Perl chapter for a complete list of quantifiers and meta-characters used to make regular expressions.

PROGRAMMING EXAMPLE 12: STRINGS

Strings in JavaScript are somehow similar to str ings in JAVA. The code below declares three strings, namely **str1**, **str2**, and **str**. The last one is the conca tenation of the first two. Then, it prints the length of the third string.

```
<html>
<head>
<title>Testing Strings</title>
</head>
<body>
<script type="text/javascript">
var str1 = new String( "Hi world!");                                    (1)
var str2 = new String("My name is John.");                              (2)
var str = str1.concat(str2);                                            (3)

document.write("The length of the string is " + str.length);            (4)
document.write("I will remove my name now:"+ str2.strike());            (5)
</script>
</body>
</html>
```

1. This statement instantiate the string *"Hi world!"*.
2. This statement instantiates another string *"My name is John."*
3. The method **concat()** is used to concatenate two strings. The resulting string is assigned to variable **str**.
4. This statement outputs to the screen the length of string **str** (i.e., 25).
5. JavaScript provides methods that allow you to format the text. These are called String HTML wrappers, and they return a copy of the string wrapped inside HTML tags. The string displayed by this statement is *"I will remove my name now: ~~My name is John~~"*. Hence, **str2.strike()** returns a copy of the string in str2 wrapped inside the tags **<strike>** and **</strike>**.

The table below lists the most popular methods in the **String** class.

Method	Description
`charAt(i)`	returns the character at index i. Indices start at 0 and end at `length-1` where `length` is a property of the array.
`concat(string)`	concatenates current string with the one specified in the parameter list.
`indexOf(i)`	returns the index of the first occurrence of `i` in the current string.
`lastIndexOf(i)`	returns the index of the last occurrence of `i` in the current string.
`match(regular_expression)`	matches the current string with the indicated regular expression.
`replace(regular_expr, new_string)`	matches the current string with the regular expression and replaces the matching string with `new_string`
`search(reg_expr)`	searches for a match between the current string and the `reg_expression` passed as a parameter.
`substring(i, j)`	returns the substring in the string starting at index `i` and ending at `j-1`.
`substr(i,j)`	returns the substring of length `j` in the string starting at index `i`.
`toLowerCase()`	returns the string in lower case.
`toUpperCase()`	returns the string in upper case.
`big()`	simulates HTML <big> tag and displays the string in big font.
`blink()`	simulates HTML <blink> tag and displays the string in blinking font.
`bold()`	simulates HTML <bold> tag and displays the string in bold
`italics()`	simulates HTML <italics> tag and displays the string in italics.
`small()`	simulates HTML <small> tag and displays the string in a small font.
`sub()`	simulates HTML <sub> tag and displays the string as a subscript.
`sup()`	simulates HTML <sup> tag and displays the string as superscript.
`fontcolor(color)`	simulates HTML <font color ="color"> tag and displays the string in the indicated color string.

PROGRAMMING EXAMPLE 13: PRINTING

Java allows you to in voke the printer connected to your P C through its **window** object. The object has associated with it the **print** method. When the method is in voked, the operating system pops up the printing dialogue box, and the user can choose the printer to send the job to. The following code displays a button with the string "Printing" as a label. If the user clicks the button, the printing dialogue box pops up.

```
<head>
<script type="text/javascript">
</script>
</head>
<body>
<form>
<input type="button" value="Print" onclick="window.print()" />
</form>
</body>
```

PROGRAMMING EXAMPLE 14: ARRAYS

Arrays are objects. They h ave properties and methods. You can stor e values of differ ent types in a single ar ray. The example belo w shows how to cr eate regular and associa tive arrays.

```
<html>
<head>
<title>Testing Arrays</title>
</head>
<body>
<script type="text/javascript">
var names=["John", "Albert", "Orlando"];                          (1)
var salaries=[170000, 25000, 37000];                              (2)
document.write(names.concat(salaries)+"<BR>");                    (3)
document.write(names.join("*")+"<BR>");                           (4)
var associative_array=new Array();                                (5)
associative_array["Alice"]=30000;                                 (6)
associative_array["Theo"]=90000;                                  (7)
associative_array["Mary"]=120000;                                 (8)
document.write("Theo's sal: "+ associative_array["Theo"]);        (9)
</script>
</body>
</html>
```

