Linux, Shell Programing and Perl

by
Willi-Hans Steeb, Gert Cronje and Yorick Hardy
International School for Scientific Computing

Contents

Preface

Linux is a Unix-like operating system. It is the result of an international collaboration, directed by Linus Torvalds, the original author. It's free - no licensing fee payable to anybody. The Linux kernel is licensed under Free Software Foundation's General Public License. The source code comes with the system. It's dependent – it contains no code written by Bell Labs or its successors. Supported architectures:

Pentium	PentiumPro	Amiga	Atari
PowerPC	SUN SPARC	DEC Alpha	

Ports in progress: ARM and MIPS (SGI machines).

Standards: Linux implements a superset of the POSIX standard.

Historical Overview
Linux did not appear out of thin air. It is the outcome of years of computer system development:

- **Multics – 1964-1965**

  – General Electric, Bell Telephone laboratories and Project MAC at MIT.

  – Security rings,

  – Multiple command interpreters (or shells),

  – A file system for secondary storage,

  – Dynamic libraries.

  – High level language (PL/I).

- **Unix – 1969**

  – Bell Telephone Laboratories.

  – Input/output redirection. (e.g. `command1 > temp_file`)

  – Pipes - i.e. the output form one program is the input for another.

  In Unix the programs execute simultaneously, the pipe implmentation in DOS does not support simultaneous execution. (e.g. `command1 | command2`)

  – Hierarchical file system,

  – SUID flag. A process started by a executable file takes on the identity and privileges of the owner of the file.

Unix Philosophy

Unix prescribes a unique philosophy of how system tools should be designed:
– Make each program do one thing well. Rather write a new program than complicate old programs by adding new ”features”.
– Expect the output of any program to become the input of any other program - Don't insist on interactive input.
– Design and build software to be tried early. Do not hesistate to throw out clumsy parts and rebuild them.
– Use tools to lighten a programming task, even if we have to detour to build the tools.
The reslt of the above is that a typical Unix system has hundreds of obscure programs, commands and tools. Unix has been described as an operating system kernel with twenty years of nifty add on utilities.

T.H.E., TUNIS, Xinu and Minix.

– Teach operating systems design.
– Minix was written for the cheap Intel 8086 platform.
– Minix prompted the development of Linux also on the Intel 80386.

Other contributions.

A number of other systems have made contributions which have become an essential part of Linux and current Unix systems. These are:

– Multi-User
– Virtual Memory
– Filename expansion in command lines (globbing or wildcard expansion),
– Bit mapped display - the X-window system

Application Programs for Linux

- Motif and the Command Desktop environment
- Several commercial X-servers
- AplixWare (Office Suite)
- CorelDraw (Graphics program)
- WordPerfect(Office Suite)
- StarOffice (Office Suite - Free for non-commercial use)
- SymbolicC++

- Mathematica
- Reduce
- Maple
- Matlab
- Octave (Matlab-like package - Free)
- Scilab (Matlab-like, Excellent for control systems - Free)
- Khoros (Image processing and visualization - Free)
- PV Wave (Visualisation)
- Microstation (CAD program)
- NexS (Internet enabled spreadsheet)
- Netscape (WWW browser)
- The GNU Project
- SUN Java Developer Kit
- Oracle.

Program Development

Available programming languages include:

C	ADA	Scheme
C++	APL	Eiffel
Objective C	Basic	Modula 2 and 3
Pascal	Logo	Java
Fortran	Smalltalk	Prolog
Forth	Perl	Rexx
Simula	Postscript	CORC and CUPL
Oberon	Common Lisp	INTERCAL
TCL/TK		

Compatibility

Binary compatibility

Intel Linux runs several other x86 Unix binaries (for example SCO Unix). SPARC Linux runs SunOs and some Solaris binaries.

Networking

- Ethernet
- Token ring
- TCP/IP
- IPX (Novell)
- SMB (OS/2 Warp server / NT Server / Windows for Workgroups)
- Appletalk

We note that MS-DOS and OS/2 use different control characters for carriage return and linefeeds that Unix does.

We note that this difference between MS-DOS and Unix may also produce problems when printing. A well known error is the staircase effect. To fix this problem under OS/2, edit the file under OS/2 and delete and retype the frist two characters.

Linux uses libraries that store dates as 32-bit integers, which count the seconds since 1970. This counter will not overflow until the year 2038, by which time the library programmers will have upgraded the system software to store dates as 64-bit integers.

The email addresses of the author is:

`whs@na.rau.ac.za`
`steeb_wh@yahoo.com`

The web sites of the author are:

`http://zeus.rau.ac.za`
`http://issc.rau.ac.za`

Chapter 1

Linux Basics

1.1 Some Basic Commands

Here we introduce some basic command line commands in LINUX and UNIX. We notice that LINUX and UNIX commands are case-sensitive. Under DOS and Windows command line commands are not case-sensitive. The command prompt is indicated by `>` in the following. We assume the use of the Bourne shell or a compatible shell.

Every UNIX and LINUX command is ended with the pressing of the RETURN (or ENTER) key. RETURN (or ENTER) says that we are finished typing things in and are ready for the UNIX system to do its thing.

Displaying the Date and Time. The `date` Command
The `date` command tells the system to print the date and time:

```
> date
  Tues Jun 27 11:38:37 SAST 2000
>
```

This means `date` prints the day of the week, month, day, time (24 hour clock, South African Standard Time), and year. DOS and Windows also have the `date` command.

The `clock` command

```
> clock
```

provides

```
 Fri Feb 20 15:34:07 2004 -0.492223 seconds
```

Finding Out Who's Logged In. The `who` Command

The `who` command can be used to get information about all users who are currently logged into the system:

```
> who
  pat     tty29    Oct 29 14:40
  ruth    tty37    Oct 29 10:54
  steve   tty25    Oct 29 15:52
>
```

Here there are three users logged in, pat, ruth, and steve. Along with each user `id`, is listed the `tty` number of that user and the day and time that user logged in. The `tty` number is a unique identification number the UNIX system gives to each terminal.

The `who` command also can be used to get information about yourself

```
> who am i
  pat      tty29    Oct  29  14:40
>
```

`who` and `who am i` are actually the same command: `who`. In the latter case, the `am` and `i` are arguments to the `who` command.

Echoing Characters. The `echo` Command

The `echo` command prints (or echoes) at the terminal whatever else we happen to type on the line. A newline is included by default.

```
> echo this is a test
  this is a test
> echo why not print out a longer line with echo?
  why not print out a longer line with echo?
> echo

> echo one         two     three        four    five
  one two three four five
>
```

From the last example we see that `echo` squeezes out extra blanks between words. That is because on a UNIX system, it is the words that are important; the blanks (whitespace with ASCII table number 32) are merely there to separate the words. Generally, the UNIX system ignores extra blanks. Normally `echo` follows all the output with a newline. The option `echo -n` suppresses that. Another option is `echo -e` it enables interpretation of backslashed special characters in the string. `\b:` backspace, `\f:` form feed, `\n:` newline, `\r:` carriage return, `\t:` horizontal tab, `\\:` backslash. The `echo` command also exists in DOS and Windows.

The command

```
printf hello how
```

provides

```
 hello
```

but

```
printf "hello how"
```

provides

```
 hello how
```

Reporting working directory. The `pwd` Command
The `pwd` command reports the present working directory.

```
> pwd
/root
>
```

Reporting the status of currently running processes. The `ps` command
The `ps` command reports the status of currently running processes. Thus this command gives us information about the processes that are running on the system. `ps` without any options prints the status of just our processes. If we type in `ps` at our terminal, we get a few lines back describing the processes we have running:

```
> ps
  PID       TTY  TIME  COMMAND
  195       01   0:21  bash    The shell
  1353      01   0:00  ps      The ps command
  1258      01   0:10  sort    The previous sort
>
```

The `ps` command prints out four columns of information: `PID`, the process `id`; `TTY`, the terminal number that the process was run from; `TIME`, the amount of computer time in minutes and seconds that process has used; and `COMMAND`, the name of the process. The `sh` process in the above example is the shell that was started when we logged in, and it used 21 seconds of computer time. Until the command is finished, it shows up in the output of the `ps` command as a running process. Process number 1353 in the above example is the `ps` command that was typed in, and 1258 is the `sort` operation.

When used with the `-f` option, `ps` prints out more information about our processes, including the parent process `id` (`PPID`), the time the processes started (`STIME`), and the command arguments.

```
>  ps -f
   UID    PID      PPID  C   STIME      TTY     TIME  COMMAND
   steve    195       1   0   10:58:29   tty01   0:21  -sh
   steve   1360     195  43   13:54:48   tty01   0:01  ps -f
   steve   1258     195   0   13:45:04   tty01   3:17  sort data
>
```

Because processes rapidly progress in their execution path, this report is only a snapshot view of what was happening when we asked. A subsequent invocation of `ps` may give different results. No options shows a picture of the currently executing processes on our terminal. The following options are available for the `ps` command.

Item	Description
`-l`	Gives a long listing.
`-u`	Prints in user format displaying the user name and start time.
`-j`	Gives output in the jobs format.
`-s`	Gives output in signal format.
`-v`	Displays output in virtual memory format.
`-m`	Show threads.
`-a`	Shows processes of other users as well.
`-x`	Displays processes without a controlling terminal.
`-S`	Adds child CPU time and page faults.
`-c`	Lists the command name from the kernel `tast_structure`.
`-e`	Shows the environment.
`-w`	Displays in wide format. Does not truncate command lines to fit on one line.
`-h`	Does not display a header.
`-r`	Displays running processes only.
`-n`	Provides numeric output for USER and WCHAN.
`txx`	Displays only processes with the controlling `tty xx`.

The `-t` and `-u` options are particularly useful for system administrators who have to kill processes for users who have gone astray. The following columns are reported:

Column	Description
PID	The process ID.
PRI	Process priority.
NI	The Linux process nice value. A positive value means less CPU time.
SIZE	The virtual image size, calculated as the size of text+data+stack.
RSS	The resident set size. The number of kilobytes of the program that is currently resident in memory.
WCHAN	The name of the kernel event that the process is waiting for.
STAT	The process status. Given by one of the following codes:
	R Runnable
	S Sleeping
	D Uninterruptible sleep
	T Stopped or traced
	Z Zombie
	Q Process has no resident pages
TTY	The name of the controlling `tty` (terminal) for the process.
PAGEIN	The number of page faults that have caused pages to be read from disk.
TRS	The text resident size.
SWAP	The number of kilobytes on the swap device.

Environment variables. The `printenv` command
The `printenv` prints the values of all environment variables. These environment variables include the following

- `USERNAME` - our current username (account name).
- `ENV=/home/gac/.bashrc` - default file for setting environment variables for this shell.
- `HOSTNAME=issc.rau.ac.za` - hostname of the computer.
- `MAIL=/var/spool/mail/gac` - where incoming mail is placed.
- `PATH=/usr/local/bin:/bin` - search path for executable files.
- `IFS` - internal field separator.

A number of programs require certain environment variables to be set. For example PVM (parallel virtual machine) need the following:

```
PVM_DPATH=/usr/local/lib/pvm3/lib
PVM_ROOT=/usr/local/lib/pvm3
```

To set this, the following lines must be added to the `.bashrc file`

```
export PVM_ROOT=/usr/local/lib/pvm3
export PVM_DPATH=$PVM_ROOT/lib
```

There are two ways to access the value of an environment variable:

- `printenv SHELL`
- `echo $SHELL`

UNIX and LINUX on-line manual pages. The `man` command
To get on-line help for each of the various Linux commands, we can type `man`. Linux then displays, a screen at a time, any information it has on the command. If we are not sure of what command to use, we can try the `-k` parameter and enter a simple keyword that represents the topic of interest. `man` then searches through its help files (called `man` pages) for a topic that contains the keyword. Linux also provides an alias for this command called `apropos`. If we enter the command `man ls`, Linux provides help on the `ls` command, including all its parameters. The command `man -k cls` provides a listing of commands that have the word `cls` in the help file; the command `apropos cls` is the same as `man -k cls`.

```
> man ls        ask for help on the ls command
> man pvm_mytid ask for help on the pvm_mytid() function
```

To quit the online help we enter `q` or `CRTL-c`.

Clearing Terminal Screen. The `clear` command
Sometimes after filling our terminal screen with information, we want a blank screen. Under LINUX we use the `clear` command.

```
> clear
```

Under DOS, we use the `cls` command.

Typing More Than One Command on a Line
We can type more than one command on a line provided we separate each command with a semicolon. For example, we can find out the current time and also our current working directory by typing on the `date` and `pwd` commands on the same line

```
> date; pwd
  Tues Jun 27 11:41:21 SAST 2000
  /usr/pat/bin
>
```

We can string out as many commands as we like on the line, as long as each command is delimited by a semicolon.

Sending a Command to the Background
Normally, we type in a command and then wait for the results of the command to be displayed at the terminal. For all the examples we have seen thus far, this waiting time is typically quite short - maybe a second or two. However, we may have to run commands that require many minutes to execute. In those cases, we have to wait for the command to finish executing before we can proceed further unless we execute the command in the background.

If we type in a command followed by the ampersand character `&`, then that command will be sent to the background for execution. This means that the command will no longer tie up our terminal and we can then proceed with other work. The standard output from the command will still be directed to our terminal. However, in most cases the standard input will be dissociated from our terminal. If the command does try to read any input from standard input, it will be as if `CTRL-d` were typed.

```
> sort data > out &      send the sort to the background
  1258                   process id
> date                   terminal is immediately available
  Tues Jun 27 11:38:37 SAST 2000
>
```

When a command is sent to the background, the UNIX and LINUX system automatically displays a number, called the process `id` for that command. In the above example, 1258 was the process id assigned by the system. This number uniquely identifies the command that we sent to the background, and can be used to obtain status information about the command. This is done with the `ps` command.

Shutting down Linux
Linux keeps a lot of information about itself and files in memory, in areas called buffers, before writing the information to the disk. This process helps to improve system performance and control access to the hardware. If we turn the power off, this information is lost, and we can damage our file-system. Linux is an operating system that has to be told that we want to turn it off.

There are several safe ways to shut down a Linux system:

- `shutdown` - Type this command while we are logged in as root. An immediate system shutdown will be initiated.

- `reboot` - The system will shut down and reboot.
- `halt` - The system will shut down, but it will not reboot.
- `CRTL-Alt-Del` - This key sequence is mapped to the `shutdown` command. This is the easiest way to shut down our LINUX system.

`shutdown` lets us control when the shutdown takes place, and it notifies the users on a regular basis. `shutdown` safely brings the system to a point where the power may be turned off.

```
> shutdown [options] time [warning]
```

Item	Description
`time`	The time to shut down the system. Refer to the man page for a full description of the available time formats.
`warning`	A warning message sent out to all users.
`-t n`	Wait `n` seconds between sending processes the warning and the kill signal.
`-k`	Does not really shut down the system. Just sends the warning messages to everybody.
`-r`	Reboots after shutdown.
`-h`	halts after shutdown.
`-n`	Doesn't synchronize the disks before rebooting or halting.
`-f`	Does a fast reboot. Does not check any file systems on reboot.
`-c`	Cancels an already running shutdown. With this option, it is not possible to give the `time` argument, but we can enter an explanatory message on the command line that is sent to all users.

`shutdown` can only be run by the super user. Messages are sent to the users' terminals at intervals bases on the amount of time left till the shutdown. For example

```
> shutdown -r +15 "System rebooting..."
```

This shuts down the system and reboots it 15 minutes from the time the command was entered. It also sends the message `System rebooting...` to all users.

```
> shutdown -h now
```

This shuts down the system and halts it immediately.

1.2 Working with Files and Directories

1.2.1 Introduction

The LINUX and UNIX system recognizes only three basic types of files: *ordinary* files, *directory* files, and *special* files. An ordinary file is just that: any file on the system that contains data, text, program instructions, or just about anything else. Directories are described later in this chapter. As its name implies, a special file has a special meaning to the UNIX system, and is typically associated with some form of I/O.

Linux allows filenames to be up to 256 characters long. These characters can be lower- and uppercase, numbers, and other characters, usually the dash `-`, the underscore `_` and the dot `.` . They cannot include reserved metacharacters such as asterisk, question mark, backslash, and space, because these all have meaning to the shell.

The Unix file system is structured as a tree. The tree has a root directory. Every file in the system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file. Directories are themselves files that contain information on how to find other files. A directory (or subdirectory) contains a set of files and/or subdirectories. All directories have the same internal format. One bit in each directory entry defines the entry as a file `(0)` or as a subdirectory `(1)`. Special system calls are used to create and delete directories. In normal use, each user has a current directory (home directory). The user can create his own subdirectories. All user data files are simply a sequence of bytes. Hardware devices have names in the file system. These device special files are known to the kernel as the device interface, but are accessed by the user by the same system calls as other files. The filename `.` in a directory is a hard link to the directory itself. The file name `..` is a hard link to the parent directory. Linux has both absolute pathnames and relative pathnames. Absolute pathnames start at the root of the file system and are distinguished by a slash at the beginning of the pathname. Relative pathnames start at the current directory.

A process is a program in execution. Processes are identified by their identifiers which is an integer. Linux supports multiple processes. A process can create new processes.

The operating system is mostly written in C. Linux consists of two separable parts, the kernel and the system programs. The kernel provides the file system, cpu scheduling, memory management, and other operating system functions through system calls.

1.2.2 The Home Directory and Path Names

The UNIX system always associates each user of the system with a particular directory. When we log into the system, we are placed automatically into a directory called the *home* directory.

Assume our home directory is called `steve` and that this directory is actually a subdirectory of a directory called `usr`. Therefore, if we had the directories `documents` and `programs` the overall directory structure would actually look something like the figure illustrated in Figure 1.1. A special directory known as

```
/
```

(pronounced *slash*) is shown at the top of the directory tree. This directory is known as the *root*.

Whenever we are inside a particular directory (called our current working directory), the files contained within the directory are immediately accessible. If we wish to access a file from another directory, then we can either first issue a command to change to the appropriate directory and then access the particular file, or we can specify the particular file by its path name.

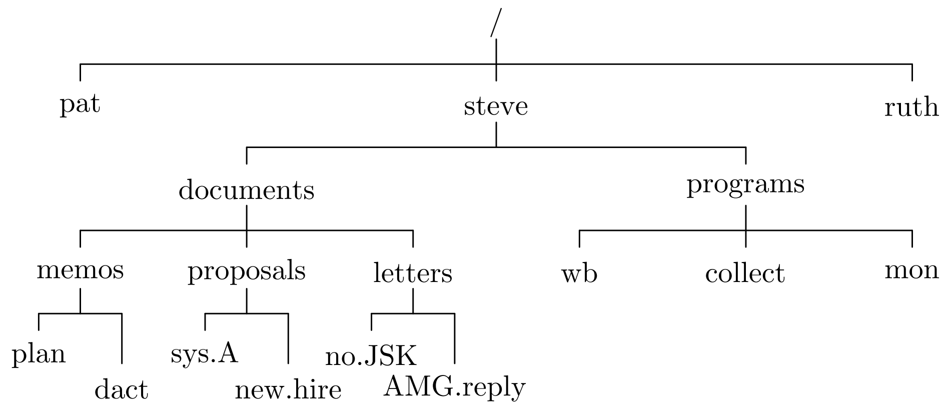

Figure 1.1. Hierarchical directory structure

A path name enables us to uniquely identify a particular file to the UNIX system. In the specification of a path name, successive directories along the path are separated by the slash character `/` . A path name that begins with a slash is known as a full path name, since it specifies a complete path from the root. So, for example, the path name

```
/usr/steve
```

identifies the directory `steve` contained under the directory `usr`. Similarly, the path name

```
/usr/steve/documents
```

references the directory `documents` as contained in the directory `steve` under `usr`. As a final example, the path name

```
/usr/steve/documents/letters/AMG.reply
```

identifies the file `AMG.reply` contained along the appropriate directory path.

In order to help reduce some of the typing that would otherwise be require, UNIX provides certain notational conveniences. Path names that do not begin with a slash character are known as relative path names. The path is relative to our current working directory. For example, if we just logged into the system and were placed into our home directory `/usr/steve`, then we could directly reference the directory `documents` simply by typing `documents`. Similarly, the relative path name `programs/mon` could be typed to access the file `mon` contained inside our programs directory.

By convention, the directory name

```
..
```

always references the directory that is one level higher. For example, after logging in and being placed into our home directory `/usr/steve`, the path name .. would reference the directory `usr`. And if we had issued the appropriate command to change our working directory to `documents/letters`, then the path name `..` would reference the `documents` directory, `../..` would reference the directory `steve`, and `../proposals/new.hire` would reference the file `new.hire` contained in the proposals directory. Note that in this case there is usually more than one way to specify a path to a particular file. Another convention is the single period

```
.
```

which always refers to the current directory.

1.2.3 Important Directories in the Linux File System

We recall that `/` is the root directory. It holds the actual Linux program as well as subdirectories.

The directory `/home` holds users' home directories. In other Unix systems this can be `/usr` or `/u` directory. The directory `/bin` holds many of the basic Linux programs. `bin` stands for binaries, files that are executable.

The directory `/usr` holds many other user-oriented directories. The directory `/usr/bin` holds user-oriented Linux programs.

In the directory `/var/spool` we find several subdirectories. `mail` holds `mail` files, `spool` holds files to be printed, and `uucp` holds files copied between Linux machines.

Linux treats everything as a file. The `/dev` directories contains devices. These are special files that serve as gateway to physical computer components. For instance if we copy to `/dev/fd0` we are actually sending data to the systems floppy disk. Our terminal is one of the `/dev/tty` files. Partitions on the hard disk are of the form `/dev/hd0`. Even the systems memory is a device. A famous device is `/dev/null`. This is sometimes called the bit bucket. All information sent to `/dev/null` vanishes - it is thrown in the trash.

The directory `/usr/sbin` holds administration files (executable files). If we enter the command `ls -l` we see that we must be the owner, root, to run this commands.

The directory `/sbin` holds files that are usually run automatically by the Linux system.

The directory `/etc` and its subdirectory hold many of the Linux configuration files. These files are usually text files and they can be edited to change the systems configuration.

The directory `/media` includes the directories `cdrecorder`, `cdrom`, `floppy`, `sda1` and `sdb1`.

1.2.4 Basic Files and Directories Commands

The UNIX and LINUX system provides many tools that make working with files easy. Here we review many of the basic file manipulation commands. The commands we describe are

`ls file(s)`	`List file(s)`
`ls dir(s)`	`List files in dir(s) or in current directory if dir(s) are not specified`
`cd dir`	`Change working directory to dir`
`cp file1 file2`	`Copy file1 to file2`
`cp file(s) dir`	`Copy file(s) into dir`
`mv file1 file2`	`Move file1 to file2 Simply rename it if both reference the same directory`
`mv file(s) dir`	`Move file(s) into directory dir`
`mkdir dir(s)`	`Create directories`
`rmdir dir(s)`	`Remove empty directory dir(s)`
`rm file(s)`	`Remove file(s)`
`cat file(s)`	`Display contents or concatenates files`
`wc files(s)`	`Count the number of lines, words, and characters in file(s) or standard input if not supplied`
`ln file1 file2`	`Link file1 and file2`
`ln file(s) dir`	`Link file(s) into dir`

These commands are execute files in the `bin` directory.

Command Options

Most UNIX and LINUX commands allow the specification of options at the time that a command is executed. These options generally follow the same format

```
-letter
```

That is, a command option is a minus sign followed immediately by a single letter.

For example, in order to count just the number of lines contained in a file named `names`, the option `-l` is given to the `wc` command (word count)

```
> wc -l names
  5 names
>
```

To count just the number of characters in a file, the `-c` option is specified

```
> wc -c names
  27 names
>
```

Finally, the `-w` option can be used to count the number of words contained in the file:

```
> wc -w names
  5 names
>
```

Some commands require that the options be listed before the file name arguments. For example

```
> sort names -r
```

is acceptable whereas

```
> wc names -l
```

is not. In general the command options should precede file names on the command line.

`ls` [*files*] - List the contents of a directory.

To see what files we have stored in our directory, we can type the `ls` command:

```
> ls
  READ_ME
  names
  first.cc
>
```

This output indicates that three files called `READ_ME`, names, and `first.cc` are contained in the current directory.

Whenever we type the command `ls`, it is the files contained in the current working directory that are listed. We can also use `ls` to obtain a list of files in other directories by supplying an argument to the command. First we get back to our home directory

```
> cd
> pwd
  /usr/steve
>
```

Now we look at the files in the current working directory

```
> ls
  documents
  programs
>
```

If we supply the name of one of these directories to the `ls` command, then we can get a list of the contents of that directory. So, we can find out what's contained in the documents directory simply by typing

```
> ls documents
  letters
  memos
  proposals
>
```

to take a look at the subdirectory `memos`, we follow a similar procedure

```
> ls documents/memos
  dact
  plan
>
```

If we specify a nondirectory file argument to the `ls` command, we simply get that file name echoed back at the terminal

```
> ls documents/memos/plan
  documents/memos/plan
>
```

There is an option to the `ls` command that tells us whether a particular file is a directory, among other things. The `-l` option provides a more detailed description of the files in a directory. If we were currently in `steve's` home directory, we obtain

```
> ls -l
  total 2
  drwxr-xr-x    5 steve   FP3725   80 Jun 25 13:27 documents
  drwxr-xr-x    2 steve   DP3725   96 Jun 25 13:31 programs
>
```

The first line of the display is a count of the total number of blocks of storage that the listed files use. Each successive line displayed by the `ls -l` command contains detailed information about a file in the directory. The first character on each line tells whether the file is a directory. If the character is `d`, then it is a directory; if it is `-` then it is an ordinary file, finally if it is `b`, `c`, or `p`, then it is a special file. An `l` indicates a link.

The next nine characters on the line tell how every user on the system can access the particular file. These access modes apply to the file's owner (the first three characters), other users in the same group as the file's owner (the next three characters), and finally to all other users on the system (the last three characters). They tell whether the user can read from the file, write to the file, or execute the contents of the file.

This means the `ls -l` shows file permissions, owner, size and last modification date.

Other command options are

```
ls -a     also list hidden files
ls *.cc   list all file ending in .cc
ls -r     reverse order
ls -d     list directories
```

`cd` *directory* - Change directory.

We can change the current working directory by using the `cd` command. This command takes as its argument the name of the directory we wish to change to.

Let us assume that we just logged into the system and were placed inside our home directory, `/usr/steve`. We know that there are two directories directly below `steve's` home directory: `documents` and `programs`. This can be verified at the terminal by issuing the `ls` command:

```
> ls
  documents
  programs
>
```

The `ls` command lists the two directories `documents` and `programs` the same way it listed other ordinary files in previous examples.

In order to change our current working directory, we issue the `cd` command, followed by the name of the directory to change to:

```
> cd documents
>
```

After executing this command, we will be placed inside the `documents` directory. We can verify at the terminal that the working directory has been changed by issuing the `pwd` command:

```
> pwd
  /usr/steve/documents
>
```

The way to get one level up in a directory is to issue the command

```
> cd ..
```

since by convention `..` always refers to the directory one level up known as the parent directory. Note the space (blank) after `cd`. The command `cd..` without the space will not work in LINUX.

```
> cd ..
> pwd
  /usr/steve
>
```

If we want to change to the `letters` directory, we could get there with a single `cd` command by specifying the relative path `documents/letters`:

```
> cd documents/letters
> pwd
  /usr/steve/documents/letters
>
```

We can get back up to the home directory with a single `cd` command as shown:

```
> cd ../..
> pwd
/usr/steve
>
```

Or we can get back to the home directory using a full path name instead of a relative one:

```
> cd /usr/steve
> pwd
  /usr/steve
>
```

Finally, there is a third way to get back to the home directory that is also the easiest. Typing the command `cd` without and argument will always place we back into our home directory, no matter where we are in our directory path.

```
> cd
> pwd
  /usr/steve
>
```

The command

```
> cd /tmp
```

change directory to `/tmp`. Note that `/` is the directory separator and not `\` as in DOS.

DOS and Windows also provide the `cd` command. The `cd ..` command in LINUX can be written as `cd..` (without space) in DOS and Windows.

`cp` *source dest* or `cp` *source... directory* - Copy files.

In order to make a copy of a file, the `cp` command is used. The `cp` command can be used to make a copy of a file from one directory into another. The first argument to the command is the name of the file to be copied known as the *source file*, and the second argument is the name of the file to place the copy into (known as the *destination file*). We can make a copy of the file names and call it `saved_names` as follows:

```
> cp names saved_names
>
```

Execution of this command causes the file named `names` to be copied into a file named `saved_names`. The fact that a command prompt was displayed after the `cp` command was typed indicates that the command executed successfully. The command

```
> cp file1 /tmp
```

copies `file1` to `/tmp/file1`. The command

```
> cp file[0-9] /tmp
```

copies files `file0` through `file9` to directory `/tmp`. For example, we can copy the file `wb` from the `programs` directory into a file called `wbx` in the `misc` directory as follows

```
> cp programs/wb misc/wbx
>
```

Since the two files are contained in different directories, it is not even necessary that they be given different names

```
> cp programs/wb misc/wb
>
```

When the destination file has the same name as the source file (in a different directory, of course), then it is necessary to specify only the destination directory as the second argument

```
> cp programs/wb misc
>
```

When this command gets executed, the UNIX system recognizes that the second argument is the name of a directory and copies the source file into that directory. The new file is given the same name as the source file. We can copy more than one file into a directory by listing the files to be copied before the name of the destination directory. If we were currently in the `programs` directory, then the command

```
> cp wb collect mon ../misc
>
```

would copy the three files `wb, collect`, and `mon` into the `misc` directory under the same names.

