Systems Programming / Programming & Environments.

This document describes a few small command line utilities. The design & coding of some of these utilities will be your tutorial work for the next few weeks. An attached note details which of these are relevant to your current module. The utilities will carry out the following tasks:

1. list text files
2. compress/uncompress program source files using run length encoding;
3. compress/uncompress program source files using tokenisation;
4. encrypt/decrypt text files;
5. display a summary of waiting mail messages;
6. search a directory tree for large files;
7. simulate the UNIX command shell.

In size/difficulty, each of these program is like an extended tutorial exercise. Each one is different in terms of the logic necessary for its design and concentrates on different features of C.

You will not have covered all of the most useful features of C when you first receive this document. You should start by using your current knowledge to build prototypes for each program. You should not try to fully complete one problem before going on to the next. It is better to do the easier parts of each one before trying the harder parts of any of them. As lectures progress (& you read through some text books on C) you can adapt & extend your early prototypes into complete programs.

You may choose which order you tackle the three programs. In the description of each problem is an outline of which parts to do first.

It is likely, as you learn more about C, that you will want to re-code some functions. This re-coding is (not a waste of time but..) a useful exercise, you will need to think about the advantages of different styles of coding & features of C. You will also do your own review of your older code.

The purpose of this work is as a learning exercise. It is intended to increase your skills in C and also in problem solving & design. You are encouraged to discuss these problems with other students and with your tutors.

The programs you produce will not be directly assessed but your module exam will have some questions based around some parts of these exercises.

Text file listing utility

This utility displays text files to the screen, allowing users to select different options like switching on/off line numbering. Have a look at the UNIX command called **more** and the manual entry for **more** to give you an idea of what you are trying to achieve.

The program should accept a number of different command switches (summarised below) and the name of a text file (which is the name of the file to be listed).
eg: `filelist  -n  -l25  report1.txt`

Files should normally be displayed a screen at a time (like **more**) but (unlike **more**) the next screen should be displayed when the user presses RETURN. More allows users to control the way their files are displayed by pressing different keys between screen displays. In order to respond directly to a single keypress (other than RETURN) the terminal must be operating in raw, unbuffered mode. A user program can make system calls to request that the terminal input is handled in this way but this is not trivial. If you want to enhance your program so it sets its input device to work in a raw mode a good source of reference is the O'Reilly book – UNIX Systems Programming. Before attempting this you should ensure that your file lister works in all other respects.

Summary of switches:

`-t`<u>*ch*</u>	define the character to be recognised as tab. This should defaults to '\t'. eg: `–t$` use $ as the tab character.
`-T`<u>*num*</u>	Set the tab stop to num spaces. eg: `-T6`
`-n`	Display line numbers
`-l`<u>*num*</u>	Display <u>*num*</u> lines in each screen-ful, rather than the default (you should provide a sensible default). eg: `-l25`
`-f`<u>*num*</u>	Start up at line number <u>*num*</u>. eg: `-f150`
`-c`	Clear screen before displaying.

Dealing with so many switches with a specific piece of code to check for the existence of each can get messy. A better solution is to write a general purpose function which deals with switches. You may be able to find similar functions in libraries of utilities provided for free on the internet or on CDs provided with text books but we encourage you to write your own version as part of this exercise.

Something suitable could be a function called switchvalue with the following prototype:

```
int switchvalue( int argc, char *argv[], char schar );
```

switchvalue checks for the occurrence of an optional switch on the command line, preceded by a '-' symbol and equal to its schar argument. switchvalue returns the following values:

- 0 if the switch is not present,
- 1 if the switch is present then followed by a space on the command line,
- an integer value if the switch is followed by numeric digits (the value returned corresponds to the value of the of the digits).
- an ascii code returned as an integer value if the switch is followed by non-numeric character (you need this for the -t switch).