1. This statement creates an array **names** and assigns three strings to its locations. The first string is stored at index 0, the last at index `names.length`.
2. This statement creates another array of three numeric values.
3. **concat()** is a method that joins two arrays into one. This statement concatenates both **names** and **salaries** and displays the following to the screen *John,Albert,Orlando, 170000,25000,37000*.
4. The **join()** method concatenates two arrays and returns a string in which the elements are delimited, with the second parameter passed to the method, in this case, a star (*). If the second parameter is omitted, the elements are delimited with a comma.
5. This statement creates an empty array. This array will be used as an associative array, i.e., one in which elements are accessed by strings.
6. This statement associates the value **30000** with the string "**Alice**".
7. This statement outputs *Theo's sal: 90000*.

The following table shows the most commonly used array methods.

Method	Description
`pop()`	removes the last element from the array and returns it.
`push(value)`	adds the indicated element to the end of the array, and returns the length of the new array.
`reverse()`	reverses the elements in the array.
`slice(i, j)`	removes from the array the elements at indices `i` through `j-1`. Returns them in another array.
`splice (i, j, element)`	removes `j` elements from the array starting at index `i`. Replaces them with the indicated element. If the last parameter is omitted, no replacement occurs.
`sort()`	sorts the array in alphabetical order.

PROGRAMMING EXAMPLE 15: THE MATH OBJECT

JavaScript provides the **Math** object to perform some calculations. It defines constants and has some very useful methods. The code below prompts the user for an angle value and returns the cosine, sine, and tangent of the angle.

```
<html>
<head>
<title>Testing the Math Object</title>
</head>
<body>
<script type="text/javascript">
```

```
var angle=window.prompt("Enter a value for an angle:");          (1)
var cos=Math.cos(angle);                                         (2)
var sin=Math.sin(angle);                                         (3)
var tan=Math.tan(angle);                                         (4)
window.alert("cosine =" +cos);                                   (5)
window.alert("sine = "+sin);                                     (6)
window.alert("tangent = "+tan);                                  (7)
</script>
</body>
</html>
```

1. This statement prompts the user to enter the value for an angle.
2. **Math.cos()** computes and returns the cosine of the angle. The statement assigns it to variable **cos**.
3. **Math.sin()** computes and returns the sine of the angle. The statement assigns it to variable **sin**.
4. **Math.tan()** computes and returns the tangent of the angle. The statement assigns it to variable **tan**.
5–7. These statements pop up dialogue boxes displaying the cosine, sine, and tangent of the angle.

The following table shows the most popular methods the **Math** object has, and the next one shows some useful constants defined in this object.

Method	Description
`abs(i)`	returns the absolute value of `i`
`acos(i)`	returns the arccosine of `i` in radians
`asin(i)`	returns the arcsine of `i` in radians
`atan(i)`	returns the arctangent of `i` in radians
`ceil(i)`	returns the smallest integer that is greater than or equal to i.
`cos(i)`	returns the cosine of `i` (`i` is the value of an angle in radians)
`exp(i)`	returns E^i where `E` is the base of the natural logarithm
`log(i)`	returns the natural logarithm base `E`) of a `i`
`pow(i,j)`	returns `i` to the power `j`
`random()`	returns a random number between 0 and 1
`round(i)`	returns the value of `i` rounded to the nearest integer
`sin(i)`	returns the sine of `i` (`i` is the value of an angle in radians)
`tan(i)`	returns the tangent of `i` (`i` is the value of an angle in radians)
`sqrt(i)`	returns the square root of `i`.

Property	Description
E	value of the Euler's constant which is 2.7182…This is also the base of the natural logarithms
LN2	Natural logarithm of 2 (0.69314…)
LN10	Natural logarithm of 10
LOG2E	Base 2 logarithm of E (0.4426…)
LOG10E	Base 10 logarithm of E (0.4342…)
PI	3.1415…
SQRT1_2	square root of ½ (0.707…)
SQRT2	square root of 2 (1.414…)

PROGRAMMING EXAMPLE 16: FORMS

A form object is cr eated using the **<form>** and **</form>** tags in the HTML documen t. To access the form, JavaScript provides you with array object in the document object, or you can name the form in the opening **<form>** tag and use the n ame to access the for m. The rest of this section assumes that you already know how to create forms. If you do not, you can refer to the XHTML chapter. The code below creates a form with two text input fields and a submit button.

```
<html>
<body>

<form name="computation_form">                                    (1)
Price: <input type="text">                                        (2)
Reduction: <input type="text"><BR>                                (3)
<input type="submit" value= "Submit">                             (4)
</form>

<script language="JavaScript">
document.write("this page has "+document.forms.length+" forms");  (5)
document.write("<BR>");
document.write("the number of elements on the form is:
"+document.forms[0].length);                                      (6)
</script>
</body>
</html>
```

1. This tag starts a form declaration. The name of the form is optional.
2. This line creates an element of type input field labeled *Price*.
3. This line creates another element of type input field labeled *Reduction*.