To copy a file from another directory into our current one and give it the same name, use the fact that the current directory can always be referenced as `'.'`

```
> pwd
> /usr/steve/misc
> cp ../programs/collect  .
>
```

The above command copies the file `collect` from the directory `../ programs` into the current directory `(/usr/steve/misc)`.

The copy command in DOS and Windows is `copy`.

`mv` *source dest* or `mv` source... directory - Move or rename files.

A file can be renamed with the `mv` command. The arguments to the `mv` command follow the same format as the `cp` command. The first argument is the name of the file to be renamed, and the second argument is the new name. Thus to change the name of the file `saved_names` to `hold_it`, for example, we enter the following command

```
> mv saved_names hold_it
>
```

When executing a `mv` or `cp` command, the UNIX system does not care whether the file specified as the second argument already exists. If it does, then the contents of the file will be lost. We assume we have the proper permission to write to the file. For example, if a file called `old_names` exists, then executing the command

```
> cp names old_names
```

would copy the file names to `old_names` destroying the previous contents of `old_names` in the process. Similarly, the command

```
> mv names old_names
```

would rename names to `old_names`, even if the file `old_names` existed prior to execution of the command. The command

```
> mv file1 /tmp
```

moves `file1` to `/tmp/file1`. The command

```
> mv file[a-g] /tmp
```

moves file `filea` through `fileg` to directory `/tmp`.

We recall that the `mv` command can be used to rename a file. However, when the two arguments to this command reference different directories, then the file is acutally moved from the first directory into the second directory. For example, first change from the home directory to the `documents` directory

```
> cd documents
>
```

Suppose now we decide that the file `plan` contained in the `memos` directory is really a proposal and not a memo. So we would like to move it from the `memos` directory into the `proposals` directory. Thus we apply the command

```
> mv memos/plan proposals/plan
>
```

As with the `cp` command, if the source file and destination file have the same name, then only the name of the destination directory need be supplied.

```
> mv memos/plan proposals
>
```

Also like the `cp` command, a group of files can be simultaneously moved into a directory by simply listing all files to be moved before the name of the destination directory

```
> pwd
  /usr/steve/programs
> mv wb collect mon ../misc
>
```

This would move the three files `wb, collect`, and `mon` into the directory `misc`.

We can also use the `mv` command to change the name of a directory. For example, the following will rename the directory `programs` to `bin`

```
> mv programs bin
>
```

`mkdir` *dir...* - Create a directory.

To create a directory, the `mkdir` command must be used. The argument to this command is simply the name of the directory we want to make. As an example, we wish to create a new directory called `misc` on the same level as the directories `documents` and `program`. If we were currently in our home directory, then typing the command `mkdir misc` would achieve the desired effect

```
> mkdir misc
>
```

Now if we execute an `ls` command, we should get the new directory listed. The command

```
> mkdir new_directory
```

create a new directory called `new_directory`. Note that there is no practical limit on the length of file or directory names, so `"This is a very long filename"` is a valid name for a file.

In DOS and Windows the commands are `md` and `mkdir`.

`rmdir` *dir...* - Delete an empty directory.

The command

```
> rmdir new_directory
```

deletes the directory we created above. Note that the directory must be empty.

The DOS and Windows command is also `rmdir`.

`rm` *name...* - Remove files or directories.

We can remove a directory with the `rmdir` command. The stipulation involved in removing a directory is that no files be contained in the directory. If there are files in the directory when `rmdir` is executed, then we will not be allowed to remove the directory. To remove the directory `misc` that we created earlier, the following could be used:

```
> rmdir /usr/steve/misc
>
```

Once again, the above command will work only if no files are contained in the `misc` directory; otherwise, the following will happen:

```
> rmdir /usr/steve/misc
  rmdir:  /usr/steve/misc not empty
>
```

If this happens and we still want to remove the `misc` directory, then we would first have to remove all of the files contained in that directory before reissuing the `rmdir` command.

To remove (delete) a file from the system, we use the `rm` command. The argument to `rm` is simply the name of the file to be removed

```
> rm hold_it
>
```

As an alternate method for removing a directory and the files contained in it, we can use the `-r` option to the `rm` command. The format is simple

```
> rm -r dir
```

where `dir` is the name of the directory that we want to remove. `rm` will remove the indicated directory and all files (including directories) in it.

We can remove more than one file at a time with the `rm` command by simply specifying all such files on the command line. For example, the following would remove the three files `wb, collect,` and `mon`

```
> rm wb collect mon
>
```

The command

```
> rm -r my_dir
```

deletes directory `my_dir` recursively, i.e. it does not have to be empty. The command

```
> rm -r *
```

deletes all files and directories recursively starting from the current directory. Note that there is no undelete command in Unix and Linux.

`cat` *name...* - Concatenate files and print on the standard output.

The `cat` command can be used to display a file or to concatenate files. Some of the options for the `cat` command have both long and short options specifiers. For space reasons, only the short versions of these options have been listed.

```
> cat [options] filelist
```

An output file name must not be the same as any of the input names unless it is a special file. The command

```
> cat first.cc
```

prints the contents of file `first.cc` on the screen, i.e. it concatenates `first.cc` with nothing and sends the output to the screen.

```
> cat letter-to-dad signature > send.let
```

This appends the file `signature` to `letter-to-dat` and creates a new file called `send.let`. We can examine the contents of a file by using the `cat` command. The argument to `cat` is the name of the file whose contents we wish to examine. For example, let the file name be `names`

```
> cat names
  Susan
  Jeff
  Henry
  Allan
  Ken
>
```

The command

```
> cat file1 file2 > file3
```

concatenates `file1` and `file2` and write the result to `file3`. The operator `>` redirects the standard output to the `cat` command to `file3`. The command

```
> cat longfile | more
```

the standard output of `cat` is used as the input of the `more` command.

Item	**Description**
filelist	This is an optional list of files to concatenate. If no files or a hyphen (-) is specified, the standard input is read.
`-b`	Numbers all nonblank output lines, starting with 1.
`-e`	Equivalent to -vE. (displays a `$` at the end of each line and control characters).
`-n`	Numbers all output lines, starting with 1.
`-s`	Replaces multiple, adjacent blank lines with a single blank line.
`-t`	Equivalent to -vT.
`-u`	Ignored; included for UNIX compatibility.
`-v`	Displays control characters except for LFD and TAB using `^` notation and precedes characters that have the high bit set with M-.
`-A`	Equivalent to -vET.
`-E`	Displays a `$` after the end of each line.
`-T`	Displays TAB characters as `^I`.
`--help`	Prints a usage message and exists with a non-zero status.
`--version`	Prints version information on standard output then exits.

`ln` [files] .. - Linking files

The `ln` command creates a link between two files, enabling us to have more than one name to access a file. A directory entry is simply a name to call the file and an inode number. The inode number is an index into the file system's table. Therefore, it is no big deal to have more than one name to inode reference in the same directory or multiple directories. A benefit of a link over a copy is that only one version of the file exists on the disk; therefore, no additional storage is required. Any file may have multiple links. In simplest terms, the `ln` command provides an easy way to give more than one name to a file. The drawback of copying a file is that now twice as much disk space is being consumed by the program.

`ln` provides two types of links:
hard links
and
symbolic links.

Linux treats hard links just like files. Each hard link is counted as a file entry. This means that the original file will not be deleted until all hard links have been deleted.

A symbolic link is treated as a place holder for the file. If a file has symbolic links, and the file is deleted, all symbolic links will be deleted automatically.

The syntax is

```
> ln [-s] source-file dest-file
```

For example

```
> ln source dest
```

This enables us to edit either the file `source` or the file `dest` and modify both of them at the same time. Now instead of two copies of the file existing, only one exists with two different names: `source` and `dest`. The two files have been logically linked by the UNIX system. It appears that we have two different files.

```
> ls
  collect
  mon
  source
  dest
>
```

Look what happens when we execute an `ls -l`:

```
> ls -l
  total 5
  -rwxr-cr-x   1 steve  DP3725   358 Jun 25  13:31 collect
  -rwxr-xr-x   1 steve  DP3725  1219 Jun 25  13:31 mon
  -rwxr-xr-x   2 steve  DP3725    89 Jun 25  13:30 source
  -rwxr-xr-x   2 steve  DP3725    89 Jun 25  13:30 dest
>
```

The number right before `steve` is 1 for `collect` and `mon` and 2 for `source` and `dest`. This number is the number of links to a file, normally 1 for nonlinked, nondirectory file. Since `source` and `dest` are linked, this number is 2 for these files. This implies that we can link to a file more than once.

The option `-s` makes a symbolic link instead of a hard link.

We can remove either of the two linked files at any time, and the other will not be removed

```
> rm source
> ls -l
  total 4

  -rwxr-xr-x  1 steve  DP3725   358 Jun 25  13:31 collect
  -rwxr-xr-x  1 steve  DP3725  1219 Jun 25  13:31 mon
  -rwxr-xr-x  1 steve  DP3725    89 Jun 25  13:30 dest
>
```

Note that `steve` is still listed as the `owner` of `dest`, even though the listing came from pat's directory. This makes sense, since really only one copy of the file exists - and it is owned by `steve`.

The only stipulation on linking files is that the files to be linked together must reside on the same file system. The `ln` command follows the same general format as `cp` and `mv`, meaning that we can link a bunch of files at once into a directory.

`wc` *names* - Counting the number of words in a file.

With the `wc` command, we can get a count of the total number of lines, word, and characters of information contained in a file. Once again, the name of the file is needed as the argument to this command

```
> wc names
     5      5      27  names
>
```

The `wc` command lists three numbers followed by the file name. The first number represents the number of lines contained in the file (5), the second the number of words contained in the file (in this case also 5), and the third the number of characters contained in the file (27).

The command

```
> wc -c names
```

counts only the number of characters. The command

```
> wc -w names
```

counts only the number of words. Finally the command

```
> wc -l names
```

counts only the number of lines, or more precisely, it counts the number of newline characters encountered.

1.2.5 File Name Substitution

The bash shell supports three kinds of wildcard:

```
*     matches any character and any number of characters
?     matches any single character
[...] matches any single character contained within the brackets
```

Each of them we discuss now in detail.

The Asterisk
One very powerful feature of the UNIX system that is actually handled by the shell is file name substitution. Let us say our current directory has these files in it:

```
> ls
   chap1
   chap2
   chap3
>
```

Suppose we want to print their contents at the terminal. We could take advantage of the fact that the `cat` command allows us to specify more than one file name at a time. When this is done, the contents of the files are displayed one after the other.

```
> cat chap1 chap2 chap3
        ...
>
```

However we can also enter

```
> cat *
       ...
>
```

and get the same results. The shell automatically substitutes the names of all of the files in the current directory for the `*`. The same substitution occurs if we use `*` with the `echo` command:

```
> echo *
  chap1 chap2 chap3
>
```

Here the `*` is again replaced with the names of all the files contained in the current directory, and the `echo` command simply displays them at the terminal.

Any place that * appears on the command line, the shell performs its substitution

```
> echo * : *
  chap1 chap2 chap3 : chap1 chap2 chap3
>
```

The * can also be used in combination with other characters to limit the file names that are substituted. For example, assume that in our current directory we have not only `chap1` through `chap4` but also files `a, b,` and `c`

```
> ls
  a
  b
  c
  chap1
  chap2
  chap3
>
```

to display the contents of just the files beginning with `chap`, we enter

```
> cat chap*
          .
          .
          .
>
```

The `cha*` matches any file name that begins with `cha`. All such file names matched are substituted on the command line.

The * is not limited to the end of a file name. It can also be used at the beginning or in the middle as well

```
> echo *p1
  chap1
> echo *p*
  chap1 chap2 chap3
> echo *x
  *x
>
```

In the first `echo`, the `*p1` specifies all file names that end in the characters `p1`. In the second `echo`, the first * matches everything up to a `p` and the second everything after; thus, all file names containing a `p` are printed. Since there are no files ending wit `x`, no substitution occurs in the last case. Therefore, the `echo` command simply display `*x`.

Matching Single Characters the ?
The asterisk `*` matches zero or more characters, meaning that `x*` will match the file `x` as well as `x1, x2, xabc`, etc. The question mark `?` matches exactly one character. So `cat ?` prints all files with one-character names. The command `cat x?` prints all files with two-character names beginning with `x`. We assume that the `ls` command tells us that the following files in the working directory

```
> ls
  a
  aa
  aax
  alice
  b
  bb
  c
  cc
  report1
  report2
  report3
```

Then

```
> echo ?
  a b c

> echo a?
  aa
> echo ??
  aa bb cc
> echo ??*
  aa aax alice bb cc report1 report2 report3
>
```

In the last example, the `??` matches two characters, and the `*` matches zero or more up to the end. The net effect is to match all file names of two or more characters.

The Brackets
Another way to match a single character is to give a list of the characters to use in the match inside square brackets `[ ]`. For example, `[abc]` matches one letter `a, b`, or `c`. It is similar to the `?`, but it allows us to choose the characters that will be matched. The specification `[0-9]` matches the characters `0` through `9`. The only restriction in specifying a range of characters is that the first character must be alphabetically less than the last character, so that `[z-f]` is not a valid range specification.

By mixing and matching ranges and characters in the list, we can perform some very complicated substitutions. For example, `[a-np-z]*` will match all files that start with the letters `a` through `n` or `p` through `z` (or more simply stated, any lowercase letter but `o`).

If the first character following the `[` is a `!`, then the sense of the match is inverted. This means, any character will be matched except those enclosed in the brackets. So

```
> [!a-z]
```

matches any character except a lowercase letter, and

```
> *[!o]
```

matches any file that does not end with the lowercase letter `o`.

The following table gives a few more examples of filename substitution.

File name substitution examples

Command	*Description*
`echo a*`	Print the *names* of the files beginning with a
`cat *.c`	print all files ending in `.c`
`rm *.*`	Remove all files containing a period
`ls x*`	List the names of all files beginning with `x`
`rm *`	Remove *all* files in the current directory (note: be careful when we use this)
`echo a*b`	Print the names of all files beginning with `a` and ending with `b`
`cp ../programs/* .`	Copy all files from `../programs` into the current directory
`ls [a-z]*[!0-9]`	List files that begin with a lowercase letter and do not end with a digit.

1.3 Standard Input/Output, and I/O Redirection

Most UNIX system commands take input from our terminal and send the resulting output back to our terminal. A command normally reads its input from a place called *standard input*, which happens to be our terminal by default. Similarly, a command normally writes its output to *standard output*, which is also our terminal by default.

We recall that executing the `who` command results in the display of the currently logged-in users. More formally, the `who` command writes a list of the logged-in users to standard output.

If a `sort` command is executed without a file name argument, then the command will take its input from standard input. As with standard output, this is our terminal by default.

When entering data to a command from the terminal, the `CTRL` and `D` keys (denoted `CTRL-d` in this text) must be simultaneously pressed after the last data item has been entered. This tells the command that we have finished entering data. As an example, we use the `sort` command to sort the following three names: Tony, Barbara, Harry. Instead of first entering the names into a file, we enter them directly from the terminal

```
> sort
   Tony
   Barbara
   Harry
   CTRL-d
   Barbara
   Harry
   Tony
>
```

Since no file name was specified to the `sort` command, the input was taken from standard input, the terminal. After the fourth name was typed in, the `CTRL` and `D` keys were pressed to signal the end of the data. At the point, the `sort` command sorted the three names and displayed the results on the standard output device, which is also the terminal.

The `wc` command is another example of a command that takes its input from standard input if no file name is specified on the command line. So the following shows an example of this command used to count the number of lines of text entered from the terminal

```
> wc -l
   This is text that
   is typed on the
   standard input device.
   CTRL-d
   3
>
```

We note that the `CTRL-d` that is used to terminate the input is not counted as a separate line by the `wc` command. Furthermore, since no file name was specified to the `wc` command, only the count of the number of lines (3) is listed as the output of the command. We recall that this command normally prints the name of the file directly after the count.

Output Redirection

The output from a command normally intended for standard output can be diverted to a file instead. This capability is known as output redirection.

If the notation `>` file is appended to any command that normally writes its output to standard output, then the output of that command will be written to file instead of our terminal

```
> who > users
>
```

This command line causes the `who` command to be executed and its output to be written into the file `users`. No output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file

```
> cat users
   oko   tty01  Sep  12  07:30
   ai    tty15  Sep  12  13:32
   ruth  tty21  Sep  12  10:10
   pat   tty24  Sep  12  13:07
   steve tty25  Sep  12  13:03
>
```

If a command has its output redirected to a file and the file already contains some data, then the data will be lost. For example

```
> echo line 1 > users
> cat users
   line 1
> echo line 2 >> users
> cat users
   line 1
   line 2
>
```

The second `echo` command uses a different type of output redirection indicated by the characters `>>`. This character pair causes the standard output from the command to be appended to the specified file. Therefore, the previous contents of the file are not lost and the new output simply gets added onto the end.

By using the redirection append characters `>>`, we can use `cat` to append the contents of one file onto the end of another

```
> cat file1
   This is in file1.
> cat file2
   This is in file2.
> cat file1 >> file2         Append file1 to file2
> cat file2
   This is in file2.
   This is in file1.
>
```

Recall that specifying more than one file name to `cat` results in the display of the first file followed immediately by the second file, and so on

```
> cat file1
  This is in file1.
> cat file2
   This is in file2.
>  cat file1 file2
   This is in file1.
   This is in file2.
>  cat file1 file2 > file3      Redirect it instead
>  cat file3
   This is in file1.
   This is in file2.
>
```

Now we can see where the `cat` command gets its name: when used with more than one file its effect is to concatenate the files together.

Incidentally, the shell recognizes a special format of output redirection. If we type

```
> file
```

not preceded by a command, then the shell will create an empty (i.e., zero character length) *file* for us. If *file* previously exists, then its contents will be lost.

Input Redirection
Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character `>` is used for output redirection, the less-than character `<` is used to redirect the input of a command. Of course, only commands that normally take their input from standard input can have their input redirected from a file in this manner.

In order to redirect the input of a command, we type the `<` character followed by the name of the file that the input is to be read from. So, for example, to count the number of lines in the file `users`, we can execute the command

```
> wc -l users
        2 users
>
```

Or, we can count the number of lines in the file by redirecting the input of the `wc` command from the terminal to the file `users`

```
> wc -l < users
  2
>
```

We notice that there is a difference in the output produced by the two forms of the `wc` command. In the first case, the name of the file `users` is listed with the line count; in the second case, it is not. This points out the subtle distinction between the execution of the two commands. In the first case `wc` knows it is reading its input from the file `users`. In the second case, it only knows that it is reading its input from standard input. The shell redirects the input from the terminal to the file `users`. As far as `wc` is concerned, it does not know whether its input is coming from the terminal or from a file.

Pipes
The file `users` that was created previously contains a list of all the users currently logged into the system. Since we know that there will be one line in the file for each user logged into the system, we can easily determine the number of users logged in by simply counting the number of lines in the `users` file

```
> who > users
> wc -l < users
  5
>
```

This output would indicate that there were currently five users logged in. Now we have a command sequence we can use whenever we want to know how many users are logged in.

There is another approach to determine the number of logged-in users that bypasses the use of a file. The UNIX system allows to connect two commands together. This connection is known as a *pipe*, and it enables us to take the output from one command and feed it directly into the input of another command. A pipe is effected by the character `|`, which is placed between the two commands. So to make a pipe between the `who` and `wc -l` commands, we simply type

```
> who | wc -l
         5
>
```

When a pipe is set up between two commands, the standard output from the first command is connected directly to the standard input of the second command. We know that the `who` command writes its list of logged-in users to standard output. Furthermore, we know that if no file name argument is specified to the `wc` command then it takes its input from standard input. Therefore, the list of logged-in users that is output from the `who` command automatically becomes the input to the `wc` command. Note that we never see the output of the `who` command at the terminal, since it is piped directly into the `wc` command.

A pipe can be made between any two programs, provided the first program writes its output to standard output, and the second program reads its input from standard input.

Suppose we wanted to count the number of files contained in our directory. Knowledge of the fact that the `ls` command displays one line of output per file enables us to see the same type of approach as before

```
> ls | wc -l
         10
>
```

The output indicates that the current directory contains 10 files.

It is also possible to form a pipeline consisting of more than two programs, with the output of one program feeding into the input of the next.

Consider, for example, the command

```
> cat myfile | sort | uniq
```

Here, the contents of the file `myfile` (the output from the `cat` command) are fed into the input of the `sort` command. The `sort` command, without any options, sorts the input alphabetically by the first field in the input. The sorted file is then piped into the `uniq` command. The `uniq` command removes any duplicate lines from the input.

The POSIX standard specifies that (in the Bourne shell) descriptors are assigned to the pipeline before redirection (i.e. any redirection does not affect the pipeline). The command `tee` can be used to write from its standard input to a file and to standard output.

```
> command | tee filename
```

This command line executes *command*, with the output piped through `tee` which will write the data into *filename* and to the console. Since the pipe is set up before any redirection, we could write the same command using the Bourne shell using `cat` and the redirection and pipeline facilities.

```
> command > filename | cat
```

Filters
The term *filter* is often used in UNIX terminology to refer to any program that can take input from standard input, perform some operation on that input, and write the results to standard output. A filter is any program that can be used between two other programs in a pipeline. So in the previous pipeline, `wc` is considered a filter. `ls` is not, since it does not read its input from standard input. For example, `cat` and `sort` are filters, while `who, date, cd, pwd, echo, rm, mv,` and `cp` are not.

Standard Error
In addition to standard input and standard output there is another place known as *standard error*. This is where most UNIX commands write their error messages. And as with the other two standard places, standard error is associated with our terminal by default. In most cases, we never know the difference between standard output and standard error

```
> ls n*                List all files beginning with n
  n* not found
>
```

Here the `not found` message is actually being written to standard error and not standard output by the `ls` command. We van verify that this message is not being written to standard output by redirect into the `ls` command's output

```
> ls n* > foo
   n* not found
>
```

We still get the message printed out at the terminal, even though we redirected standard output to the file `foo`.

The above example shows the raison d'être for standard error: so that error messages will still get displayed at the terminal even if standard output is redirected to a file or piped to another command.

We can also redirect standard error to a file by using the notation

```
> command 2> file
```

No space is permitted between the 2 and the >. Any error messages normally intended for standard error will be diverted into the specified file, similar to the way standard output gets redirected.

```
> ls n* 2> errors
> cat errors
   n* not found
>
```

Chapter 2

Linux Commands

In this chapter we describe a number of other important commands in LINUX and give examples. The commands include

```
more
less
file
find
finger
grep
id
kill
logname
sleep
split
chkconfig
sort
tee
dd
```

The commands

```
df
du
free
```

provide information about the disk space usage.

The command `chkconfig` is in the directory `sbin`. Some other commands are in the directory `bin`.

`more` *display File Contents*

The `more` command displays a screenful of a text file. We can look through a text file without invoking an editor, printing the file, or trying to pause the terminal as it displays the file. If we try to pass a binary data file to `more`, we could have some unpleasant effects; for example, our terminal can lock up. If our terminal does lock up, try pressing either the `Ctrl-q` or the `Ctrl-s` keys.

A disadvantage with `more` is that we cannot back up to see a screen of information once it passes. For example, to display the contents of our `emacs` configuration file, we can type the following command

```
>  more  .emacs
```

To quit `more` we enter `q` or `Crtl-z`.

`less` *name ...* view a file

The `less` command is considered as a better `more`. `less` displays information a screen at a time on our terminal. The program's name is a play on words for the program it's meant to replace - `more`. Like `more`, `less` can display a screen of information in a text file. Unlike `more`, with the `less` we can page back and forth within the file. The command

```
> less longfile
```

views file `longfile`. We can press `page-up` and `page-down` to scroll through the file. We enter `q` to exit.

`file` determines type of file

Determines the type of a file. The command `file` is able to recognize whether the file is executable, text, data, and so on. Many of the UNIX commands are only shell scripts. The command `file` can be used to report the UNIX commands that are and are not scripts. It is also useful to determine if the file is text based on or not and, therefore, whether it can be viewed or edited. The syntax is

```
> file [-c] [-z] [-L] [-f ffile ] [-m mfile] filelist
```

where `filelist` is a space-separated list of files we want to know the type of.

Item	Description
`-c`	Prints out the parsed form of the magic file. This is used in conjunction with `-m` to debug a new magic file before installing it.
`-z`	Looks inside a compressed file and tries to figure out its type.
`-L`	This option causes symbolic links to be followed.
`-f ffile`	Tells file that the list of files to identify is found in *ffile*. This is useful when many files must be identified.
`-m mfile`	Specifies an alternate file of magic numbers to use for determining file types.

For example, let `first.cc` be a C++ file. Then

```
> file first.cc
  C++ program text
```

If we compile and link this file we obtain an execute file `first`. Then

```
> file first
  ELF 32-bit LSB executable, Intel 80386
  version 1, dynamically linked, not stripped
```

`find` finding files

The `find` command traverses the specified directories generating a list of files that match the criteria specified. Files may be matched by name, size, creation time, modification time, and many more criteria. We can even execute a command on the matched files each time a file is found. For example

```
> find dirlist match-spec
```

where `dirlist` is space-separated list of the directories where we want to look for a file or set of files and `match-spec` are the matching specification or description of files we want to find.

Description	
`-name file`	Tells `find` what file to search for; the file to find is enclosed in quotation marks. Wild cards (* and ?) may be used.
`-perm mode`	Matches all files whose mode matches the numeric value of *mode*. All modes must be matched – not just read, write, and execute. If preceded by a negative `-`, mode take on the meaning of everything without this mode.
`-type x`	Matches all files whose type, x, is c (meaning) b (block special), d (directory), p (named pipe), l (symbolic link), s (socket), or f (regular file).
`-links n`	Matches all files with n number of links
`-size n`	Matches all files of size n blocks (512-byte blocks, 1K-byte blocks if k follows n)
`-user user-id`	Matches all files whose user ID is `user-id`. May either be the numeric value or the logname of the user.
`-atime n`	Matches all files last accesses within the previous n days.

Description	
`-mtime n`	Matches all files modified within the previous n days.
`-exec cmd`	For each file matched, the command `cmd` is executed. The notation `{}` is used to signify where the file name should appear in the command executed. The command must be terminated by an escaped semicolon (`\;`), for example `-exec ls -d {}\;`. Here, the command `ls` is executedwith the `-d` argument and each file is passed to `ls` at the place where the `{}` is found.
`-newer file`	Matches all files that have been modified more recently than the file `file`.

The options may be grouped together and combined to limit the search criteria. Multiple flags are assumed to be ANDs, meaning that both criteria must be met. To offer more control over selection, the following table describes other options

```
()  parentheses may be used to group selections.
    The parentheses are special to the shell, they must be escaped.
-o  This is the OR operator, overriding the default AND assumption.
!   This is a NOT operator and negates the expression that follows it.
```

For example

```
>  find . -name first.cc -print
```

searches the current directory (the `.` indicates the current directory) and its subdirectories for a file called `first.cc`. When and if `find` finds it, the full path name is shown on-screen. The command

```
> find . -iname '*.tex'
```

finds all Latex files starting in the current directory. The command

```
> find . -newer linux.tex
```

finds all files that were modified more recently than `linux.tex`. The command

```
> find . -iname '*.tex' -exec cp {} /tmp
```

copies all Latex files into directory `/tmp`.

`finger` information about users on the system

The `finger` command displays information about users on the system. The syntax is

```
> finger [options] users
```

Item	**Description**
`users`	This is an optional list of user names. If specified, then additional information about the user is given. Remote users can be specified by giving the user name and the remote host name as in `user@remote.computer`.
`-s`	Displays the user's logon name, real name, terminal name and write status, idle time, logon time, office location, and office phone number.
`-l`	Produces a multi-line format displaying all of the information described for the `-s` option as well as the user's home directory, home phone number, logon shell, mail status, and the contents of the user's `.plan` and `.project` files.
`-p`	Prevents the `-l` option of finger from displaying the contents of the `.plan` and `.project` files.
`-m`	Restricts matching of the user argument to the logon name.

If no options ar specified, `finger` defaults to the multi-line output format as specified by the `-l` argument.

If no arguments are specified, `finger` prints an entry for each user currently logged onto the system.

`grep`

The command **`grep`** looks for patterns found in files and reports to us when these patterns are found. The name of the command comes from the use of ”regular expressions” in the `ed` family of editors. **`grep`** stands for Global Regular Express Printer. The syntax is

```
> grep  [option] reg-express filelist

> egrep  [options] reg-express filelist

> fgrep  [options] string filelist
```

Item	**Description**
`filelist`	An optional space-separated list of files to search for the given `string` or `reg-expres`. If left blank, the standard input is searched.
`reg-express`	The regular expression to search for. Regular expressions are in the form used by `ed`. See the man page for the definition of regular expressions.
`string`	The string we want to find in the files.
`-v`	List the lines that don't match `string` or `reg-expres`.
`-c`	Counts the matching lines.
`-l`	Only the names of the files containing a match are displayed.
`-h`	Suppresses the display of the name of the file the match was found in (`grep` and `egrep` only).
`-n`	Each matching line is displayed along with its relative line number.
`i`	Causes matching to not be case-sensitive. The default behavior is to be case-sensitive.
`-e reg-expres`	useful when the regular expression or string starts with a hyphen.
`-f file`	`file` contains the strings or expressions to search for.