For example, using the command line: `a.out  -n  -f37 –t$  /data/infile`

```
switchvalue( argc, argv, 'n' ) returns 1
switchvalue( argc, argv, 'f' ) returns 37
switchvalue( argc, argv, 't' ) returns '$'
switchvalue( argc, argv, 'x' ) returns 0
```

Steps

1. Write a simple file listing utility which provides line numbers (remember that a user may wish to have these line numbers switched off). Assume the file is provided by redirection on stdin.
2. Using something like a dollar symbol as a tab character, work out how to handle tabs (NB: though sensible, you are not supposed to handle tabs by writing '\t'. You are expected to work out how many spaces should be displayed then display them).
3. Write a function to handle command line switches & amend your program so it accepts the text file name as a command line argument.
4. Extend your code to deal with the command line switches you are interested in. You will have to adapt your code to deal with each new switch & what it implies. Some are easier to implement than others, be organised or your code will get messy!
5. Complete the other exercises before dealing with this!! Investigate the ways you can change the characteristics of your terminal/window so you can allow it function in raw mode & allow different keypresses to cause different actions between screens of text. (note: the version of UNIX running in the School is BSD compliant).

Text File Listing - Task Overview

✓ Build a prototype which deals with line numbers and tabs.

✓ Write a function to handle command line switches.

✓ Amend your program so it accepts the text file name as a command line argument & extend your code to accept some command line switches. Accept at least one switch which is followed by a number.

? Deal with the remaining switches.

? Play around with the terminal settings so you can use it raw mode.

File compression & decompression.

This utility program is to compress & decompress program source files so they occupy less disk space. The techniques described here for file compression can be adapted for various file types but for the sake of program design & testing it is easier to assume you are working with text files of a typical content.

You are aiming to achieve two types of compression:
1. Run length encoding,
2. Tokenisation

This specification assumes that the files to be compressed will be program source files. You can choose whether these are Pascal source files or C source files.

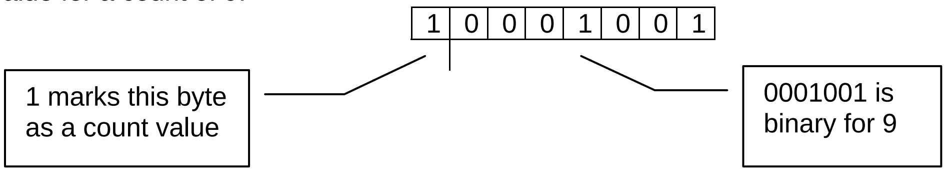
Run-length encoding

The idea behind this method is to replace multiple occurrences of the same letter by one occurrence of that letter & a count. EG: "`FISSSSSSHINGGGG`" would be compressed as: "`FI`**`6S`**`HIN`**`4G`**". This is a fairly simple compression technique that works well with program source files because of all the spaces used to indent.

As an example: when your compression program reads "FFFFFFF" from an uncompressed file it should write **7F** to the compressed file. This compression code is in two parts, the **7** & the **F**. The **F** is just a normal "F" character but the **7** needs to be encoded so that it recognised as a count by the decompressor - rather than some valid alpha-numeric character. In text files characters are stored in ASCII form (if you can't remember what ASCII is check out your 1st year notes or look in a book). ASCII character codes each occupy a byte in a text file but do not use the most significant bit (which is set to zero).

The simplest way to store a count in a text file it is to store it as a byte value that is not a valid ASCII code. One way to do this is to use the bottom 7 bits to hold the count & set the top (most significant bit) to 1. Setting the top bit to 1 ensures the count can be easily recognised by the decompressor.

IE byte value for a count of 9:



When you first design this mechanism you should use a whole byte in your compressed file to store each count (as described) but you should plan to run-length encode a sequence of two or more letters (later you may want to refine your program to store counts in half-byte fields).

Steps towards run-length encoding:
1. Read about ASCII & using C's char type to hold numeric values (it is not a problem). Read about bitwise operators in C.
2. Carefully analyse the problem & design a basic run-length encode mechanism.
3. Build a small interactive prototype to read characters from the keyboard & apply run-length encoding/decoding.
4. Build your prototype into a function called from main, allow main to read input from a file. NB: let main take care of opening & closing files.
5. Extend main to use command line arguments.