4. This line creates a button with the label *Submit* on it.
5. JavaScript creates an array in which it stores the forms. The forms can be accessed either through their name, as specified in the name attribute of the **<form>** tag or by the indexing through the array. This statement prints the number of forms in this page, which is 1.
6. **forms[0]** is the first form in this page (and the only one in this example). We could have referred to the form by its name (**computation_form**). This prints the number of elements in this form (three, in this case: two input fields and the button).

The table below shows some of the most useful properties of the form object.

Property	Description
`action`	the value of the action attribute in the XHTML <form> tag
`Elements`	an array including an element for each form element in an XHTML form
`Length`	the number of elements in an XHTML form
`Method`	the value of the method attribute of the XHTML <form> tag
`Name`	the value of the name attribute in an XHTML <form> tag
`Target`	the value of the target attribute in an XHTML <form> tag

PROBLEMS

1. Write a page that includes a form with a text field in which the user enters the temperature in Celsius and then the JavaScript computes the temperature in Fahrenheit and displays it in a dialogue box.

2. Write a page that prompts the user for a date of birth and computes and displays the age in terms of years, months, and days.

3. Write a JavaScript that takes two arrays of numbers and finds the maximum between the two.

4. Write a JavaScript that takes an array of numbers and displays the average and standard deviation of these numbers. Have the output formatted in different font styles.

CHAPTER ELEVEN

InstantJava Applets

Applets are normally embedded in an HTML page. To do this, you have to write the following HTML page where "*MyApplet.class*" is the bytecode of the applet you have written.

```
<html>
<head>
<title>Java Example</title>
</head>
<body>
Here you have other html contents and elements….<br>
<applet Code="MyApplet.class" width=200 Height=100>
</applet>
</body>
</html>
```

In all the programming examples below, we assume that you will write them inside the specified classes and then you have to reference the classes inside the HTML code (replacing string *"MyApplet.class"* with the appropriate class name).

PROGRAMMING EXAMPLE 1: A SIMPLE *HELLO WORLD* APPLET

The applet below displays the string *Hello World* at the *x,y* coordinates (25, 25).

```
import java.applet.*;                                    (1)
import java.awt.*;                                       (2)
public class HelloWorld extends Applet                   (3)
```

```
{
public void init()                                                    (4)
{       }

public void stop()                                                    (5)
{       }

public void paint(Graphics g)                                         (6)
    { g.drawString("Hello World!",25,25); }                           (7)
}
```

1. The package **java.applet** is required to create applets.
2. The classes in the package **java.awt** are required to paint on the screen.
3. Any class that you design as an applet must extend the class **Applet**.
4. The method **init()** is the one that is automatically invoked when the applet is launched. In this case, it is left empty. In the examples further below, we see what we can write inside this method.
5. The method **stop()** is invoked when the applet terminates. In this case, its body is left empty.
6. **paint()**is where the drawing occurs. You write here the code that you need to draw shapes on the screen. It is inherited from class **Applet** and overridden as needed.
7. **g** is an instance of class **Graphics**. This line invokes the method **drawstring()** on **g**. This draws the text *Hello World!* on the screen at the mentioned *x,y* coordinates (25, 25).

PROGRAMMING EXAMPLE 2: DRAWING SHAPES

The example below uses the Swing user interface toolkit. It extends class JApplet and draws a rectangle, a line, an oval, and filled arc on the screen. It also displays the words "*Some Shapes*" and shows how you can change colors in your applet.

```
import javax.swing.JApplet;                                           (1)
import java.awt.*;
public class ShapeDraw extends JApplet
{
public void paint( Graphics g )
{
       super.paint(g);
       g.setFont(new Font("Helvetica", Font.ITALIC, 28));             (2)
```

```
        g.setColor(Color.MAGENTA);                                    (3)
        g.drawString("Some Shapes", 40, 30);                          (4)
        g.drawRect( 90, 40, 30, 90);                                  (5)
        g.drawLine(50, 100, 150, 200);                                (6)
        g.setColor(Color.RED);                                        (7)
        g.drawOval( 100, 200, 100, 40);                               (8)
        g.setColor(Color.YELLOW);
        g.fillArc(200,150, 90, 90, 100,80);                           (9)
    }
}
```