`fgrep` stands for fast `grep` and can only search for fixed strings. Multiple strings may be searched for by separating each string by a new line or entering them in the `-f file` file.

`egrep` stands for extended `grep` and accepts the following enhancements to regular expressions defined by `ed`:

- `+` If this trails a regular expression, it matches one or more of that occurrence.
- `?` If this trails a regular expression, it matches 0 or 1 occurrence.
- `|` used to denote multiple regular expressions.
- `()` May be used to group expressions.

For example

```
> grep main first.cc
```

searches for the word `main` in the file `first.cc`. The command

```
> fgrep main first.cc
```

does the same thing. The command

```
> grep "[hH]ello" letter-to-dad
```

searches for the word `hello` or `Hello`. The command

```
> fgrep "hello
> Hello" letter-to-dat
```

does the same thing. The command

```
> egrep "([Ss]ome|[Aa]ny)one" letter-to-dad
```

looks for all the words `someone`, `Someone`, `anyone`, or `Anyone` in the file.

`id`

The `id` command displays our identification to the system. It reports our user name, user ID number, group name, and group Id number. The syntax is

```
> id [options]
```

Item	**Description**
`-g`	Prints only the group ID.
`-G`	Prints only the supplementary groups.
`-n`	Prints the user or group name instead of the ID number. Requires `-u, -g`, or `-G`.
`-r`	Prints the real, instead of effective, user or group ID. Requires `-u, -g`, or `-G`.
`-u`	Prints only the user ID.
`--help`	prints a usage message on the standard output and exits successfully.
`--version`	Prints version information on the standard output and then exits successfully.

For example

```
> id
```

shows the `id` information.

`kill` to kill a process

With the `kill` command we can send a signal to a process that is currently executing. This command is issued to stop the executing process, thus the `kill` name because we use it to kill the process. The syntax is

```
> kill [-signal] pid
> kill -l
```

Item	**Description**
`signal`	An optional signal that can be sent. The default is `SIGTERM`. Two other popular ones are `SIGHUP`, which is the equivalent of hanging up the phone as if on a modem, and `SIGKILL`, which cannot be ignored by a process.
`pid`	The process ID of the process we want to send the specified signal. A `pid` is a number used by the system to keep track of the process. The `ps` command can be used to report the `pid` of a process.
`-l`	Prints a list of the signal names that can be sent with `kill`.

Although `kill -9` is the sure kill, it is often best to try `SIGTERM` and `SIGHUP` first. These signals can be caught by the applications and after they receive them, properly clean up after themselves. Because `kill -9` can't be caught, we may have to do some house cleaning after the process terminates.

The messages can be

```
> kill: permission denied
```

We tried to kill a process that we don't own, or we are not the super user.

For example

```
> kill 125
```

sends the `SIGTERM` signal to the process 125. The default signal is `SIGTERM`. The command

```
> kill -9 125
```

sends signal 9, which is the numeric value for `SIGKILL`. This works if no other signal does. There are instances when even `kill -9` can't kill the process. This is when the process is using a kernel service and can't receive signals. Periodically processes get locked up in this mode. The only way to resolve this is to shut down the system.

`logname`

The command `logname` reads the `/etc/utmp` file to report what name we used to log onto the system. For example

```
> logname
```

reports what name we used to log onto the system.

`sleep` sleep

The `sleep` command suspends execution for an interval of time. The syntax is

```
> sleep n
```

Item	Description
`n`	Specifies the amount of time to sleep. This must be an integer. The default is measured in seconds. An optional identifier may be used to specify a different time unit. These are as follows: s Seconds; m Minutes; h Hours; d Days

For example at the command line we enter the code

```
> while :
  do
     date
     pwd
     sleep 10
  done
```

This shows the `date` and `pwd` every 10 seconds. The program runs forever. To exit the running program we enter `CRTL-z`.

We notice that `sleep` is not guaranteed to wake up exactly after the amount of time specified.

`split` breaks up text files

The `split` command breaks up a text file into smaller pieces. Periodically, files become too large to load into an editor or some other utility. With the command `split` we can handle the file in individual, more manageable pieces. The syntax is

```
> split -numlines file tagname
```

Item	**Description**
`-numlines`	Specifies the number of lines to include in each piece.
`file`	The file to split into smaller pieces. If left blank or `-` is used, standard input is read.
`tagname`	By default, `split` builds the output pieces by creating the following files: `xaa`, then `xab`, then `xac`, and so on. `tagname`, if specified, replaces the `x` in the previous list, thus building the list: `tagnameaa, tagnameab, tagnameac`, and so on.

There must be enough room for two copies of the file in the current file system.

For examples

```
> split -100 letter myletter
```

breaks up the file `letter` into 100 line pieces. The output files are

```
myletteraa, myletterab, ...
```

The command

```
> cat myletter* > letter
```

takes all the pieces and puts them back together again into the file `letter`.

The message could be

```
> No such file or directory
```

In this case we supplied `split` with a file name that doesn't exist.

```
chkconfig
```

The `chkconfig` command manipulates or displays settings for system run levels. The syntax is

```
> chkconfig --list [name]

> chkconfig --add name

> chkconfig --del name

> chkconfig [--level levels]
  name <on|off|reset>

> chkconfig [--level levels]
  name
```

Important Flags and Options	`--add name`
	Adds a new service for management by `chkconfig` and checks that the necessary start and kill entries exist. If entries are missing, they are created.
	`--del name:`
	Deletes the name service from management; any links to it are also deleted.
	`--level levels:`
	Specifies numerically which run level a named service should belong to.
	`--list name:`
	Lists all services that `chkconfig` knows about and any relevant information about them. If a named service is specified, only information about that service is displayed.

For example

```
> chkconfig --list random
```

gives

```
random 0:off 1:on 2:on 3:on 4:on 5:on 6:off
```

`sort` sorting files or standard input

The `sort` command sorts the lines contained in one or more text files and displays the results. If no files are indicated, data is taken from the standard input and sorted. The resulting sorted data is displayed to the standard output. The syntax is

```
> sort filename
```

Important flags are

```
-b   ignores leading blanks in lines when trying to find sort keys
-c   checks whether the input data is sorted and prints
     an error message if it is not.
     No sorting actually takes place.
-d   ignore all characters except letters, digits, and blanks
     in the sorting process
-f   converts lowercase letters to uppercase letters during the
     sorting process
-r   reverses the sort order
-t   separator
     indicates that the specified separator should serve as the
     field separator for finding sort keys on each line.
```

For example

```
> sort
  Ben
  Alda
  Willi
  Roba
  <Crtl-d>
```

provides

```
 Alda
 Ben
 Roba
 Willi
```

The command

```
> sort -r
  Ben
  Alda
  Willi
  Roba
  <Crtl-d>
```

provides

```
Willi
Roba
Ben
Alda
```

`tee` duplicating output (pipe fitting)

The `tee` command copies standard input to a file and duplicates the data to standard output. The command name is due to the analogy with the 'T' shape pipe connection which creates a fork so that flow (in our case data) can go in two different directions.

```
> tee filename
```

The command is useful for logging. For example, when compiling the Linux kernel a lot of information is generated about the process. In general we would like to watch this information for errors, but the information scrolls past very quickly so it would be better to redirect this information to a file. So we might use the command

```
> make kernel > /tmp/log
```

and view the log on another console using

```
> tail -f /tmp/log
```

On the other hand we could use the `tee` command to do this on one console

```
> make kernel | tee /tmp/log
```

which will write the output of `make kernel` to /tmp/log as well as displaying it to the console.

As an other example we compile a program, storing the compiler warnings and errors in `compile.log` while viewing the results

```
> cc -o program program.c 2>&1 | tee compile.log
> more compile.log
```

`dd` data/disk/device dump

The `dd` dumps data from one file to another (like a copy) similar to `cat` except that it allows some conversion to be specified as well as device specific information.

```
> dd parameter=value ...
```

Important parameters are

```
if    input file name, if not specified it will be
      the standard input.
of    output filename, if not specified it will be
      the standard output.
bs    block size, the size of blocks used for reading
      and writing (bytes).
ibs   input block size, the size of blocks used
      for reading (bytes).
obs   output block size, the size of blocks used
      for writing (bytes).
count number of blocks to copy, all if not specified.
conv  conversion applied (comma separated list of options)
      swab:      swap pairs of bytes
      noerror:   ignore errors
      sync:      always write the full block, even
                 when there are errors.
skip  number of blocks to skip on the input before proceeding
      with copy.
```

For example

```
> dd if=/dev/cdrom of=/tmp/cdrom.iso bs=2048
```

will copy from a `cdrom` device with block size 2048 (the sector size for a data cdrom). This is useful if your cdrom only supports copying whole sectors (not necessary on Linux systems), or for a tape drive. We can the use the file `/tmp/cdrom.iso` to create duplicate cdroms. It is also useful sometimes to specify `conv=noerror,sync` so that a device with errors can be copied as completely as possible.

To obtain information about disk space usage we use the commands `df`, `du`, and `free`.

`df` free disk space

The `df` command displays free space on one or more mounted disks or partitions. If no files or directories are specified, then the free space on all mounted file systems is displayed. If filenames are specified, then the free space for the file system containing each file is displayed. The syntax is

```
> df [-T] [-t] fstype] [-x fstype]
  [--all] [--inodes] [--type=fstype] [--exclude-type=fstype]
  [--print-type]
  [filename ...]
```

Important Flags and Options	`t/--type=fstype:` Displays information only for the specified type. `-T/--print-type:` Displays the file system type for each file system reported. `-x/--exclude-type=fstype:` Does not report information for file systems of the specified type.

`du` reports on disk space usage

The `du` command displays free space on one or more mounted disks or partitions. If no files or directories are specified, then the free space on all mounted file systems is displayed. If filenames are specified, then the free space for the file system containing each file is displayed. The `du` command displays file system block usage for each file argument and each directory in the file hierarchy rooted in each directory argument.

The syntax is

```
> du [-abcksx] [--all] [--bytes]
  [--total] [--kilobytes] [--summarize]
  [--one-file-system] [file ...]
```

Important Flags and Options	`-a/--all:` Displays usage information for files and directories. `-b/--bytes:` Displays usage information in bytes. `-c/--total:` Displays a total usage figure for all. `-k/--kilobytes:` Displays usage informaton in kilobytes. `-s/--summarize:` Displays a total for each argument and not display individual information for each file or subdirectory inside a directory. `-x/--one-file-system;` skips directories that are not part of the current file system.

For example

```
> du --summarize --bytes
```

gives

```
> 6373155
```

`free` reports of free and used memory

Displays a report of free and used memory. The syntax is

```
> free [-b|-k|-m] [-s delay]
```

Important Flags and Options	`-b`: Displays the amount of memory in bytes. `-k`: Displays the amount of memory in kilobytes (this is the default). `-m`: Displays the amount of memory in megabytes. `-s delay:` Displays continued reports separated by the specified delay in seconds. `-t`: Displays an extra line containing totals.

The command

```
> whereis program
```

searches for the specified program in standard locations. For example

```
> whereis sh
/bin/sh
> whereis startx
/usr/X11R6/bin/startx
```

The command

```
> which
```

the command searches for the specified program according to the PATH environment variable.

```
> which sh
/bin/sh
> which startx
/usr/X11R6/bin/startx
```

The command

```
> csplit file arguments
```

splits the file into multiple files according to the `arguments` which describe how to split the file according to lines.

```
> csplit /usr/X11R6/lib/X11/rgb.txt 10
246
17125
```

The first 10 lines (246 bytes) of `/usr/X11R6/lib/X11/rgb.txt` are written to the file `xx00` and the remaining lines (17125 bytes) are written to `xx01`.

```
> csplit /usr/X11R6/lib/X11/rgb.txt 10 "{10}"
246
241
236
236
238
249
228
278
237
252
258
14672
```

The first 10 lines (246 bytes) of `/usr/X11R6/lib/X11/rgb.txt` are written to the file `xx00`, the following 10 (241 bytes) to `xx01`, ..., the following 10 (258 bytes) to `xx10` and the remaining lines (14672 bytes) are written to `xx11`.

```
> csplit -f rgb /usr/X11R6/lib/X11/rgb.txt 10 "{10}"
246
241
236
236
238
249
228
278
237
252
258
14672
```

Same as above except the files are `rgb00`, `rgb01`, ..., and `rgb11`.

```
> csplit -f rgb /usr/X11R6/lib/X11/rgb.txt /gold/
3737
13634
```

Each line from `/usr/X11R6/lib/X11/rgb.txt` is written to `rgb00` (3737 bytes) until, not including, the line contains the word `gold`. Subsequent lines are written to `rgb01` (13634 bytes).

```
> csplit -f rgb /usr/X11R6/lib/X11/rgb.txt /gold/ %gold%
3737
13579
```

Each line from `/usr/X11R6/lib/X11/rgb.txt` is written to `rgb00` (3737 bytes) until, not including, the line contains the word `gold`. Subsequent lines are ignored until the word `gold` is encountered again, and then the remaining lines are written to `rgb01` (13579 bytes).

The command

```
> cut
```

selects a portion of each line presented as input.

```
> echo Name, Telephone, Office | cut -b 1-4
Name
> echo Name, Telephone, Office | cut -c 1-4
Name
> echo Name, Telephone, Office | cut -c 1,7
```

```
NT
> echo Name, Telephone, Office | cut -c -5,18-
Name,Office
> echo Name, Telephone, Office | cut -f 1,3 -d ","
Name, Office
```

The parameter `-c` works on the character level whereas `-b` works on the byte level. The parameter `-f` works on fields separated by a delimeter as specified by `-d`.

The command

```
> expand
```

converts tab characters in sequences of ordinary spaces.

```
> printf "1\t2\t3\t4\t5\n"
1       2       3       4       5
>  printf "1\t2\t3\t4\t5\n" | expand
1       2       3       4       5
> printf "1\t2\t3\t4\t5\n" | expand -t 1,2,5,9,14
1 2  3   4    5
```

The command

```
> fold
```

ensures that lines do not exceed a specified width with the default width being 80 characters.

```
> echo "a long, long, long line" | fold -w 20
a long, long, long l
ine
yorick:/tmp
> echo "a long, long, long line" | fold -w 10
a long, lo
ng, long l
ine
> echo "a long, long, long line" | fold -s -w 10
a long,
long,
long line
```

The command

```
> head
```

displays a specified number of lines (default 10) of a file or standard input.

```
>  head -n 3 /usr/X11R6/lib/X11/rgb.txt
! $Xorg: rgb.txt,v 1.3 2000/08/17 19:54:00 cpqbld Exp $
255 250 250             snow
248 248 255             ghost white
```

The command

```
> bg
```

instructs the last stopped process (SIGSTOP sent from the terminal) to run in the background.

```
> cat /dev/zero > /dev/null
^Z[1] + Stopped              cat /dev/zero > /dev/null
> jobs
[1] + Stopped              cat /dev/zero > /dev/null
> bg
[1] cat /dev/zero > /dev/null
>  jobs
[1] + Running              cat /dev/zero > /dev/null
> ps
PID TT STAT    TIME COMMAND
323 p3 Ss   0:00.05 sh
581 p3 R    0:05.31 cat /dev/zero
584 p3 R+   0:00.00 ps
> kill 581
>
[1] + Terminated           cat /dev/zero > /dev/null
```

The command

```
> fg
```

instructs the last stopped process (SIGSTOP sent from the terminal) to run in the foreground.

```
> cat /dev/zero > /dev/null
^Z[1] + Stopped              cat /dev/zero > /dev/null
yorick:/tmp
> fg
cat /dev/zero > /dev/null
^C
```

The command

```
> nice
```

runs a program with adjusted system scheduling priority.

```
> #slower
> nice -n +10 /usr/games/primes 10000 20000
> #slowest
> nice -n +20 /usr/games/primes 10000 20000
> #faster (use as root)
> nice -n -10 /usr/games/primes 10000 20000
> #fastest (use as root)
> nice -n -20 /usr/games/primes 10000 20000
```

The command

```
> apropos keyword
```

list manual pages matching keyword(s)

```
> apropos apropos editor
ed (1) - text editor
ex, vi, view (1) - text editors
sed (1) - stream editor
> man sed
```

The command

```
> whatis command
```

describes a command.

```
> whatis sed
sed (1) - stream editor
> whatis whatis
whatis (1) - describe what a command is
```

The command

```
> cksum
```

computes and displays a checksum for the listed files.

```
> cksum /bin/sh
2689572179 457456 /bin/sh
```

The command

```
> hash
```

updates the environment the shell uses when searching for utilities.

```
> hash
cat=/bin/cat
ls=/bin/ls
sh=/bin/sh
> hash bc
yorick:/tmp
> hash
bc=/usr/bin/bc
cat=/bin/cat
ls=/bin/ls
sh=/bin/sh
```

The command

```
> printenv
```

outputs the environment, or a specific environment variable.

```
> printenv
PAGER=more
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin
EDITOR=vi
...
> printenv SHELL
/bin/bash
```

The command

```
> set
```

display all shell variables and environment variables. The command can also set options for the shell.

```
> echo hello > hello
> echo hello > hello
> set -o noclobber
> echo hello > hello
sh: cannot create hello: File exists
```

The command

```
> sum
```

calculates the checksum of a file. `cksum` should be used instead.

```
> sum /bin/sh
64892 447 /bin/sh
```

The command

```
> tty
```

outputs the terminal name of the terminal we are using.

```
> tty
/dev/ttyp3
```

Finally let us summarize other important Linux commands.

The command `arch` displays the architecture on which Linux is running.

The `at` command schedules commands to be executed at a specific time. The user is prompted for the commands or the commands can be provided from a file.

The `batch` command schedules commands to be executed at a specified time as long as system load levels permit.

The command `checkalias` checks the user's file and then the system alias file in order to see if a specified alias is defined.

The command `chgrp` changes the group ownership of one or more files or directories. Using the command `chmod` we can change the access permissions of one or more files or directories. Modes can be specified in two ways: symbolically and numerically. When done symbolically, modes take the form

```
[ugoa][[+-=][rwxXstugo ... ]
```

The first element `[ugoa]` represents the users whose permissions are being affected (`u` = user who owns the file or directory, `g` = all members of the group that owns the file or directory, `o` = anyone who is not the owner or in the owner group, `a` = all users). The symbol `+` means that the specified modes should be added to already specified permissions, the `-` means the specified modes should be removed from existing permissions. The symbol `r` stands for read permission, `w` for write permission, and `x` for execute permission.

The `chown` command changes the user and/or group ownership of one or more files or directories.

The command `kdb_mode` displays or sets the keyboard mode and `kbdrate` sets the repeat rate and delay time for the keyboard. The command `showkey` displays the scancodes and keycodes generated by the keyboard. The program remains active for 10 seconds after the last key is pressed.

Using the `last` command we can display a history of user logins and logouts based on the contents of `/var/log/wtmp`. If a specific `tty` such as `tty0` or `tty1` is specified, then only logins to that `tty` are displayed.

The `paste` command merges corresponding lines from one or more files and prints sequentially corresponding lines on the same line, separated by a tab with a new line ending the line. If no file name is provided input is taken from the standard input.

The `tail` command displays the last part of one or more files. By default, unless otherwise specified, the last 10 lines of each file are displayed. The `head` command displays the first part of one or more files. By default, unless otherwise specified, the first 10 lines of each file are displayed. If no filenames are provided, then the command reads data from the standard input and displays the first section of the data, following the same rules as for files.

With the command `timeconfig` we can configure time parameters. If a time zone is specified, then the system time zone is changed to the specified time zone. Otherwise, it displays a list of available time zones.

The `touch` command changes the timestamp of files without changing their contents. If a file does not exist, it will be created with size zero. By default, it uses the current time as timestamp. The option `-c` will not create files that do not exist.

The `setclock` command sets the computer's hardware clock to the value of the current system clock.

The command `whereis` attempts to locate binary.source code, and man page files for one or more commands. For example

```
> whereis ls
```

gives the output

```
ls: /bin/ls /usr/man/man1/ls.1
```

The option `-b` searches only for binary files.

The `top` command displays regularly updated reports of processes running on the system.

The `vmstat` command reports statistics about virtual memory.

Using the `uname` command we can display selected system information. When no options are provided, the operating system name is displayed. When multiple pieces of information are requested, the display order is always: operating system, network host name, operating system release, operating system version, and machine type. For example

```
> uname -a
```

provides

```
Linux issc2.rau.ac.za 2.0.34 #1 Fri May 8 16:05:57 EDT 1998 i586 unknown
```

`bash` also supports command history using the `history` command. This means that `bash` keeps track of a certain number of commands that have been entered into the shell. For example

```
> history
```

could provide the output

```
1011 ./banner
1012 ls -l
1013 cd
1014 ls -l
1015 vi first.cc
```

Chapter 3

Advanced Linux

3.1 The Unix File System

In this chapter we consider more advanced tools and techniques in LINUX. The Unix (and thus Linux) file-system is called a hierarchical file-system. A hierarchical system is a system organized by graded categorization. The file-system consists of files and directories. Files are located in directories, and directories themselves are subdirectories of their parent directories. This concept should be familiar to anyone who has ever used a computer. A key concept in Unix is that everything that is not a directory is a file. Programs are files, device drivers are files, documents are files, even the keyboard, display and mouse are represented as files. This means that if a program can write output to a file, it can also write it to the display or sound device (if we really want it to).

The Unix file-system only has one top level directory (represented by `'/'` and pronounced "root"). This differs from other systems such as DOS or OS/2 where we have several root nodes which are assigned drive letters. Anyone who has at one stage added a new hard drive to his system will be familiar with the agony of sorting out a system where drive letters have suddenly changed. Under Unix all physical devices are mounted as subdirectories of other directories.

Commonly used environment variables are `HOME`, `IFS`, `PATH`. `HOME` is the fully-qualified pathname of the user's home directory. This is where our login session starts, and it is the default target of a `cd` command. `IFS` is the internal field separator. This string consists of characters that are used to separate fields and records. We cannot see them because they have no graphical representation, but `IFS` has a default value of `Space + Tab + Newline`. The newline explains why this variable shows up with a clear line after it in the output of the `set` command. The shell uses `IFS` to determine how to break a command line into separate arguments. `PATH` identifies the directories to be searched when we type commands. The string `SHELL` identifies the shell to use when a subshell is requested. We enter at the command prompt `$HOME` we get the information about the home directory. When we enter `$PATH` we find the path.

In general, Unix systems implement security with the concepts of **users** and **groups**. A **user** identifies a single entity for access to system resources, i.e. the access is not shared. A user usually refers to a specific person although a group of people may all have access via a single user - for example a "guest" user. In contrast a **group** identifies multiple entities for shared access to system resources. For example a user "netadmin" (which administrates the network) and a user "printadmin" (which administrates the printing devices) may both be in a group "admin" which provides access to common utilities such as file manipulation and access to `/etc` for configuration.

Another example is a group "network". All users which require network access should belong to this group.

Although users and groups provide more sophisticated access control, mostly control is achieved using file system attributes. Each file in a Unix file system has attributes. Some of the attributes include

- A single user associated with the file, and the user's access rights.
- A single group associated with the file, and the group's access rights.
- The access rights for anyone who is neither the specified user nor in the specified group (other).

The attributes can be listed using the `ls -l` command. The listing includes 10 characters, the first being the file type, the following three - user permissions, the following three - group permissions and the final three are other permissions. For example, suppose a number of people are working on a project *project1* and we (*user1*) are the project leader. Executing `ls -l` in our home directory might yield

```
user1@linuxhost:~> ls -l
total 3
drwxrwx---  17 user1   project1      1234 May 16 14:36 project
-rw-------   1 user1   user1          215 May 15 18:12 mbox
-rw-rw-r--   1 user1   project1        10 Apr 20 10:16 project.status
```

On the first line we typed the command `ls -l`. On the second line we see that there are 3 files in the directory. The third line indicates that `project` is a directory (`d`) owned by `user1` and group `project1`. Both `user1` and any user in `project1` can read and write (`rw`) the directory and change to the directory (`x`). The files in the `project` directory may have different permissions. The fourth line indicates that `mbox` is a normal file (`-`) owned by `user1` and group `user1`. Only `user1` can read or write (`rw`) the file. Lastly, the file `project.status` can be read and written by `user1` and any user in `project1`, while any other user can only read the file.

To change the file ownership we use the commands `chown` (must be root) and `chgrp` and to change the permissions we use `chmod`. To find out which groups a user belongs to we use

`groups` *user*

To add users and groups see the `adduser` (or `useradd`) and `group` manual pages.

3.2 Mounting and Unmounting

A typical Linux system might have the following structure (as given by the mount command). At the command line we enter

```
> mount
```

A typical output is

```
/dev/hda1 on / type ext2 (rw)
/dev/hdb3 on /usr type ext2 (rw)
/dev/hdb2 on /usr/local type ext2 (rw)
```

`/dev/hda1` is the first partition on the first IDE hard drive, and it is mounted as root ('/'). `/dev/hdb3` is the third partition on the second IDE drive, and it mounted on `/usr` meaning that the partition will appear to be a subdirectory of `/` named `usr`. `/dev/hdb2` is further mounted below `usr` as `/usr/local`.

It is a very elegant system, and a mount point such as `/net/applications/linux` is easier to remember than a drive letter mapping.

File-systems are mounted with the `mount` command. The standard form of the `mount` command is

```
mount -t type device dir
```

where type is the file-system type of the device. The option `-t` specifies the type of the file being mounted.

The file system types which are currently supported are

```
minix, ext, ext2, xiafs, hpfs, msdos, umsdos, vfat, proc, nfs,

iso9660, smbfs, ncpfs, affs, ufs, romfs, sysv, xenix, coherent.
```

`device` is any block device found in `/dev`.

To mount an `ext2` (`ext2` stands for "second extended file-system" and is the native Linux file-system) partition of (IDE) hard drive on `/usr` we type

```
> mount -t ext2 /dev/hdb3 /usr
```

We must be logged in as root for these commands to work. If the partition is located on a SCSI hard drive the command would be

```
> mount -t ext2 /dev/sdb3 /usr
```

We can mount our Windows95 partition on `/w95` by typing:

```
> mount -t vfat /dev/hda1 /w95
```

assuming that Windows95 is on the first partition of our first IDE hard drive.

We discuss the mounting of hard drives partitions, cdroms, floppies and network file-systems.

Usually hard drive partitions will be mounted automatically at system startup, so we rarely need to mount them manually. Because floppy drives use removable media, they must be mounted before use, and unmounted before the media can be removed from the drive (disk writes are cached, so premature removal of the disk from the drive will cause data loss). To amount a DOS format stiffy in what is drive `A:` under DOS we type

```
> mount -t msdos /dev/fd0 /mnt/floppy
```

where `/dev/fd0` is the device file for the first floppy device. A disk in drive `B:` is mounted with

```
> mount -t msdos /dev/fd1 /mnt/floppy
```

The same procedure as above works for CDROMs, but the device file differs for different CDROM types. IDE CDROMs use device files similar to IDE hard drives. Thus if we have two hard drives and a CDROM, the CDROM will most probably use `/dev/hdc` as a device file (`/dev/hdb` if we only have one hard drive). Other CDROMs use different device files. For example a Sony CDROM will use `/dev/cdu31a` or `/dev/sonycd`. A Sound-Blaster CDROM will use `/dev/sbprocd` and so on.

To simplify matters a bit, a link is usually made from the specific CDROM device file to `/dev/cdrom` by typing

```
> ln -s /dev/cdu31a /dev/cdrom
```

The command to mount a CDROM will then be

```
mount /dev/cdrom -t iso9660 /cdrom
```

For any of this to work, we must have support for our specific CDROM drive compiled into the system kernel.

The `umount` command unmounts a mounted file system. The file system is specified by either its device name, its directory name, or its network path. If unmounting fails, using the `-r` option `unmount` tries to remount the file system in read-only mode.

Let us assume that the file `first.cc` in on the harddisk and we want to copy it to the floppy (msdos file system). The floppy disk must be inserted in the drive (under DOS normally called the `A:` drive).

1. First become the superuser - use the `su` ("super user" or "substitute user") command:

   ```
   $ su
   Password: <pvm>
   #
   ```

2. See what is mounted at the moment we enter:

   ```
   $ mount
   ```

3. Mount

   ```
   # mount -t msdos /dev/fd0 /mnt
   ```

4. Copy the file `first.cc`

   ```
   $ cp first.cc /mnt
   ```

5. Unmount

   ```
   $ umount /mnt
   ```

6. See if the file is on the floppy:

   ```
   $ mdir a:
   ```

Other options for the `mount` command are: `-a`: mounts all file systems in `\etc\fstab`, `-r`: mounts the file system in read-only mode, `-w`: mounts the file system in read-write mode.