Tokenisation
With this method common sequences of characters are replaced by single tokens. With mixed documents common letter sequences are things like "ing", "ble", "the" etc. You should design your utility specifically for use with source files written in C or Pascal. Common sequences tend to be the keywords of the source language "while", "char", "for" etc. Note: you'll get a better compression with Pascal because the keywords tend to be longer & "BEGIN" & "END" will be tokenised.

In this problem tokens will be ASCII character values which will never be used in a standard text file (or a program source file).

In general common character sequences can either
1. be found by scanning the file to be compressed, identifying the most common sequences & building a token mapping;
2. built in (hard-coded in a datastructure) in the compression mechanism.

You should take the second (simpler) approach.

Steps towards tokenisation:
1. Identify the character sequences you want to tokenise.
2. Check though an ascii character map to find 7-bit ascii values which will not be used in a standard text file (do not use NULL). These will be your token values. Do not use 8-bit values as this will get confusing if you want to both tokenise and run-length encode.
3. Decide which token will represent which character sequence.
4. Design a datastructure which will serve as a lookup table so that (i) given a character sequence you can find its token (for the compressor) and (ii) given a token you can find its character sequence (for the decompressor).
5. Investigate <string.h>strtok, it will help you to isolate C/Pascal keywords.
6. Build a small prototype which allows you to test your table lookup machanism.
7. Rebuild the prototype into a function called from main. Allow main to take care of all file handling & input/output.

File compression & decompression - Task Overview

- ✓ Carry out analysis & design for both methods of de/compression.

- ✓ Build & test prototypes for each method (to do both compression and decompression for each method).

- ✓ Build separate programs for each method. Programs should be complete, robust and accept suitable command line arguments.

- ? Incorporate the two methods into a single controlling program which runs both methods of compression.

- ? Adapt the run-length encoder so counts are stored in 4-bit fields. This will require some careful analysis & design.

File encryption & decryption.

This utility is to use a password to encode & decode files, allowing the owner of the file to keep its contents private from others who may have access to it.

One method for encryption uses a password as follows: the ASCII code for each character in the input file is added to the ASCII code for a corresponding character in the password. So the 1st character from the input file is combined with the 1st character in the password, the 2nd character from the input file with the 2nd character from the password etc. When the end of the password is reached the mechanism continues by starting at the beginning of the password again. The decryption program reverses this action.

EG: With an input file starting 'confidential memo...' and a password 'code', the encrypted text would be as shown below.

input file - text	c	o	n	f	i	d	e	n	t	i	a	l		m	e	m	o	...
input file - ASCII	99	111	110	102	105	100	101	110	116	105	97	108	32	109	101	109	111	

password - text	c	o	d	e	c	o	d	e	c	o	d	e	c	o	d	e	c	...
password - ASCII	99	111	100	101	99	111	100	101	99	111	100	101	99	111	100	101	99	

encrypted version	198	222	210	203	204	211	201	211	115	216	197	109	131	220	201	210	210	

The encryption mechanism should work with binary files as well as text files. To allow this you must (i) ensure that coded values are legal byte values (ie: between 0 & 255) and (ii) use a file reading function that does not confuse a value of zero with end-of-file.

When fully developed the encryption program would be called at the command line with an input file name, an output file name and a switch triggering either encryption or decryption. In addition it may also have another optional switch indicating whether the user wants their input file deleted after en/de-cryption.

The password should be entered from the keyboard after the command line aguments have been checked (ideally the characters in the password should not be echoed to the screen).

Steps towards en/de-cryption:
1. Analyse the logic for en/de-cryption & be sure you understand the relationship between characters & ASCII.
2. Build encryption & decryption functions and test them with a simple harness. The en/de-cryption functions should not do any file or keyboard/screen input-output.
3. Build a complete program to use the encryption & decryption functions. This program should accept suitable command line arguments for the names of input & output files and read the password from the keyboard.