1. **Swing** is a user interface toolkit. This line imports the class **JApplet** found in it. The class is later on extended by class **ShapeDraw**.
2. Class**Graphics** has in it method **setFont()**, which is used to set the font with particular characteristics. In this case, it sets the font to Helvetica, italics, and size 28.
3. Method **setColor()** allows you to specify the color of the font or lines to use for drawing. The color is a constant in class **Color**.
4. **drawString()** takes three parameters, a string to display in the applet window (in this case the string "*Some Shapes*"), the *x* and the *y* coordinates at which the string should be displayed. The string uses the font specified in line (2).
5. **drawRect()** takes four parameters: the first two indicate the *x* and *y* coordinates of the lower left corner of the rectangle, and the last two the width and height. If you want the rectangle to be filled, you have to use the method **filledRect()** with the same parameters instead.
6. **drawLine()** takes four parameters: the first two indicate the *x* and *y* coordinates of the starting point, and the last two those of the end point of the line.
7. This line changes the color to red.
8. **drawOval()** draws an oval. It takes four parameters: the first two indicate the *x* and *y* coordinates of the upper leftmost rectangle that surrounds the oval. The third one is the width of the rectangle, and the fourth one its height. If you want to draw a filled rather than an empty oval, you can use the method **fillOval()** with the same parameters instead.
9. **fillArc()** draws a filled arc. It takes 6 parameters. The first two indicate the *x* and *y* coordinates of the center of the arc. The next two parameters are one minus the width and the height of the area covered by the arc. The last two indicate the starting angle and the angle at which the arc extends (in degrees). If you want the arc not to be filled, you can use the method **drawArc()** with the same parameters.

The table below summarizes some of the most popular methods associated with class Graphics.

Method	Description
void `drawPolygon` (int[] xPoints, int[] yPoints, int nPoints)	draws a closed polygon. Array `xPoints[]` stores the *x* coordinates and `yPoints[]` the *y* coordinates of the lines that form it.
void `drawRoundRect`(int x, int y, int width, int height, int arcWidth, int arcHeight)	draws a round-cornered rectangle. `x` and y indicate the *x* and *y* coordinates of the lower left corner. `Width` and `height` indicate the width and height of the rectangle. `arcWidth` and `arcHeight` indicate the horizontal and the vertical diameters of the arc that forms the corner, respectively.
void `fill3DRect`(int x, int y, int width, int height, `Boolean` raised)	draws a 3D filled rectangle. `x` and `y` are the *x* and `y` coordinates of the rectangle. `width` and `height` are self-explanatory; `raised` specifies whether the rectangle should appear as raising above the surface or sinking into it.

PROGRAMMING EXAMPLE 3: EVENTS AND LISTENERS

Often, events correspond to occurrences from the environment, such as the user pressing a key on the keyboard or clicking the mouse, a page being refreshed, etc. In JAVA, an event is an object, and the Java standard class library has classes that represent the most popular events. A listener is an object that waits for events to occur. When an event occurs, the listener deals with it. In the example below, we show an applet that displays a message on the screen telling the user when the mouse enters the applet area, when it exits it, and when the user presses and releases the mouse button.

```
import java.applet.*;                                              (1)
import java.awt.event.*;
import java.awt.*;

public class MyMouseListener extends Applet
                                  implements MouseListener{        (2)
      private String event_name="";                                (3)
      public void init()                                           (4)
      {addMouseListener(this);}                                    (5)

       public void  paint(Graphics g)
      {g.setColor(Color.blue);
        if (event_name.equalsIgnoreCase("release"))                (6)
            g.drawString("You released the mouse button", 50, 20);
        else if (event_name.equalsIgnoreCase("enter"))             (7)
            g.drawString("You entered the applet area", 50, 20);
```

```
        else if (event_name.equalsIgnoreCase("press"))              (8)
            g.drawString("You pressed the mouse buttton", 50, 20);
        else if (event_name.equalsIgnoreCase("exit"))               (9)
            g.drawString("You left the applet area", 50, 20);
    }

    public void mouseEntered (MouseEvent event)                     (10)
      {event_name="enter";
       repaint();}
    public void mousePressed (MouseEvent event)
    {event_name="press";
    repaint();}
    public void mouseReleased (MouseEvent event)
    {event_name="release";
    repaint();}
    public void mouseExited (MouseEvent event)
    {event_name="exit";
    repaint();
    }
    public void mouseClicked (MouseEvent event)                     (11)
     {/*Although  empty, this method is still required*/}
}//end of class MyMouseListener
```