After we mounted the floppy we can diplay the contents of it by using the command

```
$ ls -l /mnt
```

If we mounted the Cdrom with `mount /dev/cdrom /mnt` we can display the contents with `ls -l /mnt`. To copy the file `first.cc` on the Cdrom to the directory `cppcourse` we use the command

```
$ cp /mnt/first.cc ../cppcourse/first.cc
```

The above can become quite tedious after a while. The `/etc/fstab` file is a repository for parameters for mount points. The following listing is the contents of my `/etc/fstab` file:

```
/dev/hda1     /               ext2       defaults                 1 1
/dev/hdb6     /os2/os2d       hpfs       defaults                 0 0
/dev/hdb4     /space          ext2       defaults                 1 2
/dev/hdb3     /usr            ext2       defaults                 1 2
/dev/hdb2     /usr/local      ext2       defaults                 1 2
/dev/hdb5     swap            swap       defaults                 0 0
/dev/fd0      /mnt/floppy     ext2       noauto                   0 0
/dev/fd0      /mnt/dosfloppy  msdos      noauto,nosuid,rw,user    0 0
/dev/hdc      /cdrom          iso9660    noauto,nosuid,user,ro    0 0
none          /proc           proc       defaults                 0 0
```

The device file, mount point, file-system type and mount options are given. To mount a CDROM we only need to type `mount /cdrom`.

The option `noauto` tells the system not to mount this device at system startup. The options `nosuid` and `user` gives permission to normal users to mount and unmount devices.

The command `e2fsck` checks the state of a Linux second extended file system. This is the default file system used for Linux partitions. The syntax is

```
e2fsck [-cfncpy] [-B blocksize] device
```

Important flags and options are

```
-B blocksize: specifies a specific block size to use
   in searching for the superblock.

-c:  causes the badblocks program to be run and marks
     any bad blocks accordingly.

-f:  forces checking of file systems that outwardly seen clean.

-n: opens the file system in read-only state and answers no
    to all prompts to take action.

-p: forces automatic repairing without prompts.

-y: assumes an answer yes to all questions.
```

The device to be checked should be specified using the complete Linux device path such as

```
/dev/hda1
```

or

```
/dev/sdb3
```

It is advisable that the file system not be mounted or, if we need to check the root file system or a file system that must be mounted, that this be done in single-user mode.

3.3 MTools Package

Under Linux, a number of commands are provided to make access of DOS format floppies easier. These commands all begin with `m` (for MS-DOS). The most useful of these commands are:

- `mcd` - change MS-DOS directory
- `mcopy` - copy MS-DOS files to/from Unix
- `mdel` - delete an MS-DOS file
- `mdir` - display an MS-DOS directory
- `mformat` - add an MS-DOS file-system to a low-level formatted floppy disk
- `mlabel` - make an MS-DOS volume label
- `mmd` - make an MS-DOS subdirectory
- `mrd` - remove an MS-DOS subdirectory
- `mmove` - move or rename an MS-DOS file or subdirectory
- `mren` - rename an existing MS-DOS file
- `mtype` - display contents of an MS-DOS file
- `mattrib` - changes the attributes of a file on an MS-DOS file system
- `mbadblocks` - tests a DOS floppy disk for bad blocks.

These commands are all easy to use. For example `mdir a:` will print the directory listing of the disk in drive A: and

```
> mcopy a:file1.cc
```

will copy `file1.cc` from drive `A:` to the current directory.

The option for `mtype` are: `-s`: removes the high bit from data, `-t`: translates DOS text files to Unix text files before displaying them.

MS DOS and OS/2 use different control characters for carriage return and line feeds the Linux does. The `mcopy` command provides the switch `-t` that can be used to translate MS-DOS files to Unix and vice versa. This difference between MS-DOS and Unix may also produce problems when printing. A well known error is the staircase effect. Read the Printing-Howto for informations.

3.4 Swap Space and the Loopback Device

Like many Unix operating systems Linux uses a partition on a hard disk for swap space. The swap space is the space used by the operating system for virtual memory. When memory is not being accessed often it can be moved (copied) to the swap space (hard disk) until it is needed at a later stage when it will be copied back. In the mean time another application may use this memory. Unfortunately if we find we need more swap space it involves adding a new hard disk or re-partioning the hard disk and possibly re-installing the operating system.

Fortunately Linux can actually use any properly configured device for swap space (we actually use `/dev/hda3`, `/dev/hdb2` or some other device node representing a hard disk partition). Linux also provides us with a loopback device which can be linked to a file. Thus we can have a file in our filesystem serve as swap space. The amount of swap space is determined by the filesize. We can prepare a file using the `dd` command.

The `loconfig` command configures the loopback device. As first parameter we provide the loopback device we wish to configure, for example `/dev/loop0` or `/dev/loop1`. The second parameter is the file we associate with the loopback device. The file should not be used while associated with a loopback device as this could lead to inconsistencies.

We use the system command `mkswap` with the device node we wish to configure for swap space as the first parameter. We also use the `swapon` command to enable the swap space. Using the command `swapon -s` provides summary information about device usage as swap space.

For example, suppose we want to add 100MB of swap space to the system using a file. 100MB is 100*1024*1024 bytes. The following commands set up the swap space.

```
> dd if=/dev/zero of=/tmp/swapfile bs=102400 count=1024
> loconfig /dev/loop0 /tmp/swapfile
> mkswap /dev/loop0
> swapon /dev/loop0
> swapon -s /dev/loop0
```

3.5 Network File-System

NFS (Network File-System) is the Unix standard for sharing file-system over networks. Suppose a machine named `Chopin` wishes to mount the directory `/home/share` on a machine named `Bach` as a local directory. First `Bach` must give `Chopin` permission to do this. These permissions are called exports, and are entered into the file `/etc/exports`. To grant permission for `Chopin` to mount `/home/share` the following entry must be added to the `/etc/exports` file on `Bach`:

```
/home/share Chopin.rau.ac.za
```

Access is by default read-write, but it can be restricted to read-only. To mount this exported file system we enter

```
> mount Bach:/home/share /mnt/share
```

3.6 Linuxconf

Linuxconf is the program used to configure the Linux system. However newer distributions use their own graphical interfaces for system configuration which may be easier to use. The command we type is `linuxconf`. Linuxconf provides the following functionality.

- Console based, user friendly, interactive configuration of the system.
- Provides user customization
  1. Add, edit, delete users.
  2. Change passwords.
  3. Add system control privileges, such as system shutdown privileges.
  4. Schedule tasks for users.
- Provides network configuration.
  1. Change the hostname.
  2. Assign IP numbers to interfaces.
  3. Set the DNS server.
  4. Change routing information.
  5. Configure PPP, SLIP and PLIP.
  6. Add NFS exports.
  7. Edit `/etc/hosts` and `/etc/networks`.
- System configuration.
  1. Configure lilo. Add kernels for booting, change default kernel to boot. Add other operating systems to boot menu.
  2. Activate the configuration.
  3. Shutdown, reboot, initialize services, remove services.
  4. Set date and time.
  5. Schedule tasks for the super user.
  6. View system files.

3.7 Compressed Files

Most Linux files are stored on the installation CD-RROM in compressed form. This allows more information to be stored. When we install Linux, the installation program uncompresses many of the files transferred to our hard drive.

Any file ending with `.gz`, for example, `myfile.gz` is a compressed file. To uncompress this particular type of file, we enter

```
> gunzip filename
```

For the given file we enter

```
> gunzip myfile.gz
```

By default, `gunzip` replaces the original compressed files with the uncompressed version of the files. The option `-c` writes output to standard output, keeping original file unchanged. The command `gzip` compresses files using the Lempel-Ziv encoding. The resulting file generally replaces the original, uncompressed file and will have a `.gz` extension. For example

```
> gzip file.cc
```

The option `-c` writes output to standard output, keeping original file unchanged.

The `compress` command compresses files or standard input using Lempel-Ziv compression. The file extension is then `.Z`. For example

```
> compress first.cc
```

The uncompress the file we use `uncompress`. For example

```
> uncompress first.Z
```

To create an archive file of one or more files or directories we use the `tar` command. Although `tar` stands for tape archive, it can copy files to floppy disk or to any filename we can specify in the Linux system. The `tar` command is used because it can archive files and directories into a single file and then recreate the files and even the directory structures later. To create a `tar` file, we enter

```
> tar cvf <destination> <files/directories>
```

where `<files/directory>` specifies the files and directories to be archived and destination is where we want the `tar` file to be created. If we want the destination to be a floppy disk, we enter `/dev/fd0` as the destination. This specifies our primary floppy drive. `tar` files have the extension `.tar`. To extract a `tar` file we enter

```
> tar xvf <tarfile>
```

We can use

```
> tar tvf <filename>
```

to get an index listing of the tar file.

The command `uuencode` encodes a binary file into a form that can be used where binary files cannot (such as with some mail software). If no file is provided the standard input is encoded. The command `uudecode` decodes ASCII files created by `uuencode` to recreate the original binary file. By default the name of the decoded file will be the original name of the encoded file. If no files to decode are provided, then the standard input is decoded. The option `-o` specifies an alternate name for the resulting decoded file.

The `zip` command creates a ZIP archive from one or more files and directories. The option `-e` encrypts the archive after prompting for a password. The option `-g` adds files to an existing archive. The `unzip` command manipulates and extracts ZIP archives. The `-t` option tests the integrity of files in the archive.

The `znew` converts files compressed with `compress` (`.Z` files) into the format used by `gzip` (`.gz` files). If no files are specified, then the standard input is processed. The option `-g` uses the best, but slowest compression method.

The `zmore` command displays the contents of compressed text files, one screen at a time, allowing searching in much the same way as the `more` command.

The command `zgrep` searches one or more compressed files for a specified pattern.

The `zipinfo` command displays details about ZIP archives, including encryption status, compression type, operating system used to create the archive, and more.

The command `zcat` uncompresses one or more compressed files and displays the result to the standard output.

The `zipgrep` command searches for a pattern in one or more files in a ZIP archive using `egrep`. If no files are in the archive are specified, then all files in the archive are searched.

The command `gzexe` creates an executable compressed file. If we compress a binary file or script with `gzexe` then we can run it as if it were uncompressed. The file will simply uncompress into memory and execute, leaving the compressed version on our hard drive.

The command `cryptdir` encrypts all files in a specified directory. If no directory is specified, then all files in the current directory are encrypted. When we encrypt files, we will be prompted twice for a password. This password is needed to unencrypt files. Encrypted files will have the `.crypt` extensions added to their names. We use `decryptdir` to decrypt all files in a specified directory.

A file with the extension `.tgz` is a gzipped `tar` file. To unpack, for example `foo.tgz` we enter

```
> zcat foo.tgz | tar xvf
```

3.8 The vi and emacs Editors

3.8.1 vi Editor

vi stands for "Visual Editor". vi is popular because it is small and is found on virtually all Unix systems. People using DOS so far, find it rather strange and difficult to use, but it is extremely powerful, and flexible. To start the vi editor, we type its name at the shell prompt (command line), i.e.

```
> vi
```

If we know the name of the file we want to create or edit, we can issue the vi command with the file name as an argument. For example, to create the file first.cc with vi, we enter

```
> vi first.cc
```

To edit a given file for example first.cc we enter

```
> vi first.cc
```

When vi becomes active, the terminal screen clears and a tilde character ~ appears on the left side of very screen line, except for the first. The ~ is the empty-buffer line flag. The cursor is at the leftmost position of the first line. We probably see 20 to 22 of the tilde characters at the left of the screen. If that's not the case, check the value of TERM. When we see this display we have successfully started vi. Then vi is in command mode, waiting for our first command.

Unlike most word processors, vi starts in command mode. Before we start entering text, we must switch to input mode with the a or i keys.

Looking at vi's Two Modes

The `vi` editor operates in two modes: command mode and input mode. In command mode, `vi` interprets our keystrokes as commands. There are many `vi` commands. We can use commands to save a file, exit `vi`, move the cursor to various positions in a file, or modify, rearrange, delete, substitute, or search for text. We can even pass a command to the shell. If we enter a character as a command but the character isn't a command, `vi` beeps.

We can enter text in input mode (also called text-entry mode) by appending after the cursor or inserting before the cursor. At the beginning of the line, this doesn't make much difference. To go from command mode to input mode, press one of the following keys:

```
   a    To append text after the cursor
   i    To insert text in front of the cursor
```

We use input mode only for entering text. Most word processors start in input mode, but `vi` does not. We must go into input mode by pressing `a` or `i` before we start entering text and then explicitly press `<Esc>` to return to command mode.

There is also a command line mode. This mode is used for complex commands such as searching and to save our file or to exit `vi`. We enter this mode by typing `":"`, `"/"`, `"?"` or `"!"`. A few of these commands are:

```
 :w  - save file
 :wq - save file and exit vi
 :q! - quit without saving file.
```

`vi` has online help. Type

```
    :help
```

to access the online documentation.

Creating our First vi File

This section gives a step-by-step example of how to create a file using `vi`. If we run into difficulties, we can quit and start over by pressing `Esc`; then type `:q!` and press `<Enter>`.

1. Start `vi`. Type `vi` and press `<Enter>`. We see the screen full of flush-left tildes.

2. Go into input mode to place characters on the first line. Press the `a` key. Don't press `<Enter>`. Now we can append characters to the first line. We do not see the character `a` on-screen.

3. Add lines of text to the buffer. Type, for example the following small C++ program

   ```
   #include <iostream.h> <Enter>
                         <Enter>
   void main()           <Enter>
   {                     <Enter>
   int a = 7;            <Enter>
   int b = 8;            <Enter>
   int c = a + b;        <Enter>
   cout << "c = " << c;  <Enter>
   }                     <Enter>
   ```

   We can use the `<Backspace>` key to correct mistakes on the line we are typing.

4. Go from input mode to command mode. Press the `Esc` key. We can press `Esc` more than once without changing modes. We hear a beep from our system if we press `<Esc>` when we are already in command mode.

5. Save our buffer in a file called `first.cc`. Type `:w first.cc` and press `<Enter>`. The characters `:w first.cc` appear on the bottom line of the screen (the status line). The characters should not appear in the text. The `:w` command writes the buffer to the specified file. This command saves or writes the buffer to the file `first.cc`.

6. See our action confirmed on the status line. We should see the following on the status line:

   ```
   "first.cc" [New File] 9 lines, 178 characters
   ```

   This statement confirms that the file `first.cc` has been created, is a new file, and contains 9 lines, and 178 characters.

7. Exit `vi`. Type `:q` and press `<Enter>`.

When we type `:q`, we are still in command mode and see these characters on the status line. When we press `<Enter>`, however, `vi` terminates and we return to the logon shell prompt.

The following is a synopsis of the steps to follow:

1. Start `vi`.

   Type `vi` and press `<Enter>`.

2. Go to input mode.

   Press `a`.

3. Enter the text.

   Type the text into the buffer.

4. Go to command mode.

   Press `Esc`.

5. Save buffer to file.

   Type `:w file-name` and press `<Enter>`.

6. Quit `vi`.

   Type `:q` and press `<Enter>`.

Things to Remember about vi

- `vi` starts in command mode.
- To move from command mode to input mode, press `a` (to append text) or `i` (to insert text), respectively.
- We add text when we are in input mode.
- We give commands to `vi` only when we are in command mode.
- We give commands to `vi` to save a file and can quit only when we are in command mode.
- To move from input mode to command mode, press `Esc`.

Starting vi Using an Existing File

To edit or look at a file that already exists in our current directory, type `vi` followed by the file name and press `<Enter>`. For example

```
> vi first.cc
```

We see our C++ program on the screen. As before, tilde characters appear on the far left of empty lines in the buffer. Look at the status line: it contains the name of the file we are editing and the number of lines and characters.

Exiting vi

We can exit or quite `vi` in several ways. Remember that we must be in command mode to quit vi. To change to command mode, press `Esc`. If we are already in command mode when we press `<Esc>`, we hear a harmless beep from the terminal. Next we list the commands we can use to exit `vi`.

Command	**Action**
`:q`	Exits after making no changes to the buffer or exits after the buffer is modified and saved to a file
`:q!`	Exits and abandons all changes to the buffer since it was last saved to a file
`:wq`	Writes buffer to the working file and then exits
`:x`	Same as `:wq`
`ZZ`	Same as `:wq`

`vi` doesn't keep backup copies of files. Once we type `:wq` and press `Enter`, the original files is modified and can't be restored to its original state. We must make our own backup copies of `vi` files.

Commands to Add Text

Keystroke	Action
`a`	Appends text after the cursor position
`Shift-a`	Appends text to the end of the current line
`i`	Inserts text in front of the cursor position
`Shift-i`	Inserts text at the beginning of the current line
`o`	Opens a line below the current line to add text
`Shift-o`	Opens a line above the current line to add text

Commands to Delete Text

Keystroke	Action
`x`	Deletes character at the cursor position
`dw`	Deletes from the cursor position in the current word to the beginning of the next word
`d$`	Deletes from the cursor position to the end of the line
`Shift-d`	Same as `<d><$>`: deletes the remainder of the current line
`dd`	Deletes the entire current line, regardless of cursor position in the line

All these command can be applied to several objects – characters, words, or lines – by typing a whole number before the command. Some examples are as follows:

- Press `4x` to delete four characters
- Press `3dw` to delete three words
- Press `8dd` to delete eight lines

The Change and Replace Commands

Keystroke	Action
`r`	Replaces a single character
`Shift-r`	Replaces a sequence of characters
`cw`	Changes the current word, from the cursor position to the end of the word
`ce`	Changes the current word, from the cursor position to the end of the word (same as `<c><w>`)
`cb`	Changes the current word, from the beginning of the word to the character before the cursor position
`c$`	Changes a line, from the cursor position to the end of the line
`Shift-c`	Changes a line, from the cursor position to the end of the line (same as `<c><$>`)
`cc`	Changes the entire line

The Search Commands

Command	Action
`/string`	Searches forward through the buffer for `string`
`?string`	Searches backward through the buffer for `string`
`n`	Searches again in the current direction
`Shift-n`	Searches again in the opposite direction

A few examples:

```
    x - delete the current character.
    dd - delete the current line.
    dw - delete the current word.
    rx - replace the current character with x.
```

Most commands can be repeated

```
     5x - delete 5 characters
     5dw - delete 5 words
     5dd - delete 5 lines
     5j - move down 5 lines
```

As described above to type in text into `vi`, we must be in insert mode. To go from Normal mode to insert mode, we enter `i`. We can enter

```
    i - Insert before cursor        I - Insert at beginning of line
    a - Append after cursor         A - Append after end of line
    c - Replace word                C - Replace rest of line
    s - Replace character at cursor S - Replace whole line
    o - Open line below cursor      O - Open line above cursor
```

```
:e filename - open another file for editing.
:!shell command - suspend vi and execute shell command.
!!shell command - execute shell command and insert output into buffer
/search text -search forward for search text.
```

For example

```
/that/+1
```

places the cursor on line below the next line that contains the word `that`. The command

```
:s.x.X.g 5
```

substitute `x` by `X` in the current line and four following lines. The command

```
:1, s/now/then/g
```

replaces `now` with `then` throughout the whole file.

3.8.2 emacs Editor

Emacs is a powerful, integrated computing environment for Linux that provides a wide range of editing, programming and file management tasks. Emacs is an acronym derived from "Editor MACroS" (although vi worshipers have suggested many other explanations which range from "Eight Megabyte And Constantly Swapping" through "Easily Mangles, Aborts, Crashes and Stupifies" to "Elsewhere Maybe Alternative Civilizations Survive").

The following instructions allow us to edit our first emacs file. If we run into difficulties, we can quit and start over by pressing Ctrl-x Ctrl-c. Notice the mini buffer at the bottom of the screen; our keystrokes are appearing there because we are typing commands to the emacs editor.

1. Start emacs. Type emacs and press <Return>.

2. Add lines of text to the buffer. We enter again our small C++ program

```
     #include <iostream.h> <Enter>
                       <Enter>
     void main()           <Enter>
     {                     <Enter>
     int a = 7;            <Enter>
     int b = 8;            <Enter>
     int c = a + b;        <Enter>
     cout << "c = " << c;  <Enter>
 }                     <Enter>
```

   We can use the Backspace key to correct mistakes on the line we are typing.

3. To save our buffer in a file called first.cc we first press Ctrl-x Ctrl-s. Then at the bottom of the screen we enter first.cc. Then press <Enter> again. This command saves or writes the buffer to the file first.cc because it was the specified file. Note the number of characters in the file name. Unlike MS-DOS and Windows, Linux allows us to enter more than eight characters and a three character extension for a file name.

4. See our action confirmed on the status line. We see the following on the status line:

```
Wrote /root/first.cc
```

   This statement confirms that the file first.cc has been created, is a new file, contains 9 lines and 178 characters.

5. Exit emacs. Press <Ctrl-x><Ctrl-c> and the <Return>. emacs terminates and we return to the logon shell prompt.

The following is a synopsis of the steps you followed:

1. Start `emacs`.

   Type `emacs` and press `<Enter>`.

2. Enter the text.

   Type the text into the buffer.

3. Save buffer to file.

   Press `Ctrl-c Ctrl-s` and answer `y` to the prompt asking to save the file; then press `<Enter>` .

4. Name the file.

   Type the file name and press `<Enter>`.

5. Quit `emacs`.

   Press `Ctrl-x Ctrl-c`.

We use these steps, or variations of them, for all our editing tasks.

Starting emacs Using an Existing File

To edit or look at a file that already exists in our current directory, type `emacs` followed by the file name and press `<Enter>`. We do this with our file `first.cc`.

```
> emacs first.cc <Enter>
```

Look at the mini buffer: it contains the name of the file we are editing.

The macros which give Emacs all its functionality are written in Emacs Lisp. Emacs provides modes that are customized for the type of files you are editing. There are modes for LaTeX, C, C++, Fortran, Java, Lisp and many more. We can also send and receive E-Mail from Emacs and read Usenet news.

Here is a list of the more frequently used command sequences in Emacs: C-x means type x while holding down the control key. M-x means type x while holding down the alt or Meta key.

`C-x C-f` - Open a file
`C-x C-s` - Save a file
`C-x C-i` - Insert a file
`C-x b` - Switch to another buffer
`C-w` - Kill (cut) marked text
`C-y` - Yank (paste) killed text
`M-w` - copy marked text
`M-%` - Interactive search and replace
`M-$` - Check spelling of current word
`C-s` - Interactive search
`M-q` - Fill text
`M-u` - convert word to uppercase
`M-l` - Convert word to lowercase
`M-c` - Capitalize word
`C-h` - Display online help
`c-space` - Set mark
`C-d` - delete character at cursor position

`C-x r k` - Kill rectangle
`C-x r y` - Yank rectangle
`C-x r o` - Open rectangle
`C-x r t` - String rectangle
`C-u n command` - Repeat command n times
`C-g` - Interrupt currently executing command
`C-x (` - Define a keyboard macro
`C-x )` End macro definition
`C-x e` - Execute keyboard macro
`M-<` - Move to top of file
`M->` Move to bottom of file
`C-a` - Move to beginning of line
`C-e` - Move to end of line
`M-f` - Move forward one word
`M-d` - Delete one word

In Graphics mode we also have an emacs editor more user friendly than the text based one. We start the Graphics mode by entering

```
> startx
```

At the command prompt in Graphics mode we enter

```
> emacs
```

or

```
> emacs first.cc
```

It also allows to insert files and has search facilities.

3.9 Programming Languages

3.9.1 C and C++ Compiler

LINUX provides a C and a C++ compiler. Consider for example the C++ program

```
// first.cc

#include <iostream>
using namespace std;

int main(void)
{
   int a = 7;
   int b = 8;
   int c = a + b;
   cout << "c = " << c;
   return 0;
}
```

Instead of the extension `.cc` we could also use `.cpp` as for other compilers. To compile and link we enter at the command line the command

```
> g++ -o first first.cc
```

This provides an execute file with the name `first`. The option `-o file` places the output in file `file` in the present example `first`. To run this execute file we enter at the command line

```
> ./first
```

Obviously the output will be

```
> c = 15
```

To generate assembler code we enter

```
> g++ -S first.cc
```

The file name of the assembler code is `first.s`. The assembler code is in `AT&T` style.

To redirect the output into a file `myout` we enter

```
> ./first > myout
```

The name of the C compiler is `gcc` and the file extension for a C file is `.c`.

The header file `unistd.h` is available on all POSIX compliant systems. It includes standard symbolic constants and types. Some of the functions defined in the header file are

- `STDIN_FILENO` - the file descriptor for the standard input stream (ANSI C `stdin`)
- `STDOUT_FILENO` - the file descriptor for the standard output stream (ANSI C `stdout`)
- `STDERR_FILENO` - the file descriptor for the standard error stream (ANSI C `stderr`)
- `int chdir(const char *path)` - to change to the directory specified by `path`
- `int close(int fildes)` - to close an open file descriptor
- `int dup(int fildes)` - to create a duplicate of a file descriptor
- `int dup2(int fildes,int fildes2)` - to create `fildes2` as a duplicate of a file descriptor `fildes`
- `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp` - to execute a program, possibly modifying the path or environment
- `fork()` - to create a new process
- `int isatty(int fildes)` - to determine if the file descriptor refers to a terminal device
- `int pipe(int fildes[2])` - to create a pipe which can be used for two processes to communicate
- `ssize_t read(int fildes,void *buf,size_t nbyte)` - to read from a file identified by the file descriptor
- `ssize_t write(int fildes,const void *buf,size_t nbyte)` - to write to a file identified by the file descriptor
- `unsigned int sleep(unsigned int seconds)` - to cause a process to sleep (specified in seconds)
- `int usleep(useconds_t useconds)` - to cause a process to sleep (specified in microseconds)

More information on these functions can be obtained from the manual pages, for example type `man fork`.

3.9.2 Perl

The programming language Perl is part of Linux. For example to run a program from the command line that adds the two numbers 7 and 8 we enter

```
> perl
  $a = 7;
  $b = 8;
  $c = $a + $b;
  print($c);
```

To run the program we press `CRTL-d` under LINUX and `CRTL-z` under Windows NT.

We can also first write a program `first.pl` and then run it. Notice that the extension of a Perl program is `.pl`.

```
    # first.pl
    $a = 7;
    $b = 8;
    $c = $a + $b;
    print($c);
```

To run the program at the command line we enter

```
> perl first.pl
```

3.9.3 Lisp

LINUX also includes CLISP (Common Lisp language interpreter and compiler). For example

```
> clisp

> (car '(a b))
  a

> (cdr '(a b c))
  (b c)
```

To leave CLISP we enter

```
> (bye)
```

or `(quit)` or `(exit)`.

We can also load files into CLISP. Consider the Lisp file `first.lisp` with a function called `double`.

```
(defun double (num)
      (* num 2))
```

We can use the `load` command to load the file `first.lisp` and use the function `double`.

```
> (load "first.lisp")
  Loading first.lisp
  Finished loading first.lisp
  T
> (double 7)
  14
```

where `T` stands for true.

3.9.4 Java

There are JAVA compilers that run under LINUX. Consider for example the Java program

```
// MyFirst.java

public class MyFirst
{
   public static void main(String[] args)
   {
   int a = 7;
   int b = 8;
   int c = a + b;
   System.out.println("c = " + c);
   }
}
```

To compile we enter at the command line

```
> javac MyFirst.java
```

This generates a class file called `MyFirst.class`. This class file is platform independent. To run it we type at the command line

```
> java MyFirst
```

If the program contains `Graphics` with a `paint` method, for example `drawLine()`, `drawRect()` etc we first have to go to Graphics mode by entering at the text command line

```
> startx
```

The command `startx` starts X Windows. At the command prompt in Graphics mode the compiling and running is the same as above.

Chapter 4

Linux and Networking

4.1 Introduction

Linux provides all the commands necessary to do networking. The Internet and Email are almost synonymous these days and Linux is certainly able to handle both efficiently with mail clients such as Kmail, Spruce, Netscape and Mozilla. There are also a number of more traditional console mail programs still in use such as Pine and Mutt. Netscape is one of the more common browsers in use under Linux, featuring a familiar graphical interface and integrated mail and news. Mozilla is a project based on the Netscape source that was released in 1998. Arena, though not as flashy as some of the other Internet browsers, is a small fully functional web browser. Internet Relay Chat (IRC) is used in the Linux community as a way of working with others and getting technical assistance. One such IRC client is XChat, which is similar to the Windows programme. Another program for those that prefer console applications is BitchX.

4.2 Basic Commands

Here we summarize all the basic networking commands. The commands are

```
ifconfig
ping
netstat
mailto
nslookup
dnsdomainname
route
traceroute
hostname
rlogin
rdate
```

`ifconfig`

The `ifconfig` command configures a network interface, or displays its status if no options are provided. If no arguments are provided, the current state of all interfaces is displayed. The syntax is

```
> ifconfig interface options address
```

where `interface` specifies the name of the network interface (e.g. `eth0` or `eth1`).

To get the IP address of the host machine we enter at the command prompt the command

```
> ifconfig
```

This provides us with the network configuration for the host machine. For example

```
Ethernet adapter
   IP address        152.106.50.60
   Subnet Mask       255.255.255.0
   Default Gateway   152.106.50.240
```

Under Windows the command is `ipconfig`.

`ping`

The `ping` command sends echo request packets to a network host to see if it is accessible on the network. The syntax is

```
ping [-R] [-c number] [-d] [-i seconds] host
```

Important Flags and Options	`-c number:` Stops sending packets after the specified number of packets have been sent. `-d:` Outputs packets as fast as they come back or 100 times per second. The greater number will be generated. This option can only be used by the root user because it can generate extremely high volumes of network traffic. Care should be taken when using this option.

For example

```
> ping 152.106.50.27
```

gives the reply

```
Ping statistics for 152.106.50.27
  Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
  Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
  Reply from 152.106.50.27: bytes = 32 time = 1ms TTL = 128
  Reply from 152.105.50.27: bytes = 32 time < 10 ms TTL = 128
  Packets: Sent 4, Received 4, Lost 0 (0% loss)
  Approximate round trip i milli-seconds:
  Minimum = 0ms, Maximum = 1ms, Average 0ms
```

Another example is

```
> ping issc.rau.ac.za
```

The `ping` command also exists in Windows.