File encryption & decryption - Task Overview

✓ Carry out analysis & design for en/de-cryption.

✓ Build & test prototypes to encrypt & decrypt.

✓ Build & test a program to act as the single driver for both encryption and decryption. This program should be complete, robust and accept suitable command line arguments.

? Adapt your program so the password is not echoed to the screen as it is typed (this is not difficult).

? Allow an optional command line switch which causes the input file to be deleted after en/de-cryption.

Mail Summary.

In its finished state this utility will be activated automatically at login to display a summary of the mail messages in a user's incoming mail file. This summary will consist of one line of screen display for each message, stating the sender's name, the subject and the message date.

These notes refer to mail on the SCM UNIX system only: A user's mail is placed in a file in the /var/spool/mail directory. Each user has a file in this directory with the same name as their user name. This file is a text file which appears as a series of mail messages. New messages are appended to the end of the file so the fist message in the file is the oldest.

Each message in the mail file has two parts: the header & the message. The header is (unfortunately) not completely regular in structure. Look at your own /var/spool/mail file. Most mail messages will have similar headers but few lines of the header are guaranteed to be present for any message. This mail utility must be flexible enough to cope with mail headers which may have some information missing (eg: the subject) and do not always present header lines in the same order.

One feature of a mail header which will always be present for all messages is the sender's name. The start of a new mail message is recognised by a line which specifies the sender's name. Although it seems unlikely this line is recognised because it starts with the letters:

```
        From□
```

where □ denotes a space character. The mailing system ensures that no other line starts "From□" by editing any lines of an incoming message which start with these characters. Send yourself an appropriate mail message to check this.

A typical mail header follows. The areas marked in bold are some of those that may be of interest to you in designing this mail summary utility.

```
From m.a.lockyer@tees.ac.uk Fri Oct 17 13:58:16 1997
Status: RO
Received: from teesside (mailer.tees) by scorch; Fri, 17 Oct 97
13:58:06 BST
Received: from scm-arkengarthdale.tees.ac.uk
(scm_arkengarthdale.tees.ac.uk) by teesside.ac.uk; Fri, 17 Oct 1997
13:55:19 +0100
Received: from SCM_ARKENGARTHDALE/SpoolDir by scm-
arkengarthdale.tees.ac.uk (Mercury 1.21);
    17 Oct 97 13:53:10 GMT0BST
Received: from SpoolDir by SCM_ARKENGARTHDALE (Mercury 1.21); 17 Oct
97 13:52:30 GMT0BST
From: "Mike Lockyer" <m.a.lockyer@tees.ac.uk>
To: "SCM academic staff" <SCMacademic@scm-arkengarthdale.tees.ac.uk>
Subject: IT in Teaching
Date: Thu, 16 Oct 1997 20:46:18 +0100
Errors-To: <C.C.Marshall@tees.ac.uk>
Sender: CM-mailbase@tees.ac.uk
X-Listname: <SCMacademic@scm-arkengarthdale.tees.ac.uk>
Mime-Version: 1.0
Content-Type: text/plain
Content-Transfer-Encoding: 7bit
X-Mailer: Microsoft Outlook Express 4.71.1712.3 (via Mercury MTS
v1.21)
Message-Id: <36059A7F04@scm-arkengarthdale.tees.ac.uk>
Content-Length: 495
X-Lines: 24
```

As already noted: the structure of mail headers vary. Lines may be omitted or occur in a different order from those in the header above.

Note: This document ignores the issue of attached files which may be dealt with slightly differently on different systems. With the SCM UNIX system attached files are embedded in a user's mail file in an encoded form which resembles text. Attached files do not complicate the structure of message headers or the mail file itself.