1. These packages are needed to work with applets and events.
2. Because **MyMouseListener** is an applet, it should extend class **Applet**. Because you want to implement listeners that handle events occurring from the mouse, the class has to implement the interface **MouseListener**.
3. We use the string **event_name** to keep track of which mouse event happened.
4. **init()** is the first method that gets executed when the applet starts.
5. **addMouseListener()** adds a listener to the applet, in particular, one that awaits mouse events. It passes the reference **this** to the mouse listener, specifying that is waiting for events from this particular applet.

6–9. The string in **event_name** is checked. According to the value that it has been assigned by the methods below, an appropriate message is displayed in the applet telling the user what the event was.

10. This method is declared in the interface **MouseListener**. Since class **MyMouseListener** implements the interface, it has to implement this method. In this case, it assigns the string **"pressed"** to the string **event_name**. Then it calls the method **repaint()** to start the **paint()** method again and has the applet accept further events.

11. **mouseClicked()** is left empty because we do not want the applet to do anything when the user clicks the mouse button. However, since the method is declared in **MyMouseListener**, we have to implement it in this class, so we give it an empty body.

The table below shows the methods declared in interface **KeyListener**, which is the one to implement in case we wanted to respond to keyboard events.

Method	Description
`void keyPressed(KeyEvent e)`	invoked when the user presses a key on the keyboard. `KeyEvent` indicates that a keystroke occurred in a component.
`void keyReleased(KeyEvent e)`	invoked when a key is released.
`void keyTyped(KeyEvent e)`	invoked when a key has been typed.

PROGRAMMING EXAMPLE 4: IMAGES

Including images in your applets is very easy and straightforward. The example below displays the image from the specified **.jpg** file (**Grendizer.jpg**).

```
import java.awt.*;
import java.applet.*;

public class ImageApplet extends Applet
{    private Image image;                                          (1)

     public void init()
     {URL url=getCodeBase();                                       (2)
      image= getImage(url,"Grendizer.jpg");                        (3)
      }
     public void paint(Graphics g)
     { g.drawImage(image,20,20,this);}                             (4)

}
```

1. Object**image** is used to reference the image that we want to display in our applet.
2. Method **getCodeBase()** returns the URL of the directory that contains this applet. This assumes that the figure is in the same location as the applet.
3. Method **getImage()** takes two parameters: the URL that we got in line (2) and the name of the file that has the image in it.
4. **drawImage()** takes four parameters: the image to display, the *x* and *y* coordinates of the left corner of the image, and a reference to an observer object to be notified when the image is loaded (in this case, it is the applet).

PROGRAMMING EXAMPLE 5: GUIs

Java applets are often used to pr ovide a gr aphical user in terface. The example belo w shows how to create a text area where the user can type, two radio buttons, and a submit button.

```
import java.awt.*;
import java.applet.*;

public class GuiApplet extends Applet
{
 private TextArea text_area;                                          (1)
 private Button button;                                               (2)
 private CheckboxGroup radio_group;                                   (3)
 private Checkbox radio_button1;                                      (4)
 private Checkbox radio_button2;

 public void init()
 {
      setLayout(null);                                                (5)
      text_area = new TextArea("Enter comments here", 20, 80);        (6)
      button = new Button("Submit");                                  (7)
      radio_group = new CheckboxGroup();                              (8)
      radio_button1 = new Checkbox("Too cold", radioGroup,false);     (9)
      radio_button2 = new Checkbox("Too hot", radioGroup,true);       (10)
      text_area.setBounds(20,20,300,250);                             (11)
      button.setBounds(25,370,50,30);                                 (12)
      radio_button1.setBounds(10,300,100,30);                         (13)
      radio_button2.setBounds(120,300,100,30);                        (14)
      add(text_area);                                                 (15)
      add(radio_button1);                                             (16)
      add(radio_button2);                                             (17)
      add(button);                                                    (18)
 }
}
```

1. **TextArea** is a class that allows you to create text areas where the user can enter text. In this example, it is used to create a field for the user to enter her comments.
2. **Button** is a class that allows you to create buttons.
3. A**CheckboxGroup** object is one that groups together radio buttons so that one of them is selected at a time.
4. **Checkbox** allows you to create radio buttons.