`netstat`

The `netstat` command displays network status information, including connections, routing tables, and interface statistics. When no options are provided, a list of active sockets will be displayed. The syntax is

```
> netstat [-Mnrs] [-c] [-i interface]
          [--interface interface] [--masquerade]
          [--route] [--statistics]
```

Important Flags and Options	`-c`: Displays the select information every second until `Ctrl-C` interrupts it. `-i [interface]/--interface [interface]`: Displays information about a specified interface, or all interfaces if none is specified. `-M/--masquerade`: Displays a list of masqueraded sessions. `-n`: Shows numerical addresses instead of their host, port, or user names.

For example

```
> netstat
```

An output could be

```
Active Internet Connections (w/o servers)
Proto Rec-Q Send-Q Local Address
tcp   0     2   issc2.rau.ac.za
udp   0     0   issc2.rau.ac:netbios-ns
udp   0     0   issc2.rau.ac:netbios-dgm
                    Foreign Address            State
                    mta-v13.mail.yahoo.com:smtp SYN_SENT
                    *.*
                    *.*
```

This command also exists in Windows.

`mailto`

The `mailto` command sends an e-mail to one or more recipients. If no recipients are indicated on the command line, the user will be prompted for the recipients. If no standard input is provided, then the user is prompted for the content of the message. The syntax is

```
> mailto [-a character-set]
         [-c address,] [-s subject]
         [recipient ...]
```

Important Flags	`-a character-set`: Specifies and alternate character set, such as ISO-8859-8. The default is US-ASCII. `-c address, ...`: Specifies carbon-copy addresses. `-s subject`: Specifies the subject line of the message. If the subject is more than one word, enclose it in quotation marks.

To finish composing a message, use `Ctrl-D` or type a `.` alone on a blank line `nslookup`

The `nslookup` command queries a DNS nameserver. It can be run in interactive mode. If no host name is provided, then the program enters interactive mode. By default, the DNS server specified in

```
   /etc/resolv.conf
```

is used unless another is specified. If we want to specify a server but not look up a specified host, we must provide a - in place of the host. The syntax is

```
> nslookup [host|-[server]]
```

`dnsdomainname`

The `dnsdomainname` command displays the system's DNS domain name based on its fully qualified domain name. The syntax is

```
> dnsdomainname
```

For example

```
> dnsdomainname
  rau.ac.za
```

`route`

The `route` command displays or alters the IP routing table. When no options are provided, the routing table is displayed. The syntax is

```
> route add [-net|-host
  targetaddress [netmask Nm]
  [gw Gw] [[dev] If]

> route del [-net|-host]
  targetaddress [gw Gw]
  [netmask Nm] [[dev] If]
```

Important Flags and Options	`add`: Indicates that a route is being added. `del`: Indicates that a route is being deleted. `[def] If`: Forces the route to be connected to the specified interface. `gw Gw`: Specifies the gateway for the route. `-host`: Indicates the target is a host. `-net`: Indicates the target is a network. `netmask Nm`: Specifies the netmask for the route.

For example

```
> route
```

could give the output

```
Kernel IP routing table
Destination    Gateway  Genmask         Flags  Metric Ref Use Iface
152.106.50.0     *       255.255.255.0   U       0      0   1   eth0
127.0.0.0        *       255.0.0.0       U       0      0   1   lo
default   152.106.50.240 0.0.0.0         Ub      0      0   1   eth0
```

The `route` command also exists in Windows.

`traceroute`

The `traceroute` command displays the route a packet travels to reach a remote host on the network. The syntax is

```
> traceroute [-r] host
```

Important Flags	`-i`: Specifies a network interface for outgoing packets. This is useful in system with more than one network interface. `-r`: Bypasses normal routing tables and attempts to send directly to an attached host.

For example

```
> traceroute -r 152.106.50.27
```

The output could be

```
Tracing route to zeus.rau.ac.za [152.106.50.27]
over a minimum of 30 hops
1  3ms   3ms  4ms xylan-40.rau.ac.za [152.106.240]
2 14ms   3ms  5ms 152.106.9.241
3 12ms  26ms  2ms zeus.rau.ac.za [152.106.50.27]
```

Another example is

```
traceroute -r issc.rau.ac.za
```

This command is called `tracert` in Windows.

`hostname`

The `hostname` command displays or sets the system's host name. If no flags or arguments are given, then the host name of the system is displayed. The syntax is

```
> hostname [-a] [--alias] [-d]
           [--domain] [-f] [--fqdn] [-i]
           [--ip-address] [--long] [-s]
           [--short] [-y] [--yp] [--nis]
```

Imporloginrtant Flags and Options	`-a/--alias`: Displays the alias name of host if available. `-d/--domain`: Displays the DNS domain name of the host. `-f/--fqdn/--long`: Displays the fully qualified domain name of the host. `-i/--ip-address`: Displays the IP address of the host. `-s/--short`: Displays the host name without the domain name. `-y/--yp/--nis`: Displays the NIS domain name of the system.

For example

```
> hostname
  issc2.rau.ac.za
```

To find out the hostname in Windows we can use

```
> ipconfig /All
```

or

```
> winipcfg
```

`rlogin`

The `rlogin` command logs in to a remote host.

`rdate`

The `rdate` command retrieves the current time from one or more hosts on the network and displays the returned time. The syntax is

```
> rdate [-p] [-s] host ...
```

Important Flags	`-p`: Displays the time returned from the remote system (this is the default behavior). `-s`: Sets the local system's time based on the time retrieved from the network. This can only be used by the root user.

For example

```
> rdate rauteg.rau.ac.za
  connection refused
```

4.3 email

Unix and Linux have strong support for computer-to-computer communication. To implement electronic mail, the system maintains a set of directories and files for mail messages. Each user has a file in the system's mail directory. As new mail arrives it is added to the file for the recipient and the user is informed of its arrival. Thus messages are stored on disk in mailbox files. The mail system is implemented by the `mail` program.

Assume we want to send mail to the following address

```
whs@na.rau.ac.za
```

with the subject `Hello Willi` and the message `Give me a call` we enter at the command line

```
> mail whs@na.rau.ac.za <Enter>

Subject: Hello Willi <Enter>
Give me a call <Enter>
.<Enter>    (to send message)
```

We can also end the message with `Ctrl-d` instead of a period. Remember that the `Ctrl-d` must be on a line by itself just as the period must be on a line by itself. The computer responds by displaying `EOT`, which means end of transmission.

Sending a Prepared Message
We may want to use a text editior such as `vi` to compose a message to be sent by electronic mail. If we use a text editor, we have the tools to do things such as format the text and check our spelling. It doesn't matter what program we use to create the text as long as we end up with a text or ASCII file.

Suppose that the file we want to send is named `report.txt` and that the recipient's address is `steeb_wh@yahoo.com`. There are essentially three ways to send the file, as shown in the following list. In the examples in the following list, the mail command uses the option `-s`, and the string that serves as the subject heading is surrounded by quotation marks.

Here are the three methods we can use to send mail using a prepared message file called `report.txt`.

- *Use a pipe.* To send `report.txt` with the mail command, enter

  ```
  cat report.txt | mail -s "Sales Report" steeb_wh@yahoo.com<Enter>
  ```

- *Redirect input.* To send `report.txt` with the mail command and the `-s` option, enter

  ```
  mail -s "Sales Report" steeb_wh@yahoo.com < report.txt<Enter>
  ```

- *Use* `~r` *to include a file in a message.* To use mail to send the file (using the default Subject: prompt), enter these commands:

  ```
  mail steeb_wh@yahoo.com<Enter>
  Subject: Sales Report<Enter>
  ~r report.txt<Enter>
  ~.<Enter>
  EOT
  ```

Reading our Mail
Most Linux systems notify us when we log on that we have e-mail. We can use either `mail` or another e-mail program to read any mail we have. As we read our mail, the e-mail program marks a message as read. Depending on what commands we use and how we quit the e-mail program, the message we have read are kept either in our systems mailbox,

```
/var/spool/mail/SLOGNAME
```

or in our logon directory in the file named `mbox`.

Using `mail` to Read Mail
To read our mail with `mail`, type `mail` and press `<Enter>`. Suppose that our logon name is `imgood`: we see a display similar to this:

```
mail<Enter>
mail     Type ? for help.
"/var/spool/mail/imgood": 5 message 2 new 1 unread
     1 sarah
     -Wed Jan  5  09:17  15/363
     2 croster@turn.green.com
     -Thu Jan  6  10:18  26/657  Meeting on Friday
U    3 wjones
-Fri Jan  7  08:09  32/900  Framistan Order
> N  4 chendric
-Fri Jan  7  13:22  35/1277 Draft Report
N    5 colcom!kackerma@ps.com
-Sat Jan  8  13:21  76/3103  Excerpt from GREAT new UNI
?
```

Here are some things to note about the display:

- The first line identifies the program and says to type a question mark for help.
- The second line indicates that mail is reading our system mailbox,

  ```
  /var/spool/mail/imgood
  ```

  and that we have five messages. Two have arrived since we last checked our mail, one appeared previously but we have not yet read it, and two messages have already been read.
- The five lines give information about our mail. Ignore the first few characters for now. Each line holds a message number, the address of the sender, the date the message was sent, the number of lines and characters in the message, and the subject (if one was given). Consider the following line:

  ```
  2 croster@turn.green.com  Thu Jan 6 10:18 26/657  Meeting on Friday
  ```

  This line indicates the message numbered 2 is from `croster@turn.green.com` an address that indicates the message came to our machine from a network (mail from a local user is marked with just the user's logon ID). The message was sent on Thursday, January 6, at 10:18; it consists of 26 lines and 657 characters. The subject is `Meeting on Friday`.
- A message line starting with `N` indicates new mail - mail we did not know about before.
- A message line starting with `U` indicates unread mail – mail that we know about but have not read.
- A message line without either `N` or `U` is mail we have read and save in our system mailbox.
- A greater-than character `(>)` on a message line marks the current message - the message we act on next.
- The question mark (?) on the last line is the command prompt from mail.

Quitting and Saving Changes

To quit the mail program and save the changes that occur, press `q` and `<Enter>` when we see the `?` prompt. We see the shell prompt again. When we quit mail this way, messages we read but did not delete are saved in a file named `mbox` in our home directory.

The elm Mailer

There are several different mail programs available for Linux. Each has its own advantages and disadvantages. One mail reader that comes with the Slackware distribution of Linux is the `elm` mailer. This mail program is a screen-oriented mailer rather than a line-oriented one. It provides a set of interactive menu prompts and is extermely easy to use. Virtually everything that we can do with `mail` can be done under `elm`. Since `elm` is easy to use, we only touch on the highlights of using it. We can find more in-depth information by using `elm`'s online help or by reading the man page.

Starting elm
To start a mail session with `elm`, we enter `elm` at the command prompt. If this is the first time we have used `elm`, it will prompt for permission to set up a configuration directory in our account and create an `mbox` mail file if one does not exist. Here is what we see as we start `elm` for the first time:

```
$ elm<Enter>
Notice:
This version of ELM requires the use of a .elm directory in your home
directory to store your elmrc and alias files. Shall I create the
directory .elm for you and set it up (y/n/q)? y<Return>
Great! I'll do it now.

Notice:
ELM requires the use of a folders directory to store your mail folders in.
Shall I create the directory /home/gunter/Mail for you (y/n/q)? y<Return>
Great! I'll do it now.
```

After `elm` creates its directory and `mbox` file, it runs the main mail program. This is a full-screen-oriented mailer. Our screen clears and the following is displayed

```
  Mailbox is '/var/spool/mail/gunter' with 2 messages [ELM 2,4 PL23]
  N  1  Nov  11  Jack  Tackett    Linux book
  N  2  Nov  11  Jack  Tackett    more ideas

  You can use any of the following commands
  by pressing the first character;
  d)elete or u)ndelete mail, m)ail a message,
  r)eply or f)orward mail, q)uit
  To read a message, press <return>,  j = move down,  k = move up, 7 = help

  Command:
```

At the top of the screen, `elm` tells us where our system mail box is located, how many messages are in it, and what version of `elm` we are running, `elm` then lists one line for each message in our mailbox. It places the letter `N` before each new message, just like the `mail` program. The summary line for each message tells us if the message is new, the message date, the sender, and the subject. Our display may vary slightly depending on the version of `elm`. The current message is highlighted in the list.

At the bottom of the screen is a command summary that tells us what commands we have available for the current screen. We can

delete or undelete mail,
mail a message,
reply to a message,
forward mail,
or quit.

Pressing the `j` key moves the message selection to the previous message; the `k` key moves it to the next message. Help is available by pressing the ? key. The `Command:` prompt at the bottom of the screen tells us to press a command key for `elm` to do something.

To send a file to `whs@na.rau.ac.za` named `qc.ps` we enter the following command

```
> elm -s testing whs@na.rau.ac.za < qc.ps
```

To display the current message we use `<Enter>` or `<Spacebar>`. Other `elm` commands are summerized in the following table.

Command Summary for `elm`

Command	**Description**
`|`	Pipes current messsage or tagged message to a system command
`!`	Shell escape
`$`	Resynchronizes folder
`?`	Displays online help
`+`, `<Right>`	Display next index page
`-`, `<Left>`	Displays previous index page
`=`	Sets current message to first message
`*`	Sets current message to last message
`<Number><Return>`	Set current message to `<Number>`

Command	Description
`/`	Searches subjects for pattern
`//`	Searches entire message texts for pattern
`>`	Saves current message or tagged messages to a folder
`<`	Scans current message for calendar entries
`a`	Alias, changes to 'alias' mode
`b`	Bounces (remail) current message
`C`	Copies current message or tagged messages to a folder
`c`	Changes to another folder
`d`	Deletes current message
`<Ctrl-d>`	Deletes messages with a specified pattern
`e`	Edits current folder
`f`	Forwards current message
`g`	Groups (all recipients) reply to current message
`h`	Displays header with message
`J`	Increments current message by one
`j, <Down-Arrow>`	Advances to next undeleted message
`K`	Decrements current message by one
`k, <Up-Arrow>`	Advances to previous undeleted message
`l`	limits messages by specified criteria
`<Ctrl-l>`	Redraws screen
`m`	Mails a message
`n`	Next message, displaying current, then increment
`o`	Changes `elm` options
`p`	Prints current message or tagged messages
`q`	Quits, maybe prompting for deleting, storing, and keeping messages
`Q`	Quick quit-no prompting
`r`	Replies to current message
`s`	Saves current message or tagged messages to a folder
`t`	Tags current message for futher operatons
`T`	Tags current message and go to next message
`<Ctrl-t>`	Tags messages with a specified pattern
`u`	Undeletes current message
`<Ctrl-u>`	Undeletes messages with a specified pattern
`x, <Ctrl-q>`	Exists leaving folder untouched; ask permission if folder changed
`X`	Exits leaving folder untouched, unconditionally

4.4 ftp Commands

The `ftp` application is an interface program for the ARPAnet standard file-transfer protocol. We can transfer text and binary files to and from remote sytems of any supported type. Anonymous FTP is a popular use of this application. It provides access to public databases without the need for an account on the system, where the data resides. We use `ftp` as follows.

1. Start `ftp` by typing the command `ftp` and the ftp-host at the prompt.

   ```
   $ ftp eng.rau.ac.za
   ```

2. The ftp server will ask us for your login name. Type `anonymous` and enter our email address as password (or just `pvm@`).

3. Now let us see what is available on the ftp site. Type `ls` to get a file listing:

   ```
   ftp> ls
   200 PORT command successful.
   150 Opening ASCII mode data connection for /bin/ls.
   total 8
   drwxr-x---    7 ftp    ftp       1024 Oct 31  18:00 .
   drwxr-x---    7 ftp    ftp       1024 Oct 31  18:00 ..
   d--x--x--x    2 root   root      1024 Oct 22  11:07 bin
   drwxr-xr-x    2 root   root      1024 Aug  7   1996 dev
   d--x--x--x    2 root   root      1024 Aug  7   1996 etc
   drwxr-xr-x    2 root   root      1024 Aug  7   1996 lib
   dr-xr-xr-x    5 root   root      1024 Apr 15   1997 pub
   -rw-r--r--    1 root   root       505 Feb 19   1997 welcome.msg
   226 Transfer complete.
   ftp>
   ```

4. Change into the `/pub` directory by using the `cd` command, and get a new listing.

   ```
   ftp> cd pub
   250 CWD command successful.
   ftp> ls
   200 PORT command successful.
   150 Opening ASCII mode data connection for /bin/ls.
   total 564
   dr-xr-sr-x   5 root     ftp       1024 Nov 18 15:38 .
   drwxr-xr-x   9 root     root      1024 Aug 26  1996 ..
   -rw-rw-r--   1 501      501       3562 Nov  3 14:43 bnc-2.2.0.tar.gz
   ```

```
drwxr-xr-x   2 ftp      root      1024 Jun  5  1996 irc
drwxrwsr-x   3 root     ftp       1024 Nov 18 07:36 linux
-rw-r--r--   1 root     ftp     555437 Oct 16 14:12 sdc51.zip
-rwxr-xr-x   2 root     ftp       7399 Nov 18 15:38 tear
drwxrwsr-x   2 root     ftp       1024 Feb 17  1997 uml
226 Transfer complete.
ftp>
```

5. Let us see what is the `linux` directory:

```
ftp> cd linux
250 CWD command successful.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 2382
drwxrwsr-x    3 root   ftp   1024 Nov 25 11:47 .
dr-xr-sr-x    5 root   ftp   1024 Nov 18 15:38 ..
drwxrwsr-x    2 root   ftp   1024 Mar 26  1997 MuPAD
-rw-r--r--    1 root   ftp   22824 Nov 18 07:36 freefont-0.10.tar.gz
-rw-r--r--    1 root   ftp   66 Nov 25 11:47 please_download_me
226 Transfer complete.
ftp>
```

6. Now we wish to download the file `please_download_me`. We use the `get` command. Always switch to binary mode before downloading a file.

```
ftp> bin
200 Type set to I.
ftp> get please_download_me
local: please_download_me remote: please_download_me
200 PORT command successful.
150 Opening BINARY mode data connection for please_download_me (66 bytes).
226 Transfer complete.
66 bytes received in 0.00399 secs (16 Kbytes/sec)
ftp>
```

7. To disconnect from the server type `bye`:

```
ftp> bye
221 Goodbye
$
```

Let us summarize the main commands for `ftp`.

`ascii`	sets the file transfer type to network ASCII
`bell`	sounds the bell after each file transfer
`binary`	sets the file transfer to binary mode
`bye`	terminates the FTP session with the remote server
`cd remote-directory`	changes the working directory on the remote machine
`cdup`	goes to the parent
`chmod mode file-name`	changes the permission modes of the file
`close`	terminates the FTP session with the remote server
`delete remote-file`	deletes the remote file on the remote machine
`get remote-file`	retrieves the remote file and stores it on the local machine
`ls`	lists the contents of the remote directory
`open host [port]`	establishes a connection to the specified host FTP server
`put local-file`	stores a local file on the remote machine
`pwd`	prints the name of the current working directory
`quit`	same as bye
`rmdir directory-name`	deletes a directory on the remote machine

4.5 Telnet

The telnet application is a user interface program for remote system access. We can call any remote system that is accessible on the network, but we need an account on the remote system. To start a `telnet` session we simply enter the command name at the command line.

```
> telnet hostname [port]
```

where `hostname` is the host we want to connect to and port indicates a port number. If a port number is not specified the default telnet port is used. In telnet we can log in either in one of two modes:

character-by-character
or
line-by-line.

In the character-by-character mode, most text typed is immediately sent to the remote host for processing. In line-by-line mode all text is echoed locally and only completed lines are sent to the remote host.

When we connected to a remote host, we can enter the telnet command mode by typing `Ctrl ]`. This puts us in the `telnet` command interpreter. We do this if we want to send commands directly to the `telnet` program and not to our remote computer session.

Let us look at a telnet session.

```
$ telnet hewey
Trying 152.106.50.64...
Connect to hewey.rau.ac.za,
Escape characters is '^]'.

School for Scientific Computing
RedHat Linux 4.2
Kernel 2.0.30 on an i586
login: pvm
Password:
Last login: Tue Nov 25 13:11:22 on tty1
Hewey: ~$ cd course
Hewey:~/course$ ls
Makefile          file2.student   iofile.cc    product.cc  test12.cc
add               file_to_sort    master.cc    pvm
add.cc            first.cc        names.dat    slave1.cc
add.s             gnu.cc          new.cc       slave2.cc
ex2m.cc           hosts           pmx.tex      subcourse
```

```
file1.student     iofile          pow.c        test.cc
Hewey:~/course$   exit
$
```

The default port to which `telnet` connects in 23. We can connect to another port, for example port 80 which is the port on which the www server listens:

```
$ telnet zeus 80
Trying 152.106.50.27...
Connected to zeus.rau.ac.za
Escape character is '^]'.
GET ? HTTP/1.0

HTTP/1.0 200 OK
Server: Microsoft-PWS/3.0
Date: Tue, 25 Nov 1997 11:55:45 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Mon, 01 Sep 1997 10:21:46 GMT
Content-Length: 2207

<head><title>

Home Page: Prof Willi Hans Steeb

</title></head>
...
```

Which gives us another way to read www pages (if we can decipher HTML).

4.6 Remote Shell

The `rsh` command (remote shell) allows us to start a ”remote shell” on a Unix machine on the network. For example

```
Chopin:~$ rsh hewey
Last login: Tue Nov 25 14:31:24 from Atilla
Hewey:~$ cd course
Hewey: ~/course$ ls
Makefile        file2.student   iofile.cc   pow.cc        test.cc
add             file_to_sort    master.cc   product.cc    test13.cc
add.cc          first.cc        names.dat   pvm
add.s           gnu.cc          new.cc      slave1.cc
ex2m.cc         hosts           pmx.tex     slave2.cc
file1.student   iofile          posix       subcourse
Hewey:~/course$
```

Our system is set up so that `rsh` does not ask for a password (for pvm to work properly). This is contolled by the `.rhosts` file in the `/home/pvm` directory.
The `rsh` command can be used to start a program other that shell

```
Hewey:~$ rsh atilla exec w
  4:54pm  up  3:37,  1 user,  load average: 0.00,  0.00, 0.00
USER     TTY      FROM           LOGIN@  IDLE   JCPU   PCPU  WHAT
pvm      tty1                    4:07pm 46:15   0.19s  0.08s  -bash
Hewey:~S
```

The command `rcp` (remote copy) is useful for distributing files over the network:

```
Hewey:~/course$ echo "This text is going to do some travelin" > foo
Hewey:~/course$ rcp foo dewey:~/course
```

By combining `rcp` with an interactive shell script, we can copy a file to as list of computers:

```
Hewey:~/course$  for name in dewey louie atilla
> do
> rcp foo $name:~/course
> done
Hewey:~/course$
```

4.7 Web

Netscape provides a Web browser for Linux including HTML and JavaScript.

There is a wealth of web browsers available for Linux, depending on the needs of the user. We list some of them here.

- **Lynx.** A popular console (text) based web browser. It is very fast since it does not render any graphics. Lynx supports the use of cookies, CGI forms and SSL. It is possible to browse websites with frames although this is somewhat less intuitive. No support for scripting (Javascript, Vbscript etc.) or Java.
- **Links.** Another console (text) based web browser. Similar to Lynx but can also display frames and tables.
- **Netscape.** A popular graphical web browser, with support for the newest standards including Java, Javascript and XML. Also available for windows. The only option for some advanced websites (also see Mozilla below).
- **Mozilla.** Opensource graphical web browser derived from publicly released source code from the Netscape browser. All the latest web browser features are available, tends to be an 'experimental' browser.
- **Konqueror.** The graphical web browser which comes with the KDE package. Also used for reading KDE help files. This browser is becoming very popular but is still weak with respect to Java and Javascript support. A good and stable browser for simple browsing of the internet.

4.8 INETD and Socket Programming

Inetd is a daemon which listens on specified network ports for network connections. When a client connects to one of these ports inetd starts a server program with it's standard input and output redirected to use then network connection. Thus `inetd` can be used to write simple network server programs, without knowing much about the underlying network programming. Inetd provides the following.

- Allows programs to act as servers without implementing any network programming.
- Inetd listens on a number of specified ports (see `inetd.conf`). When a connection is made, `inetd` starts the program associated with the port according to the contents of `inetd.conf`.
- Inetd redirects the standard input and output descriptors to use the socket for the connection.
- The server program reads from `stdin` and writes to `stdout` to communicate with the client.
- To reload `inetd.conf`:

    1. `ps ax | grep inetd`
       to determine the process id (*inetdpid*) of inetd
    2. `kill -HUP inetdpid`
       to reload `inetd.conf`. Type in the process id number of inetd in place of `inetpid`.

The following is an example of `/etc/inetd.conf`.

```
ftp     stream  tcp  nowait  root    /usr/libexec/ftpd         ftpd -l
telnet  stream  tcp  nowait  root    /usr/libexec/telnetd      telnetd
comsat  dgram   udp  wait    tty:tty /usr/libexec/comsat       comsat
aol     stream  tcp  nowait  root    /usr/local/bin/rshell.sh rshell.sh
```

We use the port `aol` since it is unlikely the system will ever need to use this. By default, this port is not used. Usually we would add the last line to the existing `/etc/inetd.conf` installed on the system. To select a port to use, examine the file `/etc/services` for an unused service and use that one. Alternatively we could modify `/etc/services` and add an unused port and name for use on our system only. For example we could add the line

```
rshell      60000/tcp
```

to /etc/services. We can use any port number from 0 up to and including 65535. See the services(5) man page. The port must be specified by a name from /etc/services. To illustrate the flexibility of the inetd system we use a Bourne shell script for the server. Do not use this script when connected to a network since the service provides full access to the machine. The following is the contents of /usr/local/bin/rshell.sh.

```
#!/bin/sh

echo "Remote shell"
date

while true; do
 echo Enter command:
 read command
 $command
done
```

The script asks for a command and then executes it. We could test a server using the telnet command.

```
telnet localhost aol
```

Testing this will show that every command entered fails. This is because telnet sends a carriage return and linefeed code whenever we press the enter key. The shell expects all commands to end with a linefeed code. Thus the carriage return is used as part of the command, and the command will not be recognised. We can overcome this by removing the carriage return from the command in the shell script. Alternatively, we can write our own client using sockets. This is done in the following program (rshell_client.c).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>    /*            */
#include <sys/time.h>     /* for select */
#include <unistd.h>       /*            */
#include <netinet/in.h>   /* for sockaddr_in */
#include <sys/socket.h>
#include <netdb.h>        /* for gethostbyname and others */

#define BUFSIZE 4096

int main(int argc,char *argv[])
{
 struct hostent *host;
 int s;
 struct sockaddr_in sin;
 struct servent *sp;
 fd_set fselect;
 FILE *skt;
 char buffer[BUFSIZE];

 if(argc<3)
 {
  printf("Host and service expected.\n");
  exit(EXIT_FAILURE);
 }

 host=gethostbyname(argv[1]);
 if(host==NULL)
 {
  printf("Host lookup failed.\n");
  exit(EXIT_FAILURE);
 }

 sp=getservbyname(argv[2],(char*)0);
 if (sp==NULL)
 {
  printf("Service lookup failed.\n");
  exit(EXIT_FAILURE);
 }

 s=socket(host->h_addrtype,SOCK_STREAM,
          getprotobyname(sp->s_proto)->p_proto);
 if(s==-1)
 {
  printf("Socket open failed.\n");
  exit(EXIT_FAILURE);
 }
 skt=fdopen(s,"r+");

 sin.sin_family=host->h_addrtype;
 sin.sin_port=sp->s_port;
 sin.sin_addr=*((struct in_addr*)(host->h_addr_list[0]));

 if(connect(s,(const struct sockaddr*)&sin,sizeof(sin))==-1)
 {
  close(s);
  printf("Socket connect failed.\n");
  exit(EXIT_FAILURE);
 }
```