Steps towards the mail utility:

1. Analyse the structure of the mail file and mail headers. Send yourself a variety of different types of message using different mailers to send the messages. View your mail file before and after using (eg) mailtool to read the messages. The result of this analysis should be an unambiguous description of the file structure.
2. Read about the use of scanf, fscanf & sscanf for scanning lines of text.
3. Build a prototype to read a mail file and display only the sender's name for each message. NB: it is probably easier at this stage to use a copy of a suitable mail file rather than take input from your file in /var/spool/mail.
4. Extend the prototype to display subject & date information as well as the sender's name.
5. Investigate the ways your program can be automatically executed at login.
6. Complete the prototype by making it execute system calls to find out the name of the user who is running it - so it can examine the correct mail file. Devise a method to allow its output to be displayed one page at a time.

Mail Summary Utility - Task Overview

- ✓ Carry out analysis of the mail file and mail headers.
- ✓ Build & test a prototype to display the sender's name for each message.
- ✓ Build & test a prototype to display subject & date information as well as the sender's name.
- ? Set up your program so it is automatically executed at login.
- ? Complete the prototype as in step 6.

FileSpace search

This is a utility which searches through a directory tree structure reporting on the files & sub-directories within it. You will develop the utility to the point where it can be used as like the UNIX **ls -FR** command (which lists files within subdirectories within directories) or as a means for searching an entire user's filespace for particular types of file - for example those that are larger than a specified size. This type of utility can be extended into a general purpose filespace searcher which can report on all files of a particular type or files altered since a given date etc.

Steps

1. Investigate the use of the **stat** command to distinguish between a file & a directory and the use of **opendir** & **readdir** to read the contents of directories.
2. Write a simple version of the **ls** command. Do this by writing two functions, one called **scandir**, which opens a directory & reads its contents and another which handles the displaying of file names. Don't worry about the style of the display too much.
3. Adapt the file display mechanism so it postfixes a "/" character on the end of sub-directory names.
4. Adapt your ls command so it recurses through sub-directories. Do this by modifying your display function so it calls scandir when it is given a directory to display.
5. Use the ideas you have developed to build a program which displays those files in a directory tree which are greater than a specified size.
6. Complete the utility by using a command line switch to specify the size of files which should have their names displayed. (NB: some people have had problems using the function **atol** with one PC compiler. This is caused by a bug in the library software supplied with the compiler - I think!).
7. Think about the additional problems you would face if this utility were to be extended into a general purpose filespace search utility.

✓ Carry out investigation into the structure of directories & the use of the **stat** command.

✓ Build & test a prototype to display the names of files within a directory and put a "/" against sub-directory names.

✓ Extend the prototype to recurse through sub-directories, printing the names of all files in a directory tree.

✓ Extend the prototype again, this time so it checks the sizes of all files in the tree before printing their names.

? Add the command line switches feature.

? Adapt the search so it can report on all files which are more recent than a date given on the command line.

Command Shell

The aim of this work is to develop a simple UNIX command shell. Some of the concepts are quite advanced so, as always, take it step by step.

Steps:

1. Write a program which reads a line of text from stdin and, assuming it represents a command line, structures it into **argv** format.

2. Investigate the use of **execv** & **execvp** as a means of calling executables from a program. Note that exec'ed processes are overlaid so the calling processes no longer exists after a successful exec call.

3. Explore the use of **fork** for producing cloned processes. Use fork to overcome the problem of master processes being overlaid by exec'ed processes. Do this by forking a clone child process and use this child to make the exec call while the parent waits for it to finish then accepts another command line from the user.

4. Investigate the means of trapping interrupts like ^C and so prevent the user from terminating your shell in this way. The user should only be able to terminate the shell by typing "exit" (which should be intercepted by your shell & not exec'ed) or typing the appropriate kill command.

5. Write a shellmode command which allows the user to change the behaviour of your shell in the following ways:
    - toggle the display of ok/failed message after a command is exec'ed;
    - toggle waiting for exec'ed process before accepting a new command line;
    - change the style of prompt.

Command Shell - Task Overview

✓ Read a line of text from stdin & structure it into argv format (you will need to write a small harness to check on the results).

✓ Write the exec call to trigger the required executable with your argv structure.

✓ Code up an appropriate call to fork to overcome the overlaying problem.

? Build a ^C interrupt trap into your command shell.

? Add the shellmode feature.