5. **setLayout(null)** tells the applet not to use the layout manager since we will specify how the objects will be laid out.
6. The constructor of **TextArea** takes three parameters. The first parameter is a string to display in the field, and the second two specify the number of rows and number of columns that form the text area.
7. A button with the string **Submit** is displayed.
8. **radio_group** is used to group checkboxes together so that only one is selected(checked) at a time.
9. **radio_button1** is the first radio button in the group. The second parameter in the method (**false**) specifies that it is, by default, unchecked.
10. **radio_button2** is the second radio button in the group. The second parameter in the method (**true**) specifies that it is, by default, checked.

11–14. **setBounds()** is used to specify where the objects should be displayed in the applet. The first two parameters specify the *x* and *y* coordinates of the beginning of the object and the second two its width and height.

15–18. **add()** is called to display the objects.

PROGRAMMING EXAMPLE 6: AUDIO

If we wanted to add a udio to the applet sho wn in Programming Example 4, the code would look as shown in the example below.

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class ImageandSoundApplet extends Applet
{
      private Image image;
      private AudioClip music;                                    (1)
      private Button stop;

     public void init()
     {
            URL url=getCodeBase();
            image= getImage(url,"Grendizer.jpg");
            music=getAudioClip(url, "song.wav");                  (2)
     }

     public void paint(Graphics g)
     {
```

```
            g.drawImage(image,20,20,this);
            music.play();                                          (3)
        }
    }
```

1. Class**AudioClip** allows you to play sounds in your applets.
2. **getAudioClip()** takes two parameters: the URL of the path where the audio found is saved and the name of the sound file (in this example, "**song.wav**"). To use **URL** class, we have to import **package java.net.***.
3. **play()** plays the sound.

PROGRAMMING EXAMPLE 7: ANIMATIONS

Animations are a series of changing images or the same one moving on the screen. The example below shows an applet that displays different pictures at a specific time interval.

```
import java.awt.*;
import java.applet.*;
import javax.swing.Timer;
import java.awt.event.*;

public class AnimationApplet extends Applet
{
      private final int DELAY =1000;
      private final String IMAGE_STEM="image";
      private final int NUMBER=3;
      private Timer timer;
      private Image[] images = new Image[NUMBER];
      private Image image;
      private int i=0;
     public void init()
     {      timer = new Timer(DELAY,new ReboundActionListener()); (1)
            timer.start();                                         (2)
      }
      public void paint(Graphics g)
     {
            g.drawImage(images[i],10, 10,this);
      }
      public class ReboundActionListener implements
ActionListener                                                     (3)
      {
            public void actionPerformed (ActionEvent event)
```

```
                {
                i++;
                if(i<0 || i >images.length-1)                          (4)
                       i=0;
                images[i]=getImage(getCodeBase(), IMAGE_STEM+i+"
.jpg");                                                                (5)
                repaint();                                             (6)
                }
           }//end of RebounActionListner
  }//end of AnimationApplet
```

1. Class**Timer** is used to generate action events at regular intervals (specified by **DELAY**). The **ReboundActionListener** handles these events.
2. **timer.start()** starts the timer and, hence, the generation of events.
3. **ReboundActionListener** is an inner class. It is instantiated with the creating of a timer object.
4. This conditional makes sure we do not get out of the boundary of the array.
5. The image is stored in array **images**.
6. **paint()** method is invoked again to display the image at array location **i**.

Note: If you want to animate the same image and have it move around the applet window, you can use the same image name and specify different coordinates each time you invoke the method **paint()**.

The table below shows some of the most popular methods in class Timer (found in **javax.util** package).

Method	Description
`void addActionListener (ActionListener l)`	adds an action listener to the timer.
`void cancel()`	terminates the timer.
`void schedule(TimerTask task, Date time)`	schedules `task` to be executed at the specific time.
`void schedule (TimerTask task, long delay)`	schedules `task` to be executed after the specified delay.
`void scheduleAtFixedRate(TimerTask task, Date time, long m)`	schedules `task` to be executed starting at the specified time, repeating but waiting `m` milliseconds between successive executions.