```
 FD_SET(STDIN_FILENO,&fselect);
 FD_SET(s,&fselect);
 while((select(s+1,&fselect,NULL,NULL,NULL)>0)
        &&!feof(stdin)&&!feof(skt))
 {
  if(FD_ISSET(s,&fselect))
  {
   fgets(buffer,BUFSIZE,skt);
   fputs(buffer,stdout);
   fflush(stdout);
  }
  if(FD_ISSET(STDIN_FILENO,&fselect))
  {
   fgets(buffer,BUFSIZE,stdin);
   fputs(buffer,skt);
   fflush(skt);
  }
  FD_SET(STDIN_FILENO,&fselect);
  FD_SET(s,&fselect);
 }
 close(s);
}
```

Chapter 5

Shell Programming

5.1 What is a shell

The shell is a program used to interface between the user and the Linux kernel. The shell is a command line interpreter. Whenever we enter a command it is interpreted by the Linux shell. To understand what the shell is, and what it actually does, we review the sequence of events that happen when we log into a Unix system. We normally access a Unix system through a terminal (be it an old fashioned serial terminal, or a terminal emulator such as a xterm window) connected to certain ports on the Unix machine.

1. On a Unix system, a program called `init` automatically starts up a `getty` program on each terminal port. `getty` determines things such as the baud rate, displays the message `Login:` at its assigned terminal and then just waits for someone to type something in.

2. As soon as someone types some characters followed by `RETURN`, the `getty` program disappears; but before it goes away, it starts up a program called `login` to finish the process of loging in. It also gives `login` the characters we typed in at the terminal – characters that presumably represent our login name.

3. When `login` begins execution, it displays the string `Password:` at the terminal, and then waits for us to type our password.

4. Once we have typed our password, `login` proceeds to verify it against the encrypted entry in the `/etc/passwd` file. This file contains one line for each user on the system. That line specifies, among other things, the login name, password, home directory, and program to start up when that user logs in.

5. After `login` checks the password, it checks for the name of the program to execute. In most cases it will be `/bin/sh` (or `/bin/bash` on Linux machines), or some custom-designed application. The point is that absolutely any program can be started when a specific user logs in. The shell just happens to be the one most often selected.

When the shell starts up, it displays a *command prompt* – typically a dollar `$` sign or `>` sign – at our terminal and then waits for us to type some commands. Each time we type in a command and press the `Enter` key, the shell analyses the line we have typed and then proceeds to carry out our request. If we ask it to execute a program, the shell proceeds to *search* the disk for that program. Once found, the shell asks the kernel to initiate the programs execution, and then "goes to sleep" until the program has finished. The kernel copies the specified program into memory and begins to execute it. This program is called a *process*; in this way the distinction is made between a *program* that resides in a file on disk and a *process* that is in memory doing things.

If the program writes output to the standard output, it will appear on our terminal unless redirected or piped into a file or another command. Similarly, if the program read from standard input, it will wait for us to type something at our terminal, unless input is redirected or piped in from a file or command. When the command has finished, control once again returns to the shell which awaits our next command. This cycle continues as long as we are logged in. When we log off, execution of the shell terminates and the Unix system starts up a new `getty` at the terminal that waits for someone else to log in.

It is important to realize that the shell is just a program. It has *no* special privileges on the system, meaning that anyone with the capability and the devotion can create their own shell program. This is in fact the reason why various flavours of the shell exist today, including the standard Bourne Shell `sh`, developed by Stephen Bourne; the Korn shell `ksh`, developed by David Korn and the C shell `csh`, developed by Bill Joy.

The shell found on most Linux systems is the Bourne-again-shell `bash`, developed by the GNU project of the Free Software Foundation. This shell is compatible with the Bourne shell, but has several additional features not offered by the standard Bourne shell.

The shell has several responsibilities namely:

1. **Program execution**
   The shell is responsible for the execution of all programs that we request from our terminal. Each time we type in a line to the shell, the shell analyses the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format:

   program-name arguments

   The line that is typed to the shell is known more formally as the *command line*. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

The shell uses special characters to determine where the program name starts and ends. These characters are collectively called *whitespace* characters, and are the space character, the horizontal tab character, and the end-of-line (or newline) character. When we type the command

```
$ mv progs/wonderprog.cc outbox
```

the shell scans the command line and takes everything from the start of the line to the first whitespace character as the name of the command to execute: `mv`. The next set of characters is the first argument to the program and so forth. We note that multiple occurences of whitespace is ignored.

2. **Variable and File name substitution**
   Like any other programming language, the shell allows us to assign values to *variables*. Whenever we specify one of these variables on the command line, preceded by an dollar sign, the shell substitues the value that was assigned to the variable at that point.

   The shell also performs name substitution on the command line. The shell scans the command line looking for name substitution characters `*`, `?`, or `[...]` before determining the name of the program and its arguments to execute.

   Suppose our current directory contains the files:

```
$ ls
template.cc         thread         varargs.cc      virtual.cc
test                thread.cc      vector.cc       worm.c
test.cc             thread.o       vector.h
test.ps             varargs        virtual
```

   The echo command can be used to illustrate file name substitution:

```
$ echo *
template.cc test test.cc test.ps thread thread.cc thread.o
varargs varargs.cc vector.cc vector.h virtual virtual.cc worm.c
```

   Question. How many arguments were passed to the `echo` command? The correct answer is 14. We have to remember that the filename substitution is done by the shell before the arguments are passed to the program.

3. **Standard Input/Output and I/O redirection**
   It is the shells' responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters `<`, `>` or `>>`. When we type the command:

```
$ echo This is some text that is redirected into a file > text
```

The shell recognizes the redirection character `>` and uses the next word on the line as the name of the file to which the input stream is to be redirected. If the file already exists, it is overwritten. Use the `>>` operator to append to the file. Before the shell starts the execution of the desired program, it redirects the standard output of the program to the indicated file. The program is never aware of the fact that its output is being redirected.

Consider the following nearly identical commands:

```
$ wc -l users
      5 users
$ wc -l < users
      5
```

In the first case, the shell analyses the command line and determines that the name of the program to execute is `wc` (a program that counts the number of lines, word and bytes in a file) and that it is to be passed two arguments namely `-l` and `users`. The first argument tell it to only count the number of lines, and the second argument is the file whose lines have to be counted. So `wc` opens the file `users`, counts its lines, and then prints the count together with the file name at the terminal.

The second case, however, operates entirely differently. The shell spots the redirection character `<` when it scans the command line. The word that follows is the name of the file input is to be redirected from. Having "gobbled up" the `< users` from the command line, the shell start the `wc` program with only the one argument `-l`, and redirects its standard input from the file `users`. Seeing that it was called with only one argument, `wc` knows it has to read input from standard input. It again prints the final count, but without a filename, because it wasn't given one.

4. **Pipes**
Just as the shell scans the command line for redirection characters, it also scans for the pipe character `|`. For each such character it finds, it connects the standard output from the program preceding the `|` to the standard input of the program following the `|`. It then initiates execution of both commands. So, when we enter

```
$ who | wc -l
```

the shell connects the standard output of the command `who` (which generates a list of all users currently logged into the system) and connects it to the standard input of the command `wc -l`. The two commands execute simultaneously, and will in effect count the number of currently logged in users.

5.2 The Shell as a Programming Language

There are two ways of writing shell programs. We can type in a sequence of commands and allow the shell to execute them initeractively, or we can store those commands in a file which we can then invoke as a program.

Interactive programs

Typing in the script on the command line is a quick and easy way of trying out small code fragments. Suppose we want to find out whether the file `first.cc` exists in the present directory. Then we could enter at the command line

```
$ if [ -f first.cc ]
> then
> echo "file exists"
> fi
$
```

If the file exists, the system displays `file exists`.

Note how the shell prompt `$` changes to a `>` when we type in shell commands. The shell will decide when we have finished, and execute the script immediately. However, typing in a script every time we need it is too much trouble. The solution is to store the commands in a file, conventionally refered to as a shell script.

The following interactive script shows how to do mathematics (addition, multiplication, division) using the shell. The shell can only do integer arithmetic. Thus division is integer division. At the command line we enter

```
> count=1
> count=$((count+1))
> echo $count
  2
> count=$((count+2))
> echo $count
  4
> count=$((count*5))
> echo $count
  20
> count=$((count/7))
  2
>
```

5.2.1 Creating a Script

First, using any text editor, we create a file containing the commands we have just typed. Call the file `first.sh`

```
#!/bin/sh

# first.sh
# This program looks through all the files in the current directory
# for the string POSIX, and then prints those files to the standard
# output.

for file in *
do
  if grep -l POSIX $file
  then
     more $file
  fi
done

exit 0
```

Comments start with `#` and continue to the end of a line. Conventionally, though, `#` is usually kept in the first column. Having made such a sweeping statement, we next note that the first line, `#!/bin/sh`, is a special form of comment, the `#!` characters tell the system that the one argument that follows on the line is the program to be used to execute this file. In this case `/bin/sh` is the default shell program.

The `exit` command ensures that the script returns a sensible exit code. This is rarely checked when commands are run interactively, but if we want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important. Even if we never intend to allow our script to be invoked from another, we should still exit with a reasonable code.

A zero denotes success in shell programming. Since the script as it stands can't detect any failures, we always return success.

5.2.2 Making a Script executable

Now that we have our script file, we can run it in two ways. The simpler way is to invoke the shell with the name of the script file as a parameter, thus

```
$ /bin/sh first.sh
```

This should work, but it would be much better if we could simply invoke the script by typing its name, giving it the respectability of other Unix commands. We do this by changing the file mode to make the file executable for all users using the `chmod` command

```
$ chmod +x first.sh
```

We then execute the script using the command

```
$ ./first.sh
```

We note that it is necessary to type the

```
./
```

because for security reasons our current directory should never be in our `PATH` (especially not if we are logged in as root).

5.3 Shell Syntax

Having seen an example of a simple shell program, it is now time to look in more depth at the programming power of the shell. The shell is an easy programming language to learn, not least because it is easy to test small program fragments interactively before combining them into bigger scripts. We can use the modern Unix shell to write large structured programs.

5.3.1 Variables

Unlike C we don't declare variables in the shell before we use them. Instead we create them when we first use them, for example, when we pass an initial value to them. By default, all variables are considered and stored as strings, even when they are assigned numerical values. The shell and some utilities will convert 'numeric strings' to their values before operating on them. Unix is a case sensitive system and considers the variables `Foo`, `FOO`, and `foo` to be different.

Within the shell, we can get at the contents of a variable by preceding its name with a `$` character and outputting its contents with the `echo` command. Whenever we use them, we need to give variables a preceding `$`, except when an assignment is being made to the variable. On the command line, we can set various values of the variable `message`:

```
$ message=hello
$ echo $message
  hello
$ message="Hello, World"
$ echo $message
  Hello, World
$ message=7+11
$ echo $message
  7+11
```

Note how a string has to be delimeted by quotation marks if it contains spaces. Also note that there may be no spaces on either side of the equals sign. We can assign user input to a variable by using the `read` command. The `read` command takes one argument namely the name of the variable, and waits for the user to type in some text and press the return key.

Quoting

Before we move on we have to clear about one feature of the shell, namely the use of quoting. Normally, parameters are separated by whitespace characters, i.e. a space, a tab, or a newline character. If we want a parameter to contain one or more whitespace characters, we have to quote the parameter.

The behaviour of parameters such as `$message` inside quotes depends on the type of quotes we use. If we enclose a `$` variable expression in double quotes, it's replaced with its value when the line is executed. If we enclose it in single qoutes, no substitution takes place. We can also remove the special meaning of the `$` symbol by prefacing it with a backslash `\`.

Normally strings are enclosed in double quotes, which protects variables from being seperated by whitespace, but allows `$` expansion to take place.

The following script illustrates shell variables and quoting

```
#!/bin/bash

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

This gives the output:

```
Hi there
Hi there
$myvar
$myvar
Enter some text
Foo
$myvar now equals Foo
```

We see that the double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do. We also use the `read` command to get a string from the user.

Environment Variables

When a shell script starts, some variables are initialized from values in the environment. These are normally capitalized to distinguish them from user-defined (shell) variables in scripts, which are conventionally lowercase. The variables created will depend on our personal configuration. Many are listed in the manual pages, but the principle ones are

`$HOME`	The home directory of the current user.
`$PATH`	A colon-seperated list of directories to search for commands.
`$PS1`	A command prompt, usually `$`.
`$PS2`	A secondary prompt, used when prompting for additional input, usually `>`.
`$IFS`	An input field separator. A list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.
`$$0`	The name of the shell script.
`$#`	The number of parameters passed.
`$$`	The process id of the shell script, often used inside a script for generating unique temporary filesnames, for example `/tmp/junk_$$`.

Parameter Variables

If our script is invoked with parameters, some additional variables are created. Even if no parameters are passed, the environment variable `$#` listed above does still exist, but has a value of 0. The parameter variables are

`$1`, `$2`,...	The parameters given to the script.
`$*`	A list of all the parameters, in a single variable, separated by the character in environment variable `IFS` - the input field separator character.
`$@`	A subtle variation on `$*`, that doesn't use the `IFS` environment variable.

When the parameter expansion occurs within a double-quoted string, `$*` expands to a single field with the value of each parameter separated by the first character of the `IFS` variable, or by a space if `IFS` is unset. If `IFS` is set to a null string, which isn't equivalent to unsetting it, the parameter values will be concatenated.

For example:

```
$ IFS=''
$ set foo bar goo
$ echo "$@"
  foo bar goo
$ echo "$*"
  foobargoo
$ unset IFS
$ echo "$*"
  foo bar goo
```

Within the double quotes, `$@` expands the positional parameters as separate fields regardless of the `IFS` value. For example

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The users's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

If we run the script with

```
$ ./try_variable foo bar goo
```

we get the output:

```
Hello
The program ./try_variable is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar goo
The users's home directory is /home/gac
Please enter a new greeting
My leige
My leige
The script is now complete
```

We also can write a shell program which only contains a function and then using `$1`, `$2`, etc to pass parameters to this function. For example given the shell program

```
# mod.sh

modulus()
{
echo $(($1%$2))
}
```

At the command prompt we enter

```
$ . mod.sh
$ modulus 20 3
```

Then `$1` will be assigned to 20 and `$2` to 3. The output will be `2`.

5.3.2 Conditions

Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions. A shell script can test the exit code of any command that can be invoked from the command line, including scripts that we have written ourself. That is why it is important to always include an `exit` command at the end of any scripts that we write.

The `test` or `[ ]` Command

In practice, most scripts make extensive use of the `[ ]` or `test` command, the shell's Boolean check. On most systems these commands are synonymous. Having a command `[  ]` might seem a little odd, but actually, within the code it does make the syntax neat and more like other programming languages.

In older Unix systems, the `test` and `[ ]` commands used to call an external command `/bin/test` or `/usr/bin/test`, but in most modern Unix systems these commands are built into the shell. Since the `test` command is infrequently used outside of shell scripts, many users who have never written a shell script try to write a simple programs and call them `test`. If such a program doesn't work, it's probably conflicting with the builtin `test` command. One way to avoid this problem is by specifying the complete path to our executable when calling it, as in `./test`

To introduce the `test` command, we use it to test for the existence of a file. The command for this is `test -f <filename>`, so within a script we can write:

```
if test -f first.cc
then
  echo "file exists"
else
  echo "file does not exist"
fi
```

We can also write it like this:

```
if [ -f first.cc ]
then
  echo "file exists"
else
  echo "file does not exist"
fi
```

The `test` command's exit code (whether the condition is satisfied) determines whether the conditional code is run. We have to put spaces between the `[ ]` and the condition being checked. It helps to think of the opening brace as a command that takes the condition and the end brace as arguments.

If we prefer to put the `then` on the same line as the `if`, we must add a semicolon to separate the test from the `then`:

```
if [ -f first.cc ]; then
  echo "file exists"
else
  echo "file does not exist"
fi
```

The condition types that we can use with the `test` command fall into three categories:

String Comparison:

`string`	True if the string is not an empty string
`string1 = string2`	True if the strings are the same.
`string1 != string2`	True if the strings are not equal.
`-n string`	True if the string is not `null`.
`-z string`	True if the string is `null` (an empty string).

For example

```
$ string1="hallo"
$ string2="hallooo"
$ if [ $string1 = $string2 ] ; then
> echo "strings are the same"
> else
> echo "strings are not the same"
> fi
  strings are not the same
```

Arithmetic Comparison:

`expr1 -eq expr2`	True if the expressions are equal.
`expr1 -ne expr2`	True if the expressions are not equal.
`expr1 -gt expr2`	True if `expr1` is greater than `expr2`.
`expr1 -ge expr2`	True if `expr1` is greater than or equal to `expr2`.
`expr1 -lt expr2`	True if `expr1` is less than `expr2`.
`expr1 -le expr2`	True if `expr1` is less than or equal to `expr2`.
`!  expr`	The `!` negates the expression and returns `true` if the expression is `false`, and vice versa.

For example

```
$ number1=10
$ number2=12
$ if [ $number1 -gt $number2 ] ; then
> echo "$number1 is greater than $number2"
> else
> echo "$number1 is less than $number2"
> fi
```

The output is

```
 10 is less than 12
```

File Conditional:

`-d file`	True if the file is a directory.
`-e file`	True if the file exists.
`-f file`	True if the file is a regular file.
`-g file`	True if `set-group-id` is set on the file.
`-r file`	True if the file is readable.
`-s file`	True if the file has non-zero size.
`-u file`	True if `set-user-id` is set on the file.
`-w file`	True if the file is writable.
`-x file`	True if the file is executable.

We note that before any of the file conditionals can be true `test` looks whether the file exists. This is only a partial list of options to the `test` command. See the `bash` manual page for a complete list (type `man bash`, or `help test`).

5.3.3 Control Structures

The shell has a number of control structures, and once again they are very similar to other programming languages. For some structures (like the `case` statement), the shell offers more power. Others are just subtle syntax changes.

In the following sections, the *statements* are the series of commands that are to be performed when/while/until the *condition* is fulfilled.

`if` Command

The `if` statement is very simple. It tests the result of a command and then conditionally executes a group of statements:

```
if condition
then
  statements
else
  statements
fi
```

Example. A common use for the `if` statement is to ask a question, and then to make a decision based on the answer:

```
#!/bin/bash

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

This would give the following output:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

The script used the `[ ]` command to test the contents of the variable `timeofday`. The result of this is evaluated by the `if` command, which then allows different line of code to be executed.

`elif` Command

There are several problems with this very simple script. It will take any answer except `yes` as meaning `no`. We can prevent this using the `elif` contruct, which allows us to add a second condition to be checked when the `else` portion of the `if` is executed.

Example. We can modify our previous script so that we report an error mesage if the user types in anything other than `yes` or `no`. We do this by replacing the `else` with an `elif`, and adding another condition.

```
#!/bin/bash

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

This is quite similar to the last example, but now uses the `elif` command which tests the variable again if the first `if` command was not true. If neither of the tests are successful, an error message is printed and the script exits with the value 1, which the caller can use in a calling program to check if the script was successful.

This fixes the most obvious defect, but a more subtle problem is lurking. If we execute this new script, but instead of answering the question, we just press `<Return>`, we get the following error message:

```
[: =: unary operator expected
```

The problem is in the first `if` clause. When the variable `timeofday` was tested, it consisted of a blank string, so the `if` clause looks like,

```
if [ = "yes" ]
```

which isn't a valid condition. To avoid this problem, we have to put quotes around the variable,

```
if [ "$timeofday" = "yes" ]; then
```

so an empty string gives us a valid test:

```
if [ "" = "yes" ]
```

`for` Command

We use the `for` construct to loop through a range of values, which can be any set of strings. They could simply be listed in the program or, more commonly, the result of a shell expansion of filenames. The syntax is:

```
for variable in values
do
  statements
done
```

Example. The first example shows the use of `for` with fixed strings.

```
#!/bin/bash

for count in one two three four
do
  echo $count
done
exit 0
```

The output is:

```
one
two
three
four
```

Example. The next example uses the `for` construct with wild card expansion.

```
#!/bin/bash

for file in $(ls chap[345].txt); do
  lpr $file
done
```

This script also illustrates the use of the `$(command)` syntax, which will be reviewed in more detail later. The parameter list for the `for` command is provided by the output of the command enclosed in the `$()` sequence. This script will send the files called `chap3.txt`, `chap4.txt` and `chap5.txt` to the default system printer via the `lpr` command.

`while` Command

Since all shell variables is considered to be strings, the `for` loop is good for looping through a series of strings, but a little awkward to use when we wish to execute a command a fixed number of times. It can become tedious if we want to use `for` to repeat a command say twenty times

```
#!/bin/bash

for foo in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
  echo "Here we go again"
done
exit 0
```

Even with wild card expansion, we might be in a situation where we just don't know how many times we need to loop. In that case we can use a `while` loop, which has the following syntax:

```
while condition
do
  statements
done
```

Example. By combining the `while` construct with arithmetic substitution, we can execute a command a fixed number of times. This is less cumbersome than the `for` loop we used earlier.

```
#!/bin/bash

foo=1

while [ "$foo" -le 20 ]
do
  echo "Here we go again"
  foo=$(($foo+1))
done

exit 0
```

This script uses the `[ ]` command to test the value of `foo` against the value `20` and executes the while loop if it's smaller or equal. Inside the `while` loop the syntax `$(($foo+1))` is used to perform arithmetic evaluation of the expression inside the braces, so `foo` is incremented by one each time around the loop.

In the next program we use two nested `while` loops to display a triangle of asterisks on the screen.

```
> outer=1
> while [ $outer -le $1 ]
> do
> inner=1
> while [ $inner -le $outer ]
> do
> echo -n "*"
> inner=$(($inner+1))
> done
> echo
> outer=$(($outer+1))
> done
>
> exit 0
```

From the command line we pass the number of lines of asterisks we want to display.

`until` Command

The `until` statement has the syntax:

```
until condition
do
  statements
done
```

This is very similar to the `while` loop, but the condition test is reversed. In other words, the loop continues until the condition becomes true, not while the condition is true.

Example. Often the `until` statement adds clarity. It also fits naturally when we want to loop forever until something happens. As an example, we can set up an alarm which goes off when another user logs in:

```
#!/bin/bash

until who | grep "$1" > /dev/null
do
  sleep 10
done

# ring the bell and announce the users' presence
```

```
echo -e \\a
echo "$1 has just logged in!!!"

exit 0
```

The `-e` option to `echo` enable interpretation of the following backslash-escaped characters in the strings:

`\a`	alert (bell)
`\b`	backspace
`\c`	suppress trailing newline
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\nnn`	the character whose ASCII code is nnn (octal)

`case` Command

The `case` statement is a little more complex than the other structures we have discussed up to now. Its syntax is:

```
  case variable in values
    pattern [ | pattern] ...)  statements;;
    pattern [ | pattern] ...)  statements;;
    ...
  esac
```

The `case` construct allows us to match the contents of a variable against patterns in quite a sophisticated way, and then allows execution of different statements, depending on which pattern was matched. This ability to match multiple patterns and execute multiple statements makes the `case` construct a good way of dealing with user input.

Example. We present a version of our input testing script that uses `case`.

```
#!/bin/bash

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
```

```
   "yes") echo "Good Morning";;
   "no" ) echo "Good Afternoon";;
   "y"  ) echo "Good Morning";;
   "n"  ) echo "Good Afternoon";;
   *    ) echo "Sorry, answer not recognized";;
esac

exit 0
```

When the `case` statement is executing, it compares the contents of `timeofday` to each string in turn, and executes the statements following the first pattern that matches. The `case` command performs normal expansion, so that the `*` is a wild card that will match any string. The asterisk `*` is used as the default action to be performed if no other pattern is satisfied.

Example. By putting patterns with the same actions together we can make a much cleaner version:

```
#!/bin/bash

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
   "yes" | "y" | "Yes" | "YES" )
   echo "Good Morning";;
   "n*" | "N"  )
   echo "Good Afternoon";;
   *    )
   echo "Sorry, answer not recognized";;
esac

exit 0
```

Example. To make the script reusable, we have to change it to return an error code when the default pattern had to be used. To do this we need to execute more than one statement when a certain `case` was matched:

```
#!/bin/bash

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
   "yes" | "y" | "Yes" | "YES" )
         echo "Good Morning"
         ;;
   "[nN]*")
         echo "Good Afternoon"
         ;;
   *)    echo "Sorry, answer not recognized"
         echo "Please answer yes or no"
         exit 1
         ;;
esac

exit 0
```

To show a different way of matching multiple patterns, we have changed the ‘no’ pattern to ‘`[nN]*`’ which is read as “an ‘n’ or an ‘N’ followed by anything”.

5.3.4 Logical AND and Logical OR

Sometimes we want to connect commands together in a series. For instance, we may want several different conditions to be met before we execute a statement, for example:

```
if [ -f this_file]; then
   if [ -f that_file]; then
      if [ -f the_other_file]; then
         echo "All files are present and correct"
      fi
   fi
fi
```

or we may want at least one of a series of conditions to be met:

```
if [ -f this_file]; then
   foo="True"
elif [ -f that_file]; then
   foo="True"
elif [ -f the_other_file]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo" = "True"]; then
   echo "One of the files exists"
fi
```

Although these can be implemented using multiple `if` statements, the results are awkward. The shell has a special pair of constructs for dealing with lists of commands: the *AND* list and the *OR* list.

The *AND* List

The AND list construct allows us to execute a series of commands, executing the next commnd only if all of the preceding commands have succeeded. The syntax is:

*statement*1 `&&` *statement*1 `&&` *statement*1 `&&` ...

We notice that `&&` is the logical AND in languages such as C, C++ and Java. Starting at the left, each statement is executed and, if it returns `true`, the next statement to the right is executed. This continues until a statement returns `false`, or when no more statements in the list are executed. The `&&` tests the condition of the preceding command. Each statement is executed independently, allowing us to mix many different commands in a single list. The AND list as a whole succeeds (true) if all commands are executed successfully, and it fails (false) otherwise.

Example. In the following script, we use the command

```
touch file_one
```

which checks whether the file exists and create it if it does not. Then we remove `file_two`. The `touch` command changes the access and modification times of a file, or creates a new file with specified times. Then the AND list tests for the existence of each of the files and echoes some text in between

```
#!/bin/bash

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
   echo "in if"
else
   echo "in else"
fi

exit 0
```

The script gives the output:

```
hello
in else
```

The `touch` and `rm` commands ensure that the files are in a known state. The `&&` list then executes the `[ -f file_one ]` statement, which succeeds because we have just made sure that the file existed. Since the previous statement succeeded, the `echo` command is called. This also succeeds. The `echo` command always returns `true`. The third test, `[ -f file_two ]` fails since we have just erased `file_two`, therefore the last statement `echo "there"` is not executed. Since one of the commands in the AND list failed, the AND list as a whole returns `false` and the statement in the `else` condition is executed.

The ***OR*** list

The OR list construct allows us to execute a series of commands until one succeeds. the synax is:

*statement*1 `||` *statement*1 `||` *statement*1 `||` ...

Starting at the left, each statement is executed. If it returns `false`, the next statement to the right is executed. This continues until a statement returns `true`, or when no more statements are executed.

```
#!/bin/bash

rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
   echo "in if"
else
   echo "in else"
fi

exit 0
```

This will give us the output:

```
hello
in if
```

The first test `[ -f file_one ]` fails, and the `echo` command has to executed. `echo` returns `true` and no more statements in the OR list is executed. The `if` succeeds because one of the commands in the OR list was `true`.

Statement blocks

If we want to use multiple statements in a place where only one is allowed, such as in an AND or an OR list, we can do so by enclosing them in braces `{ }` to make a statement block. For example

```
#!/bin/bash

tempfile=temp_$$

yes_or_no() {
    while true
do
   echo "Enter yes or no"
   read x
   case "$x" in
      y | yes ) return 0;;
  n | no )  return 1;;
  * ) echo "answer yes or no"
       esac
    done
}

    yes_or_no && {
   who > $tempfile
   grep "secret" $tempfile
    }
```

Depending on the return value of the function `yes_or_no` the following block of statements will be executed.

5.3.5 Functions

The shell provides functions, and if we are going to write any large scripts, it is a good idea to use functions to structure our code. To define a shell function, we simply write its name, followed by empty `()` parentheses and enclose the statements in `{ }` braces. The syntax is

```
function_name() {
  statements
}
```

For example

```
#! /bin/bash

foo() {
    echo "Function foo is executing"
}

echo "script is starting"
foo
echo "script ended"

exit 0
```

Running the script will show:

```
script is starting
Function foo is executing
script ended
```

As always, the script starts executing at the top. When it finds the `foo() {` construct, it knows that a function called `foo` is being defined. It stores the fact `foo` refers to a function and continues executing after the matching }. When the single line `foo` is executed, the shell now knows to execute the previously defined function. When this function completes, execution resumes at the line after the call to `foo`.

We must always define a function before it can be executed. This isn't a problem in the shell, since all scripts start executing at the top. Thus we put all the functions before the first call of any function.

Function Arguments

When a function is invoked, the positional parameters to the script,

```
$*,  $@,  $#,  $1,  $2
```

and so on are replaced by the parameters to the function. That is how we read parameters passed to the function. When the function exits, they are restored to their previous values.

For example

```
#! /bin/bash

foo() {
    echo "The argument to the function is $1 $2"
}

foo "bar" "willi"

exit 0
```

The script will give the output:

```
The argument to the function is bar willi
```

For example

```
#! /bin/bash

foo() {
    echo "The argument to the function is $@"
}

foo "bar" "willi" "olla"

exit 0
```

The script will give the output

```
The argument to the function is bar willi olla
```

Returning Values

Shell functions can only return numeric values with the `return` command. The only way to return string values is to store them in a global variable, which can then be used after the function finishes.

For example

```
#! /bin/bash

yes_or_no() {
    echo "Parameters are $*"
    while true
    do
        echo "Enter yes or no"
        read x
        case "$x" in
            y | yes ) return 0;;
            n | no ) return 1;;
            * ) echo "answer yes or no"
        esac
    done
}

echo "original parameters are $*"

if yes_or_no "Is your name $1"
then
    echo "Hi $1"
else
    echo "Never mind"
fi

exit 0
```

Running the program with

```
$ ./function4.sh John and Ben
```

gives

```
original parameters are John and Ben
Parameters are Is your name John
Enter yes or no
dunno
answer yes or no
```

```
Enter yes or no
yes
Hi John
$
```

Local variables

Local variables can be declared inside shell function by using the `local` keyword. The variable is then only in scope within the function. Otherwise, the function can access the other shell variables which are essentially global in scope. If a local variable has the same name as a global variable, it shadows that variable but only within the function. For example

```
#! /bin/bash

text="global variable"

foo() {
    local text="local variable"

    echo "Function foo is executing"
    echo $text
}

echo "script is starting"
echo $text

foo

echo "script ended"
echo $text

exit 0
```

The output is

```
script is starting
global variable
Function foo is executing
local variable
script ended
global variable
```

If we execute an `exit` command from inside a function, its effect is not only to terminate execution of the function, but also of the shell program that called the function. If we instead would like to just terminate execution of the function, we can use the `return` command, whose format is

```
return n
```

The value `n` is used as the return status of the function. If omitted, then the status returned is that of the last command executed. The value of its return status we can access with `$?`. For example

```
x=0

foo()
{
if [ $1 -eq 0 ]
then
x=7
return 0;
else
x=9
return 1;
fi
}
```

Calling the function `foo` with

```
foo 5
echo $x
echo $?
```

provides the output `9` and `0`. However

```
foo 5
echo $x $?
```

provides the output `9` and `1`. Calling the function `foo` with

```
foo 0
echo $x
echo $?
```

provides the output `7` and `0`.

5.4 Shell Commands

5.4.1 Builtin Commands

We can execute two types of commands from shell scripts. There are the 'normal' commands that we could also execute from the command prompt and there are the 'builtin' commands that we mentioned earlier. These builtin commands are implemented internally to the shell and can not be invoked as external programs. Most internal commands are, however, also provided as stand-alone programs–it's part of the POSIX specification. It generally does not matter whether the command is external or internal, except that the internal commands execute more efficiently.

`break` Command

We use this command for escaping from an enclosing `for`, `while` or `until` loop before the controlling condition has been met. For example

```
#! /bin/bash

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
    if [ -d "$file" ]; then
        break
    fi
done

echo first directory starting with fred was $file

exit 0
```

The output is

```
$ fred3
```

We recall that the option `-d` tests whether the file is a directory. We recall that the command `echo > fred1` creates the file `fred1`. The size of the file is 1 byte. Explain why?

":" Command

The colon command is a null command. It is occasionally useful to simplify the logic of conditions, being an alias for `true`. Since it is a builtin command, it runs faster than `true`. We may see it used as a condition for `while` loops: `while :` implements an infinite loop, in place of the more conventional `while true`.

`continue` Command

Rather like the C keyword of the same name, this command make the enclosing `for`, `while` or `until` loop continue at the next iteration, with the loop variable taking the next value in the list. For example

```
#! /bin/bash

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for i in fred*
do
    if [ -d "$i" ]; then
        continue
    fi
    echo i is $i
done

exit 0
```

We run the program with

```
$ ./continue.sh
```

The output is:

```
i is fred1
i is fred2
i is fred4
$
```

"." Command

The dot command executes the command in the current shell. The syntax is

```
  .  shell_script
```

Normally when a script executes an external command or script, a new shell (a sub shell) is spawned, the command is executed in the sub shell and the sub shell is then discarded, apart from the exit code which is returned to the parent shell. The external `source` and the internal dot command run the commands in the same shell that called the script. Usually any changes to environment variables that take place in a script is lost when the script finishes. The dot command, however, allows the executed command to change the current environment. This is often useful when we use a script as a wrapper to set up our environment for the later execution of some command.

In shell scripts, the dot command works like the `#include` directive in C or C++. Though it doesn't literally include the script, it does execute the script in the current context, so we can use it to incorporate variable and function definitions into a script.

`echo` Command

The `echo` is also a builtin command. We have been using it throughout, so it won't be discussed further. The X/Open standard has introduced the `printf` command in newer shells, and it should rather be used for outputting strings.

`exec` Command

The `exec` command has two different uses. It is normally used to replace the current shell with a different program. For example

```
exec wall "Thanks for all the fish"
```

in a script will replace the current shell with the `wall` command. No lines in the script after the `exec` will be processed, because the shell that was executing the script no longer exists. The second use of `exec` is to modify the current file descriptors.

```
exec 3 < afile
```

This causes file descriptor three to be opened for reading from file `afile`. It is rarely used.

`exit` *n* Command

The `exit` command causes the script to exit with exit code `n`. If we use it at the command prompt of any interactive shell, it will log us out. If we allow the script to exit without specifying the exit status, the status of the last command execute in the script will be used as the return value. In shell programming, exit code 0 is success, codes 1 through 125 inclusive are error codes that can be used by scripts. The remaining values have reserved meanings.

`export` Command

The `export` command makes the variable being exported visible to sub shells. By default variables created in a shell are not available in further sub shells invoked from that shell. The `export` command creates an environment variable from its parameter which can be seen by other scripts and programs invoked from the current program.

`expr` Command

The `expr` command evaluates its arguments as an expression. It is most commonly used for simple arithmetic. `expr` can perform many expression evaluations:

Expression	Description
`expr1 | expr2`	`expr1` if `expr1` is non-zero, otherwise `expr2`.
`expr1 & expr2`	Zero if either expression is zero, otherwise `expr1`
`expr1 = expr2`	Equal
`expr1 > expr2`	Greater than.
`expr1 >= expr2`	Greater than or equal to.
`expr1 < expr2`	Less than.
`expr1 <= expr2`	Less than or equal to.
`expr1 != expr2`	Not equal.
`expr1 + expr2`	Addition.
`expr1 - expr2`	Subtraction.
`expr1 * expr2`	Multiplication.
`expr1 / expr2`	Integer division.
`expr1 % expr2`	Integer modulo.

For example

```
a=1
a=‘expr $a + 1‘
echo $a
```

The output is 2. Note that in newer scripts, `expr` is normally replaced with the more efficient `$((...))` syntax.

`eval` Command

`eval` [*arg* ...]

The args are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the `eval` command. If there are no args, or only null arguments, `eval` returns true. For example

```
#!/bin/bash

command="ls"
files="*.sh"

eval $command $files

exit 0
```

The output displays all the files with the extension `.sh`.

`printf` Command

The `printf` command is only available in more recent shells. The syntax is:

`printf "format string"` *parameter*1 *parameter*2 `...`

The format string very similar to that used in C or C++, with some restrictions. Floating point arithmetic is not supported. All arithmetic in the shell is performed as integers. The format string consists of any combination of literal characters, escape sequences and conversion specifiers. All characters in the format string, other than `%` and \, appear literally in the output.

The following escape sequences are supported:

Escape Sequence	Description
\\	Backslash character.
`\a`	Alert (rings the bell or beep).
`\b`	Backspace character.
`\f`	Form feed character.
`\n`	Newline character.
`\r`	Carriage return.
`\t`	Tab character.
`\v`	Vertical tab character.
`\ooo`	The single character with actual octal value `ooo`

The conversion specifier is quite complex, so we list only the commonly used ones here. More details can be found in the manual. Enter `man bash` at the command prompt. The conversion specifier consists of a `%` character, followed by a conversion character. The principal conversions are:

Conversion Specifier	Description
`d`	Output a decimal number.
`c`	Output a character.
`s`	Output a string.
`%`	Output the `%` character.

The format string is then used to interpret the remaining parameters and output the result. For example

```
$ printf "%s\n" hello
hello
$ printf "%s %d\t%s" "Hi there" 15 people
Hi there 15   people
```

Note how we have to use `" "` to protect the `Hi there` string and make it a single parameter.

`set` and `unset` Commands

The `set` command sets the parameter values for the shell. It can be a useful way of setting fields in commands that output space-separated values. For example, `set x y z` assigns `x` to `$1`, `y` to `$2` and `z` to `$3`. The output

```
set x y z
echo $2
```

is `y`. Without any argument `set` gives in alphabetical order the list of all of the variables that exists in our environment.

Suppose we want to use the name of the current month in a shell script. The system provides a `date` command, which gives the date in a format that contains the name of the month as a string, but we need to separate it from the other fields. We can do this using a combination of the `$(...)` construct to execute the `date` command and return the result and the `set` command. The `date` command output has the month string as its second parameter

```
#! /bin/bash

echo the date is $(date)
set $(date)
echo The month is $2

exit 0
```

The script output is

```
the date is Thu Nov 13 23:16:33 SAT 1997
The month is Nov
```

We can also use the `set` command to control the way the shell executes, by passing it parameters. The most commonly used is `set -x` which makes a script display a trace of its currently executing command. The `unset` command is used to remove a variable from the environment. It cannot do this to read-only variables (such as `USER` or `PWD`).

`shift` Command

The `shift` command moves all the parameter variables down by one, so `$2` becomes `$1`, `$3` becomes `$2`, and so on. The previous value of `$1` is discarded, while `$0` remains unchanged. If a numerical argument is specified the parameters will move that many spaces. The variables `$*`, `$@` and `$#` are also modified to reflect the new arrangement of parameter variables. The `shift` command is often useful for scanning through parameters, and if we write a script that takes ten or more parameters, we have no choice but to use the `shift` command. For example

```
#!/bin/bash
# shell1.sh

while [ "$1" != "" ]; do
   echo "$1"
   shift
done

exit 0
```

If we run the script with

```
> ./shell1.sh willi hans steeb
```

the output is

```
willi
hans
steeb
```

`trap` Command

The `trap` command is used for specifying the actions to take on receipt of a signal. A common use is to tidy up a a script when it is interrupted. The `trap` command is passed the action to take, followed by the signal name (or names) to trap on. The `trap` *command signal*

command will be executed whenever any of the signals specified by `signal` is received. Remember that scripts are normally interpreted from 'top' to 'bottom' so we have to specify the `trap` command before the part of the script we wish to protect. To reset a trap condition to the default, simply specify the action as `-`. To ignore a signal, set the action to the empty string `''`. A `trap` command with no parameters prints out the current list of traps and actions.

The following table lists the more important signals covered by the X/Open standard that can be caught:

Signal(*number*)	Description
`HUP` (1)	Hang up; usually sent when a terminal goes offline, or a user logs out.
`INT` (2)	Interrupt; usually sent by pressing `<Ctrl-C>`.
`QUIT`(3)	Quit; usually sent by pressing `<Ctrl-\>`.
`ABRT`(6)	Abort; usually sent on some serious execution error.
`ALRM`(14)	Alarm; usually used for handling time-outs.
`TERM`(15)	Terminate; usually sent by the system when it's shutting down.

For example, in the following program the process `id` number of the program being executed.

```
#! /bin/bash

trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (CRTL-C) to interrupt ... "
while [ -f /tmp/my_tmp_file_$$ ]; do
    echo File exists
    sleep 1
done

echo The file no longer exists

trap - INT

echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (CRTL-C) to interrupt ... "
while [ -f /tmp/my_tmp_file_$$ ]; do
```

```
    echo File exists
    sleep 1
done

echo We never get here
exit 0
```

To run the script we enter

```
> ./trap.sh
```

The output from this script is

```
creating file /tmp/my_tmp_file_1010
press interrupt (CRTL-C) to interrupt ...
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_1010
press interrupt (CRTL-C) to interrupt ...
File exists
File exists

[1]+  Terminated              ./trap.sh
```

This script uses the `trap` command to arrange for the command

```
rm -f /tmp/my_tmp_file_$$
```

to be executed when an `INT` (interrupt) signal occurs, The script then enters a `while` loop which continues while the file exists. When the user presses `<Ctrl-C>`, the statement

```
rm -f /tmp/my_tmp_file_$$
```

is executed, then the `while` loop resumes, since the file has now been removed, the first `while` loop terminates normally.

The script then uses the `trap` command again, this time to specify that no command is to be executed when an `INT` signal occurs. It then recreates the file and loops inside the second `while` construct. When the user presses `<Ctrl-C>` this time, there is no handler installed for the signal, so the default behavior occurs, which is to immediately terminate the script. Since the script terminates immediately, the final `echo` and `exit` statements are never executed.

5.4.2 Command Execution

When we are writing scripts, we often need to capture the output from a command's execution for use in the script, i.e. we want to execute a command and put the output of the command in a variable. We do this by using the `$(command)` syntax. There is also an older form, `` `command` ``, which is still in common usage.

All new script should use the `$(...)` form, which was introduced to avoid some rather complex rules covering the use of the characters `$`, `` ` `` and \ inside the back quoted command. If a back quote is used inside the `` `...` `` construct, it must be escaped with a \ character. The result of the `$(command)` is simply the output from the command. Note that this is not the return status of the command, but the string output. For example

```
#! /bin/bash

echo "The current working directory is $PWD"
echo "The current users are $(who)"

exit 0
```

Because the current directory is a shell environment variable, the first line doesn't need to use this command execution construct. The result of `who`, however, does need this construct if it is to be available to the script. The concept of putting the result of a command into a script variable is very powerful, as it make it easy to use existing commands in scripts and capture their output. A problem sometimes arises when the command we want to invoke outputs some whitespace before the text we want, or more output than we actually require. In such cases we can use the `set` command as we have already shown.

Arithmetic Expansion

We have already used the `expr` command, which allows simple arithmetic commands to be processed, but this is quite slow to execute, since a new shell is invoked to process the `expr` command. A newer and better alternative is `$((...))` expansion. By enclosing the expression we wish to evaluate in `$((...))`, we can perform simple arithmetic much more efficiently. For example

```
#! /bin/bash

x=0
while [ "$x" -ne 10 ]; do
    echo $x
    x=$(($x+1))
done

exit 0
```

Parameter Expansion

We have seen the simplest form of parameter assignment and expansion, where we write:

```
foo=fred
echo $foo
```

A problem occurs when we want to append extra characters to the end of a variable. Suppose we want to write a script to process files called `1_temp`, `2_temp` and `3_temp`. We could try:

```
#!/bin/bash

for i in 1 2 3
do
    cat $i_temp
done
```

But on each loop, we get the error message

```
   too few arguments
```

What went wrong? The problem is that the shell tried to substitute the value of the variable `$i_temp`, which doesn't exist. To protect the expansion of the `$i` part of the variable, we need to enclose the `i` in `{ }` like this:

```
#!/bin/bash

for i in 1 2 3
do
    cat ${i}_temp
done
```

On each loop, the value of `i` is substituted for `${i}`, to give the actual file names.

We can perform many parameter substitutions in the shell. Often, these provide an elegant solution to many parameter processing problems. The common ones are:

Parameter Expansion	Description
`${param:-default}`	If `param` is null, set it to the value of `default`.
`${#param}`	Gives the length of `param`.
`${param%word}`	From the beginning, removes the smallest part of `param` that matches `word` and returns the rest.
`${param%%word}`	From the beginning, removes the longest part of `param` that matches `word` and returns the rest.
`${param#word}`	From the end, removes the smallest part of `param` that matches `word` and returns the rest.
`${param##word}`	From the end, removes the longest part of `param` that matches `word` and returns the rest.

These substitutions are often useful when we are working with strings. The last four that remove parts of strings are especially useful for processing filenames and paths, as the example below shows. For example

```
#! /bin/bash

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11R6/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

This gives the output:

```
bar
fud
usr/bin/X11R6/startx
startx
/usr/local/etc/
/usr/
```

Since Unix is based around filters, the result of one operation must often be redirected manually. For example we want to convert a `gif` file into a `jpeg` file using the `cjpeg` filter available with most `Linux` distributions. We can type the following:

```
$ cjpeg image.gif > image.jpg
```

Sometimes, however, we want to perform this type of operation on a large number of files. The following script will automate this process:

```
#!/bin/bash
for image in *.gif
do
    cjpeg $image > ${image%%gif}jpg
done
```

5.5 Advanced examples

Shell programming can be quite flexible. Consider self extracting archives (common in the Windows environment). This can easily be implemented using shell scripts. The tar archive for example leads to the idea of a shar or shell archive. Consider the shell script `shar.sh`

```
#!/bin/sh

tar xvfz << "EOF"
```

The redirection `<< "EOF"` is for the so called “HERE” document, i.e. the file used for redirection follows this line and ends when `EOF` appears at the beginning of a line on its own. We could use any word we want to end the redirection. The double quotes around `EOF` indicate to the shell that lines should not be interpreted (shell variables should not be expanded) Now the following commands construct a self extracting archive.

```
> tar cfz newar.tar.gz some_directory
> cat shar.sh newar.tgz > newar.shar
> echo "EOF" >> newar.shar
> chmod +x newar.shar
```

To extract the shell archive we simply use the command

```
> ./newar.shar
```

which uses `tar` to extract the archive. Thus the shell script and the archive are contained in the same file.

We describe a way to use `ftp` to do a parallel download, i.e. given n the number of `ftp processes` to use we download a file in n pieces of equal size using n `ftp` processes running simultaneously.

We use the `reget` command of `ftp`. Suppose we know the size (use the `size` command in `ftp`). We divide the file into n chunks of equal size. For each chunk we

> dd if=/dev/zero of=*filename*.*n* count=*n* bs=*chunksize*

We start `ftp` and use the command

> reget *filename* *filename*.*n*

For each of the n files (at the same time). When each ftp has downloaded at least *chunksize* bytes we stop the `ftp` session. Then we execute for $i = 0$, $i = 1$, ..., $i = n - 1$

> dd if=*filename*.*i* bs=*chunksize* count=1 skip=*i* >> *filename*

to obtain the desired file *filename*. The following script automates the process.

```
#!/bin/sh

#Assume $1=n processes $2=host $3=user $4=password $5=filename

n="$1"
host="$2"
ftpuser="$3"
ftppass="$4"
filename="$5"
file=$(basename "$filename")

#obtain the filesize
name=$(\
{
 echo user "$ftpuser" \'"$ftppass"\'
 echo bin
 echo size "$filename"
 echo quit
} | ftp -n "\$host" | \
{
 while [ "$name" != "$filename" ]
 do
  read name filesize
  echo -n $name $filesize
 done
})

set --
set $name
filesize=$2
chunksize=$(($n-1))
chunksize=$(($filesize/$chunksize))
if [ "$chunksize" -eq 0 ]
then
 chunksize=$filesize
 n=1
fi

i=0
set --
#start the ftp processes
while [ $i -lt $n ]
do
 if [ ! -e \$\{file\}.\$i ]
 then
   dd if=/dev/zero of=${file}.$i bs=$chunksize count=$i \
      > /dev/null 2> /dev/null
 fi
 rm -f $file 2> /dev/null

 {
  echo user "$ftpuser" \’"$ftppass"\’
  echo bin
  echo reget $filename ${file}.$i
  echo quit
 } | ftp -nv \$host > /dev/null &

 # a list of process ids for the ftp processes
 set $* $!
 i=$(($i+1))
done

i=0
for j in $*
do
 csize=0
 k=$(($i+1))
 k=$(($k*$chunksize))
 while [ "$csize" -le "$k" ] \&\& [ "$csize" -lt $filesize ]
 do
  set -- $(ls -l ${file}.$i)
  csize=$5
  printf "$csize / $filesize of $file               \r"
  sleep 1
 done
 kill $j > /dev/null 2> /dev/null
 dd if=${file}.$i bs=$chunksize skip=$i count=1 >> $file 2> /dev/null
 rmfiles="$rmfiles ${file}.$i"
 i=$(($i+1))
done
echo
#only do this at the end so that an interrupted
# download can continue
rm $rmfiles
```

We give a shell script which, given directory names as arguments, displays a tree of the directory structure (similar to the DOS `tree` command). It is an option to include normal files in the tree.

```
#!/bin/sh

topmost=YES
normalfiles=NO

dirlist()
{
 local tabbing="$1"
 local oldtopmost="$topmost"
 shift
 for i in "$@"
 do
  if [ -e $i ]
  then
   if [ -d $i ] || [ "$normalfiles" = YES ]
   then
    if [ "$topmost" = YES ]
     then printf "${tabbing\}\-\-%s\\n" $i
     else printf "${tabbing\}\-\-%s\\n" $(basename $i)
    fi
   fi
  fi
  topmost=NO
  if [ -d $i ]
   then dirlist "$tabbing  |"  ${i}/*
  fi
 done
 topmost="$oldtopmost"
}

if [ $# -gt 0 ] && [ "$1" = '-n' ]
then
 normalfiles=YES
 shift
fi

if [ $# -gt 0 ]
then dirlist '' "$@"
else dirlist '' .
fi
```

Chapter 6

Perl

6.1 Introduction

Perl is a *Pratical Extraction and Report Language* freely available for Unix, Linux, MVS, VMS, MS/DOS, Macintosh, OS/2, Amiga, and other operating systems. Perl has powerful text-manipulation functions, i.e. string manipulations. Perl has enjoyed recent popularity for programming World Wide Web electronic forms and generally as glue and gateway between systems, databases and users. In Perl we can do all the fundamental programming constructs such as variables, arrays, loops, subroutines and input and output. We can program in Perl for data manipulation, file maintenance, packaging or interfacing system facilities, and for Common Gateway Interface Web applications (CGI programming).

For example, if we want to change the same text in many files we can use Perl. Instead of editing each possible file or constructing some cryptic `find`, `awk`, or `sed` comands from shell programming, we could simple give the command

```
perl -e 's/gopher/World Wide Web/gi' -p -i.bak *.html
```

This command, issued at the Linux prompt, executes the short Perl program specified in single quotes. This program consists of one perl operation; it substitutes for the original word `gopher` the string `World Wide Web`. This is done globally and ignoring case, indicated by `/gi`. The remainder of the Linux command indicates that the perl program should run for each file ending in `.html` in the current directory. If any file, for example `myfile.html` needs changing, a backup of the original is made as file `myfile.html.bak`.

6.2 My First Perl Program

We can run Perl from the command line. For example

```
> perl <Enter>
$a = 5;
$b = 4;
$c = $a + $b;
print($c);
```

Under Linux we press `Ctrl-d` to run the program. Under Windows we press `Ctrl-z` to run the program.

We can also write a program, for example `myfirst.pl`.

```
# myfirst.pl
$a = 5;
$b = 4;
$c = $a + $b;
print($c);
```

To run the file `myfirst.pl` we enter at the command line

```
perl myfirst.pl
```

To use Perl on a Linux and Unix system we first have to know where Perl is installed. Often it is installed in `/usr/bin/perl` or `/usr/local/bin/perl`. Thus at the beginning of the program we add the line

```
#!/usr/bin/perl -w
```

for the one Perl is localized. Under Windows we put the Perl binary in our path. For example we include

```
set path=C:\perl5\bin
```

in the `autoexec.bat` file.

After we have entered and saved the program we must make the file executable by using the command

```
chmod u+x filename
```

at the Linux prompt.

6.3 Data Types

Perl has three data types, namely scalars, arrays (also called lists) and associative arrays (also called hashes). Scalars can be numeric or characters or strings as determined by context. Scalars start with `$`. Arrays of scalars are sequentially arranged scalars. Arrays start with a `@`. Associative arrays (also called hashes) contains keys and associated values. Associative arrays start with an `%`.

```
# perl1.pl

# scalars
$a = 7;
$b = 1.23;
$c = 'A';
$d = "willi";
$e = false;

print($a);
print("\n");  # newline
print($b);
print("\n");
print($c);
print("\n");
print($d);
print("\n");
print("$e\n");

# indexed arrays
@arr1 = (3, 4, 5, 2, -1, 'a', 1.23, "egoli");
print(@arr1);
print("\n");
print($arr1[2]);  # accessing element no 2 in array => 5
print("\n");

# associative arrays
%id_numbers = ("Willi", 2, "Carl", 14, "Otto", 25);
print($id_numbers{"Otto"});  # returns 25
print("\n");
print($id_numbers{"Willi"}); # returns 2
```

The keys are `Willi`, `Carl`, `Otto` and the associated values are `2`, `14`, `25`.

Perl also allows to use of hex numbers, for example `$x = 0xffa3`.

The most commonly used global scalar variable is the `$_`. For example

```
# global.pl

$_ = 4;
$a = 7;
$b = $_ + $a;
print("b = " + $b);  # => 11
print("\n");

$_ = "willi";
print($_);         # => willi

# in the following code block
# we replace the substring the by the string xxx
$_ = " the cat is in the house";
$search = "the";
s/$search/xxx/g;    # acts on $_
print("$_\n");      # => xxx cat is in xxx house
```

6.4 Arithmetic Operation

The arithmetic operations are

```
+, -, *, /, %, ++, --, **
```

Division is floating point division.

```
# perl2.pl

$a = 4;
$b = 13;
$c = $a + $b;
print("The sum is: $c\n");

$d = $a - $b;
print("The difference is: $d\n");

$e = $a*$b;
print("The product is: $e\n");

$f = $b/$a;
print("The division yields: $f\n"); # returns 3.25

$g = $b%$a; # remainder
print("The remainder is: $g\n"); # returns 1

$a++; # post-increment
++$a; # pre-increment;
print("$a\n"); # returns 6

$b--; # post-decrement
--$b; # pre-decrement
print("$b\n"); # returns 11

# exponentiation operator is **
$x = 2.5;
$y = 3.6;
$z = $x ** $y;
print("z = $z\n");
```

6.5 Standard Input

The Perl program is referencing the file handle `STDIN` which represents the standard input file. The `<` and `>` on either side of `STDIN` tell the Perl interpreter to read a line of input from the standard input file.

The library function `chop` gets rid of the closing newline character `'\n'` that is part of the input line we entered.

```
# perl3.pl

print("Enter the distance in kilometers: ");
$dist = <STDIN>;
chop($dist);
$miles = $dist * 0.6214;
print($dist, " kilometers = ", $miles, " miles\n");

print("Enter the distance in miles: ");
$dist = <STDIN>;
chop($dist);
$kilometers = $dist * 1.609;
print($dist, " miles = ", $kilometers, " kilometers\n");
```

6.6 Basic String Operations

To concatenate two strings we can use the . operator. To compare two strings we use the `cmp` operator. The operator `cmp` is case-sensitive. If the strings are identical the function `cmp` returns 0.

```
$a = "Egoli-";
$b = "Gauteng";
print "$a$b";     # returns Egoli-Gauteng
print "\n";

# concatenate $c and $d
# the concatenate operators is the .
$c = "alpha-";
$d = "3";
$e = $c . $d;
print $e;       # returns alpha-3
print "\n";

# interpolation
$str1 = "apples";
$str2 = "pears";
print "$str1 and $str2";  # returns apples and pears
print "\n";

# comparing strings with cmp
$x = "egoli";
$y = "gauteng";
$z = $x cmp $y;
print($z);
print("\n");

$name1 = "willi";
$name2 = "willi";
$result = $name1 cmp $name2;
print($result);
print("\n");

$u = "X";
$v = "x";
$w = $u cmp $v;
print("w = ");
print($w);
```

6.7 Conditions

The comparison operators in Perl are

Operation	Numeric Version	String Version
less than	<	lt
less than or equal to	<=	le
greater than	>	gt
greater than or equal to	>=	ge
equal to	==	eq
not equal to	!=	ne
compare	<=>	cmp

The `if` statement is the simplest conditional statement used in Perl. It executes when a specified condition is true. The statement `if-else` chooses between two alternatives. The statement `if-elsif-else` chooses among more than two alternatives. The statements `while` and `until` repeat a group of statements of specified number of times. Perl also has other conditional statements we discuss later.

```
# testing.pl

$a = 1;
$b = 7;

if($a == $b)
{
print "numbers are equal\n";
}
else
{
print "numbers are not equal\n";
}
#
$c = 7.3;
$d = 7.3;
if($c == $d)
{
print "numbers are equal\n";
}
else
{
print "numbers are not equal\n";
}
#
```

```
$char1 = 'a';
$char2 = 'A';
if($char1 eq $char2)
{
print "characters are equal"
}
else
{
print "characters are not equal";
}
print "\n";
#
$string1 = "willi";
$string2 = "willI";
if($string1 ne $string2)
{
print "string1 is not the same as string2\n";
}
else
{
print "strings 1 and string2 are the same\n";
}
```

6.8 Logical Operators

The logical operators are

```
&&  logical AND
||  logical OR
xor logical XOR
!   logical NOT
```

`xor` is the logical xor operator: true if either a or b is nonzero, but not both.

```
# logical.pl

$x = 8;
if(($x > 0) && (($x%2) == 0))
{
print "integer number is positive and even";
}
else
{
print "number is either not positive or not even";
}
print "\n";

$y = 0;
if(!($y))
{
print "number is zero";
}
else
{
print "number is nonzero";
}
print("\n");

$u = 6;  $v = 5;
if(($u > 0) xor ($v > 0))
{
print("abba");
}
else
{
print("baab");
}
# what happens if you change to:
# $v = -5; ?
```

6.9 Loops

Perl provides a `for` loop, a `while` loop, a `until` loop, and a `foreach` loop. The next program shows how the `for` loop is used.

```
# forloop.pl
#
$sum = 0;
for($i=0; $i<10; ++$i)
{
   $sum = $sum + $i;
}
print "$sum";
print "\n";

$product = 1;
for($i=1; $i<12; ++$i)
{
    $product *= $i;
}
print "$product";
print "\n";
```

The following program shows how the standard input can be used within a `for` loop. The program reads four input lines and writes three of them to the screen.

```
# forloop1.pl
#
for($line = <STDIN>, $count = 1; $count <= 3;
    $line = <STDIN>, $count++)
{
   print($line);
}
```

The next program shows the use of the `while` loop.

```
# while.pl
#
# $x ne $y  Is $x string unequal to $y ?
#
print "Password? ";
$a = <STDIN>;
chop $a;                # remove the newline at end

while($a ne "willi")
{
   print "sorry. try again\n";
   print "Password? ";
   $a = <STDIN>;
   chop $a;
}

print "you did it";
print("\n");
```

The `until` statement is similar to the `while` statement, but it works in a different way. The `while` statement loops while its conditional expression is true. The `until` statement loops until its conditional expression is true. This means it loops as long as its conditional expression is false.

```
# until.pl

print("What is 27 plus 26?\n");
$correct_answer = 53;
$input_answer = <STDIN>;
chop($input_answer);

until($input_answer == $correct_answer)
{
   print("wrong! keep trying!\n");
   $input_answer = <STDIN>;
   chop($input_answer);
}
print("You go it!\n");
```

In the following example the `foreach` statement assigns a word from `@worldlist` to the local variable `$word`. The first time the loop is executed, the value stored in `$word` is the string `Good`. The second time the loop is executed, the value stored in `$word` is `Morning`. The loop defined by the foreach statement terminates after all of the words in the list have been assigned to `$word`. The local variable `$word` is only defined for the duration of the `foreach` statement.

```
# foreach.pl
#
@wordlist = ("Good", "Morning", "Egoli");
foreach $word (@wordlist)
{
   print("$word\n");
}

@numbers = ("3", "7", "3", "12", "15", "3", "3");
$count = 0;

foreach $x (@numbers)
{
if($x == "3")
{
++$count;
}
}

print $count;   # => 4
print "\n";
```

6.10 goto Statement

Like C and C++ Perl has a `goto` statement. In the following program `L1` and `L2` are labels.

```
# goto.pl

print("What is 20 + 36?\n");
$correct_answer = 56;
$input_answer = <STDIN>;
chop($input_answer);
$count = 0;

until($input_answer == $correct_answer)
{
   print("wrong! keep trying!\n");
   $input_answer = <STDIN>;
   chop($input_answer);

   if($count == 5)
   {
   goto L1;
   }
   $count++;
}

print("You got it!\n");
goto L2;

L1: print("Let's give up\n");

L2:

print("end of session");
```

6.11 Mathematical Functions

Perl has a large number of built-in mathematical functions. Perl includes all standard mathematical functions For example

```
# math.pl

$x = 3.14159;
$y = sin($x);
print("y = $y\n");

$z = cos($y);
print("z = $z\n");

# atan2 function calculates and returns
# the argument of one value divided by another,
# in the range -pi to pi
$u = atan2(1.0,4.0);
print("u = $u\n");

$v = sqrt(6.5);
print("v = $v\n");

$w = exp($v);
print("w = $w\n");

$r = log(4.0);   # log base e
print("r = $r\n");

$s = abs(-3.5);
print("s = $s\n");
```

6.12 Bitwise Operations

Perl allows the bitwise operation AND `&`, OR `|` (inclusive OR), XOR `^` (exclusive OR), and the NOT operator (one's complement) `~`. Any integer can be written in binary representation. For example

$$9 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Thus the bitstring is `1001`. If we assume we store the bitstring in 32 bits we have

```
00000000 00000000 00000000 00001001
```

The *one's complement* of this bitstring is

```
11111111 11111111 11111111 11110110
```

The bitwise operations are

```
& is the bitwise AND
| is the bitwise inclusive OR
^ is the bitwise exclusive OR
~ is the NOT operator (one's complement)
```

```
# bitwise.pl

$a = 9;
$b = 7;
$c = $a & $b;
print(" $c\n");   # => 1

$d = $a | $b;
print(" $d\n");   # => 15

$e = $a ^ $b;
print(" $e\n");   # => 14

$f = ~$a;
print(" $f\n");   # => 4294967286

# two complement
$z = -45;
$z = ~$z;
$z++;
print(" $z\n");
```

6.13 List Operations

The next four programs show how to use list operations. The operator `[ ]` is used to access the elements in the list. The first element in the list has the index 0. Thus in the following list

```
@myarray = (7, "Egoli", 1.23, "\"Having fun?\"", 9.23e-10);
```

we have

```
myarray[0] = 7; myarray[1] = "Egoli";
```

and so on. In the first program we count the number of elements in the list.

```
# mylist.pl
#

@myarray = (7, "Egoli", 1.23, "\"Having fun?\"", 9.23e-10);
$count = 0;

while($count <= 5)
{
   print("element $count is $myarray[$count]\n");
   $count++;
}
```

```
# array operations
# array.pl
#
@fruit = ("apples ", "pears ", "banana ");
@cities = ("Egoli ", "Cape Town ", "Durban ", "Pretoria ");
@numbers = ("4", "7", "9", "12", "15");

print @fruit;
print "\n";

print $cities[2]; # => Durban
print "\n";

if($numbers[2] != $numbers[4])
{
$sum = $numbers[2] + $numbers[4];  # => 24
print($sum);
print("\n");
}
else
{
$difference = $numbers[2] - $numbers[4];
print $difference;
print "\n";
}

($x, $y, $z) = @cities;
print $x;                 # => Egoli
print "\n";
print $z;                 # => Durban
print "\n";
```

In the following program we show how the `push` and `pop` operators can be used.

The operator `push` adds an element to the end of an array.

The operator `pop` removes an element from the right of an array and assign to variable, i.e. to remove the last item from a list and return it we use the `pop` function.

```
@fruit = ("apples ", "pears ", "banana ");
@cities = ("Egoli ", "Cape Town ", "Durban ", "Pretoria ");
@morecities = ("George", "Bloemfontain");

push(@cities, "Grahamstown");
print @cities;
print "\n";

push(@fruit, "strawberries ", "peach ", "plums ");
print @fruit;
print "\n";
push(@cities, @morecities);

$mycity = pop(@cities);
print $mycity;
print "\n";
```

The `pop` operator is used as follows.

```
@s = ("AAAA","BBB");
$tmp = pop(@s);
print $tmp;
```

The next program show the use of the `split` and `join` function.

To split a character string into separate items we call the function `split`. The `split` function puts the string into an array.

To create a single string from a list we use the function `join`.

```
# mysplit.pl
# splitting a string into a list
#

$string = "otto, carl, ludwig, berlin:postdam";
@myarray = split(/,/, $string);

print "@myarray\n"; # => otto carl ludwig berlin:postdam
print("\n");

print $myarray[2];   # => ludwig
print("\n");

@names = ("Egoli-", "Gauteng ", "Johannesburg");
$concatstring = join("", @names);

print("$concatstring");
```

6.14 Associative Arrays Operations

Perl defines associate arrays which enable us to access array variables using any scalar value we like. Associative arrays are also called hashes.

```
# assoc.pl

%ages = ( "Ludwig Otto", 40,  "Franz Karl", 32,
          "Jane Coppersmith", 21, "Dan Smithlone", 67);

print $ages{"Ludwig Otto"};  # => 40
print("\n");
print $ages{"Jane Coppersmith"}; # => 21
print("\n");

# adding an element
$ages{"Henry VIII"} = 1654;  # !! curly brackets

print %ages;

print("\n");

# deleting an element
delete($ages{"Franz Karl"});

print %ages;
print("\n");

@myarray = keys(%ages);
print @myarray;          # => only the names

print("\n");

@valuearray = values(%ages);
print("@valuearray");       # => only the values

print("\n\n");

while(($name, $value) = each(%ages))
{
   print($name);
   print(" ");
   print($value);
   print("\n");
}
```

6.15 Pattern Matching

Matching involves use of patterns called regular expressions. The

```
=~
```

operator performs pattern matching and substitution. This operator binds the pattern to the string for testing whether a pattern (substring) is matched. The following shows how this is applied.

```
# matching.pl

print("Ask me a question politely:\n");
$question = <STDIN>;

if($question =~ /please/)
{
   print("Thank you for being polite!");
}
else
{
   print("That was not very polite!\n");
}
```

6.16 Regular Expressions

Regular expressions are patterns used to match character combinations in strings. For example, to search for all occurences of `the` in a string, we create a pattern consisting of `the` to search for its match in a string. Regular expressions also exist in JavaScript. The syntax for the substitution operator is

```
s/pattern/replacement/

options for the substitution operator

g change all occurence of the pattern
i ignore case in pattern
e evaluate replacement string as expression
m treat string to be matched as multiple lines
o evaluate only once
s treat string to be matched as single line
x ignore white space in pattern
```

In our first program we replace the substring `123` by `4567`.

```
# replace1.pl

$string = "abc123def";
$string =~ s/123/4567/;

print("$string\n");      # => abc4567def
```

In our second example we use the global variable `$_`.

```
# regexp1.pl

$_ = "Egoli Gauteng";
s/([A-Z])/+\1+/g;     # acts on global scalar variable $_
print("$_");          # => +E+goli +G+auteng
print("\n");

s/([a-z])/\1+/g;
print("$_");          # => +E+g+o+l+i+ +G+a+u+t+e+n+g+
print("\n");
```

The next program is a white-space clean up program.

```
# replace2.pl

@input = <STDIN>;
$count = 0;
while($input[$count] ne "")
{
  $input[$count] =~ s/^[ \t]+//;
  $input[$count] =~ s/[ \t]+\n$/\n/;
  $input[$count] =~ s/[ \t]+/ /g;
  $count++;
}
print("Formatted text:\n");
print(@input);

# the line $input[$count] =~ s/^[ \t]+//;
# checks whether there are any spaces or tabs at the
# beginning of the line. If any exist, they are removed.
#
# the line $input[$count] =~ s/[ \t]+\n$/\n/;
# checks whether there are any spaces or tabs at the end
# of the line (before the trailing newline character).
# If any exist, they are removed.
#
# the line $input[$count] =~ s/[ \t]+/ /g uses a
# global substitution to remove extra spaces and
# tabs between words.
```

Here are some special regular epression characters and their meaning

```
.  any single character except a newline
^  the beginning of the line or string
$  the end of the line or string
*  zero or more of the last character
+  one or more of the last character
?  zero or one of the last character
```

The syntax for the trace function (translation) is:

```
tr/sourcelist/replacelist/
```

where `sourcelist` is the list of characters to replace, and `replacelist` is the list of characters to replaced it with.

In the second part of the following program the search pattern and the replacement pattern are the same. Every character is being translated, the number of characters translated is equivalent to the length of the sentence. This length is assigned to the scalar variable `length`.

```
# trace.pl

$string = " the moon is rising ";
$string =~ tr/oi/uo/;

print("$string");     # the muun os rosong

$word = "egoli-Gauteng";
$word =~ tr/a-z/A-Z/;

print("$word\n");       # EGOLI-GAUTENG

$sentence = "here is a sentence";
$_ = $sentence;
$length = tr/a-zA-Z /a-zA-Z /;
print("the sentence is $length characters long\n");
```

6.17 Map Operator

The `map` function enables us to use each of the elements of a list, in turn, as an operand in expression. The syntax is

```
resultlist = map(expr,list);
```

The following program gives two examples. In the first example we add 3 to all elements in the list. In the second example we concatenate `xxx` to all elements in the list.

```
# mymap.pl

@numberlist = (100, 239, 400, -1, "234");
@result = map($_+3, @numberlist);
print("@result\n");           # 103 242 403 2 237
print("$result[2]\n");        # 403

@charlist = ( "aba", "cel" );
@charres = map($_ . xxx, @charlist);
print("@charres\n");          # abaxxx celxxx
```

We can also chain the `map` operator together. The following example shows this. First all elements in the array are multiplied by 3 and then we add 2 to each element.

```
# mapchain.pl

@numbers = (1, 2, 3);
@result = map($_+2,map($_*3,@numbers));
print @result;   # 5811
```

6.18 Two-Dimensional Arrays

The operator `->` is used to de-reference the reference that has been created to point to the array of arrays.

```
# twodim.pl

$myarray = [ [ 4, 5 ], [ 6, -9]];
print($myarray -> [1][1]);       # => -9
print("\n");
$a = $myarray -> [0][1];
print($a);                       # => 5
print("\n");

$stringarray = [[ "yuli", "uilopi"], ["hulopi", "fulipok"]];
$s1 = $stringarray -> [0][1];
$s2 = $stringarray -> [1][0];

$s3 = $s1 . $s2;
print("$s3\n");     # => uilopihulopi

# addition of two 2 x 2 matrices
$a1 = [[2,3],[4,-10]];
$a2 = [[6,-9],[2,19]];
$a3 = [[0,0],[0,0]];

for($i=0; $i < 2; $i++)
{
for($j=0; $j < 2; $j++)
{
$a3 -> [$i][$j] = $a1 -> [$i][$j] + $a2 -> [$i][$j];
}
}

# display the result of the matrix addition
for($i=0; $i < 2; $i++)
{
for($j=0; $j < 2; $j++)
{
print("a3[", $i, "][", $j, "]",  " = ",  ($a3 -> [$i][$j]));
print("\n");
}
}
```

6.19 Subroutines

In Perl a subroutine is a separate body of code designed to perform a particular task. The keyword `sub` tells the interpreter that this is a subroutine definition. Subroutines are referenced with an initial `&`.

The function `srand()` sets a seed for the random number generator.

```
# subrand.pl

$res1 = &myrand;
print("res1 is: $res1\n");

$res2 = &myrand;
print("res2 is: $res2\n");

$res3 = &myrand;
print("res3 is: $res3\n");

srand();

sub myrand
{
   rand(10.0);
}
```

The function `add` adds two numbers. The variable `$c` can be seen globally.

```
# add.pl

$a = 7; $b = 5; $z = 0;

sub add
{
   $c = $a + $b;
}

$z = &add;
print("c = $c\n");  # => 12
print("z = $z\n");  # => 12
```

The subroutine `inverse` in the next example finds the inverse of a scalar variable. We recall that `$_` is the default scalar variable.

```
# inverse.pl

sub inverse
{
  -$_[0];
}

$r1 = &inverse(37);
print("r1 = $r1\n");  # => -37

$_[0] = 88;

$r2 = &inverse;
print("r2 = $r2\n");  # => -88

print("default scalar variable = ");
print $_[0];  # => 88
```

The subroutine `maximum` finds the largest of three numbers.

```
# largest.pl

sub maximum
{
   if(($_[0] > $_[1]) && ($_[0] > $_[2]))
   {
   $_[0];
   }
   else
   {
   if($_[1] > $_[2])
   {
   $_[1];
   }
   else
   {
   $_[2];
   }
   }
}  # end sub maximum

$result = &maximum(2,9,7);
print("largest number = $result\n");
```

In the following program we calculate the area of a circle.

```
# subrout1.pl

sub calc
{
   $pi = 3.14159;
   $r = 12.0;
   $calc = $pi*($r*$r);
}

$area = calc;
print $area;
print("\n");
```

The subroutine `even_odd` finds out whether a given integer number is even or odd.

```
# subrout2.pl

sub even_odd
{
   my $num = $_[0];
   print $num;
   print("=");
   return "even" if(($num % 2) eq 0);
   return "odd" if(($num % 2) eq 1);
}

$st = even_odd(7);
print $st;
print("\n");

$st = even_odd(14);
print $st;
print("\n");
```

Using the `@_` operator we can pass arrays to a subroutine. The following example demonstrates this.

```
# passarray1.pl

sub sum
{
   my $sum = 0;
   foreach $num(@_)
   {
   $sum += $num;
   }
   return $sum;
}

$total = sum(3.1,5.6,8.9,13.17);
print $total;
```

A slightly modified version is given in the next example.

```
# passarray2.pl

@mylist = ( "14", "24", "-45" );
&addlist(@mylist);

sub addlist
{
   my(@list) = @_;
   $sum =  0;
   foreach $item(@list)
   {
   $sum += $item;
   }
   print("The sum is: $sum\n");
}

@otherlist = ( 1.23, 4.5, 7.90, -34.6 );
&addlist(@otherlist);
```

The subroutine `pass` splits the list into a scalar (first item) and the rest which is again a list.

```
# passarray3.pl

sub pass
{
   my ($name, @rest) = @_;
   print $name;
   print("\n");
   print @rest;
   print("\n");
}

pass("egoli", '1', '2', '3', "Johannesburg");

pass("begin", 3.4, 4.5, 6.7, "end");
```

The output is

```
egoli
123Johannesburg
begin
3.44.56.7end
```

In the following program the subroutine `getnumbers` reads a line of input from the standard input file and then removes the leading whitespace (including the trailing newline) from the input line. We recall that `\s` stands for any whitespace. Then it is splitted into numbers. This is the return value.

```
# subsplit.pl

$sum = 0.0;
@numbers = &getnumbers;

foreach $numbers(@numbers)
{
   $sum += $numbers;
}

print("The sum is: $sum\n");

sub getnumbers
{
  $line = <STDIN>;

  $line =~ s/^\s+|\s*\n$//g;

  split(/\s+/,$line);
}
```

If variables are only used inside a subroutine and not to be seen outside, we have to make the variables local in scope. The `my` statement defines variables that exist only inside a subroutine. The following two programs demonstrate this.

```
# local1.pl

$value = 100; # global variable

sub return_value
{
  my $value;
  $value = 50;

  return $value;
}

print return_value;  # => 50

print $value;  # => 100
```

```
# local2.pl

$value = 100; # global variable

sub return_value
{
  $value = 50;

  return $value;
}

print return_value;  # => 50

print $value;  # => 50
```

6.20 Recursion

Perl also allows recursion.

In our first example we find

$$n! := 1 \times 2 \times ... \times n$$

using recursion.

```
# Recur1.pl

sub fac
{
   my $temp;
   if($_[0] == 1)
   {
   return 1;
   }
   $temp = $_[0]*&fac($_[0]-1);
   return $temp;
}  # end subroutine fac

print("enter value = ");
$n = <STDIN>;
chop($n);

$f = &fac($n);
print("Factorial of $n is $f\n");
```

As a second example we consider the Fibonacci numbers which are defined by

$$f_{n+2} = f_{n+1} + f_n, \qquad f_0 = 0,\ f_1 = 1$$

```
# Recur2.pl

sub fib
{
   my $temp;
   if($_[0] == 0)
   {
   return 0;
   }
   if($_[0] == 1)
   {
   return 1;
   }
   $temp = &fib($_[0]-1) + &fib($_[0]-2);
   return $temp;
}  # end subroutine fib

print("enter value = ");
$n = <STDIN>;
chop($n);

$fibo = &fib($n);
print("Fibonacci number of $n is $fibo\n");
```

In the following example we evaluate recursively a mathematical expression from right-to-left with the operator as prefix. For example,

```
- 98 * 4 + 12 11
```

is in standard notation

$$98 - 4 * (12 + 11)$$

which evaluates to 6. For example

```
+ 100 / -5 + 15 -16
```

is

$$100 + (-5)/(15 - 16)$$

which evaluates to 105. For our first example we get the list

```
("-", "98", "*", "4", "+", "12", "11")
```

which can be split into three parts: the first operator, `-`; the first operand, `98`; and a sublist (the rest of the list). The first three lines in the program put the input from the keyboard

```
- 98 * 4 + 12 11
```

into the list given above. Then the subroutine `rightcalc(0)` starts with the first element of the list.

```
# recur3.pl

$inputline = <STDIN>;
@list = split(/\s+/,$inputline);
$result = &rightcalc(0);
print("The result is $result\n");

sub rightcalc
{
   my ($index) = @_;
   my ($result,$op1,$op2);

   if($index+3 == @list)
   {
   $op2 = $list[$index+2];
   }
   else
   {
   $op2 = &rightcalc($index+2); # recursion
   }
   $op1 = $list[$index+1];
   if($list[$index] eq "+")
   {
   $result = $op1 + $op2;
   }
   elsif($list[$index] eq "*")
   {
   $result = $op1 * $op2;
   }
   elsif($list[$index] eq "-")
   {
   $result = $op1 - $op2;
   }
   else
   {
   $result = $op1 / $op2;
   }
}
```

A slightly simpler version with support for command line values is given below. We use `&postfix` so that PERL will use `@_` as the parameters, i.e. we consume parameters from the list `@_`. Thus `&postfix` refers to the `sub postfix` without explicitly giving parameters, also `&postfix` works on `@_` directly (it does not get its own copy).

```
#!/usr/bin/perl

#Assume all postfix expressions are valid.

sub postfix
{
#For prefix expression evaluation use
#my $node=shift;
 my $node=pop;
 my ($op1,$op2);
 if("$node" eq "+")
 { $op2=&postfix; $op1=&postfix; return $op1+$op2; }
 if("\$node" eq "-")
 { $op2=&postfix; $op1=&postfix; return $op1-$op2; }
 if("\$node" eq "*")
 { $op2=&postfix; $op1=&postfix; return $op1*$op2; }
 if("\$node" eq "/")
 { $op2=&postfix; $op1=&postfix; return $op1/$op2; }
 return $node;
}

if(@ARGV > 0)
{
 foreach $i (@ARGV)
 { print postfix(split(' ',$i)); print "\n"; }
}
else
{
 while($line=<STDIN>)
 {
  chop($line);
  print postfix(split(' ',$line));
  print "\n";
 }
}
```

6.21 The system Function

The syntax for the `system` function is `system(list)`. This function is passed a list as follows: The first element of the list contains the name of a program to execute, and the other elements are arguments to be passed to the program. When `system` is called, it starts a process that runs the program and waits until the process terminates. When the process terminates, the error code is shifted left eight bits, and the resulting value becomes `system`'s return value.

```
# system0.pl
@mylist = ("echo","Hello Egoli");
system(@mylist);

@yourlist = ("date");
system(@yourlist);
```

We can use back quotes to invoke system commands. Then the value is printed as a string.

```
# system1.pl
# using system function
# Windows

system("date");
system("dir");

$t = `time`;
print "t is now\n";
print $t;

# system2.pl
# using system function
# Linux

system("printenv PWD");
system("date");
system("pwd");
system("cd ..");
system("pwd");
system("ps");

$t = `date`;
print "t is now\n";
print $t;
```

6.22 File Manipulations

The line

```
open(MYFILE,"file1.txt")
```

opens the file `file1.txt` in read mode, which means that the file is to be made available for reading. The file `file1.txt` is assumed to be in the current working directory. The file handle `MYFILE` is associated with the `file1.txt`.

If the call to open returns a nonzero value, the conditional expression

```
open(MYFILE,"file1.txt")
```

is assumed to be true, and the code inside the if statement is executed. The next lines print the contents of `file1.txt`. The sample output shown here assumes that `file1.txt` contains the following three lines:

```
Good morning Egoli.
How are you ?
Good night Egoli!
```

```
# file1.pl

if(open(MYFILE,"file1.txt"))
  {
  $line = <MYFILE>;
  while($line ne "")
  {
  print($line);
  $line = <MYFILE>;
  }
}
```

The `open` command allows different options

```
open(INFO,$file);     open for input
open(INFO,">$file");  open for output
open(INFO,">>$file"); open for appending
open(INFO,"<$file");  also open for input
```

```
# file2.pl
#
# terminating a program using the
# die
# function

unless(open(HANDLE, "file1.txt"))
{
   die("cannot open input file file1.txt\n");
}

# if the program gets so far, the file was
# opened successfully

$line = <HANDLE>;
while($line ne "")
{
   print($line);
   $line = <HANDLE>;
}
```

```
# file3.pl
#
# writing to a file

unless(open(INFILE,"file1.txt"))
{
   die("cannot open input file file1.txt\n");
}

unless(open(OUTFILE,">file2.txt"))
{
   die("cannot open output file file2.txt\n");
}

# the > in >file2.txt indicates that the file
# is opened in write mode.

$line = <INFILE>;

while($line ne "")
{
   print OUTFILE ($line);
   $line = <INFILE>;
}

# the line
# print OUTFILE ($line);
# writes the contents of the scalar variable
# $line
# on the file specified by OUTFILE.
# file4.pl
# closing a file

$file = "willi.dat";
open(INFO, $file);
@lines = <INFO>;
close(INFO);
print @lines;
print("\n");

# file willi.dat contains
# 20 3 1960
```

Let us assume that the file named `FOO` contains the text

```
This line equals: 3
This line equals: -4
This line equals: 6
```

At the command line run:

```
perl file5.pl FOO
```

The output is:

```
This line equals: 6
This line equals: -8
This line equals: 12
```

The result is stored in the file `FOO`. Thus we multiply every integer in a file by 2. The pattern `\d+` matches a sequence of one or more digits.

```
# file5.pl

$count = 0;
while($ARGV[$count] ne "")
{
   open(FILE, "$ARGV[$count]");
   @file = <FILE>;
   $linenum = 0;
   while($file[$linenum] ne "")
   {
   $file[$linenum] =~ s/\d+/$& * 2/eg;
   $linenum++;
   }
   close(FILE);
   open(FILE, ">$ARGV[$count]");
   print FILE (@file);
   close(FILE);
   $count++;
}
```

In the following program we count how often the name `Smith` (case-sensitive) is in the file `smith.txt`.

```
# findSmith.pl

unless(open(INFILE,"smith.txt"))
{
   die("cannot open smith.txt");
}

$line = <INFILE>;
$nbSmith = 0;
$single = 0;

while($line ne "")
{
   print("$line ");
   $single = &countName($line);
   print("$single \n");
   $nbSmith += $single;
   $line = <INFILE>;
}

print("Number of Smith in file smith.txt = $nbSmith \n");
close(INFILE);

sub countName
{
   $i = 0;
   @narray = split(/ /,$_[0]);

   foreach $x (@narray)
   {
   if($x =~ /Smith/)
   {
   $i++;
   }
   }
   return $i;
}
```

In this program we made extensive use of `man perlop perlfunc perlipc`. The program draws a tree structure of a given directory.

```
#!/usr/bin/perl

$normalfiles=NO;

sub dirlist
{
 my ($tabbing,$curdir,@dir)=@_;
 foreach $i (@dir)
 {
  if ( -e "$curdir$i" )
  {
   if (( -d "$curdir$i" ) || ( "$normalfiles" eq YES ))
   {
    print "$tabbing\-\-$i\n";
   }
  }
  if ( -d "$curdir$i" )
  {
   opendir(DIRHANDLE,"$curdir$i")
    or print("Failed to open directory $curdir$i.\n");
   my @newdir=readdir(DIRHANDLE);
   shift @newdir;
   shift @newdir;
   closedir(DIRHANDLE);
   dirlist("$tabbing  |","$curdir$i/",@newdir);
  }
 }
}

if (( @ARGV > 0 ) \&\& ( "\$ARGV[0]" eq "-n" ))
{
 $normalfiles=YES;
 shift @ARGV;
}

if ( @ARGV > 0 )
{ dirlist("","",@ARGV); }
else
{ dirlist("","","./"); }
}
```

6.23 Networking

```
# net1.pl
# networking
# the gethostbyaddr function
# searches the hostname for a
# given network address.

print("enter an Internet address:\n");
$machine = <STDIN>;

# these three lines convert the address into
# a four-byte packed integer
$machine =~ s/^\s+|\s+$//g;
@bytes = split(/\./, $machine);
$packaddr = pack("C4", @bytes);


if(!(($name, $altnames, $addrtype, $len, @addrlist)
      = gethostbyaddr($packaddr,2)))
{
   die ("Address $machine not found.\n");
}

print("principal name: $name\n");

if($altnames ne "")
{
   print("alternative names:\n");
   @altlist = split(/\s+/, $altnames);
   for($i=0; $i < @altlist; $i++)
   {
   print("\t$altlist[$i]\n");
   }
}
# if we enter:
# 152.106.50.60
# we get
# principal name: whs.rau.ac.za
# net2.pl
# networking
# the gethostbyname function
# searches for an entry that matches
# a specified machine name or Internet site name.

print("enter a machine name or Internet sie name:\n");
$machine = <STDIN>;

# machine name prepared
$machine =~ s/^\s+|\s+$//g;
print $machine;
print("\n");

if(!(($name, $altnames, $addrtype, $len, @addrlist)
      = gethostbyname ($machine)))
{
   die ("Machine name $machine not found.\n");
}

print("equivalent addresses:\n");

for($i=0; $i < @addrlist; $i++)
{
   @addrbytes = unpack("C4", $addrlist[$i]);
   $realaddr = join (".", @addrbytes);
   print("\t$realaddr\n");
}
# if we enter:
# whs.rau.ac.za
# we get
# 152.106.50.60
```

In the following program we create a minimal telnet client.

```
#!/usr/bin/perl

use IO::Handle;
use Socket;

$server=shift||"localhost";
$port=shift||"telnet";
$proto=getprotobyname("tcp")||die("getprotobyname failed.\n");
$port=getservbyname($port,"tcp")||die("getservbyname failed.\n");
$addr=gethostbyname($server)||die("gethostbyname failed.\n");
$addr=sockaddr_in($port,$addr)||die("sockaddr_in failed.\n");

socket(SOCKET,PF_INET,SOCK_STREAM,$proto) ||
 die("Failed to open socket.\n");
connect(SOCKET,$addr) ||
 die("Failed to connect.\n");

print "Connection established.\n";

STDIN->autoflush(1);
SOCKET->autoflush(1);

if($childpid=fork())
{
 while( $toprint=<SOCKET> ) { print STDOUT "$toprint"; }
 kill(TERM,$childpid);
}
else
{ while(($tosend=<STDIN>) && (print SOCKET "$tosend")) {} }

close(SOCKET);
```

Bibliography

[1] Horton Ivor, *Beginning Java 2*, WROX Press, Birmingham, UK, 1999

[2] Jaworski Jamie, *Java 1.2 Unleashed*, SAMS NET, Macmillan Computer Publishing, USA, 1998

[3] Johnson Marc, *JavaScript Manual of Style*, Macmillan Computer Publishing, USA, 1996

[4] McComb Gordon, *JavaScript Sourcebook*, Wiley Computer Publishing, New York, 1996

[5] Tan Kiat Shi, Willi-Hans Steeb and Yorick Hardy, *SymbolicC++: An Introduction to Computer Algebra Using Object-Oriented Programming*, 2nd edition Springer-Verlag, London, 2000

Index