# CS1110 Introduction to Systematic Programming

In this lecture I will discuss:

the UNIX file system
use of some of the basic UNIX commands for handling files and directories

## UNIX Filenames and Extensions

In UNIX a filename is (usually) of the form

```
basename.extension
```

The basename may be chosen freely by the user; normally one should give a file a meaningful name which indicates briefly its purpose or contents. Thus for a file containing a student's submission for the first ISP coursework problem the basename `ISPCwk1` would in general be preferable to (say) `file1`.

Usually UNIX filenames have a short extension (usually three or fewer characters) to indicate the nature of each file. For example the extension `.adb` (short for **Ad**a **b**ody) might be used for a file containing an Ada program, `.c` for a file containing a C program, `.o` for a object file (containing binary machine code), `.txt` for files containing text, `.dat` for a files containing data for a program, `.ali` for files **A**da **L**ibrary **I**nformation, `.ps` for a files PostScript (a page description language), etc.. In UNIX filenames without an extension are normally reserved for executable programs (or applications) and for directories (or folders).

For example whilst doing their first Ada coursework assignment students might create the files `ISPCwk1.adb` (for the Ada program itself) , `ISPCwk1.dat` (for test data), `ISPCwk1.res` (for program results), `ISPCwk1.txt` (a text file containing an account of the design of the program). The final executable program might be called `ISPCwk1`. The Ada compiler would also create files `ISPCwk1.o` and `ISPCwk1.ali`; these contain the object (binary) code and Ada library information respectively produced by the Gnat Ada compiler.

In UNIX (unlike Ada) the case of characters in file names and in UNIX commands is significant. Thus for example `ISPCwk1.adb`, `ISPcwk1.adb` and `ispcwk1.adb` are different filenames and `cp` is the UNIX file-copying command whereas `CP` is not.

Although filenames in UNIX can contain virtually any characters available on a keyboard, generally for convenience filenames in UNIX should contain only letters, numbers, underscores and hyphens. In particular spaces and punctuation characters (except full-stops to separate basenames and extensions) should be avoided in UNIX filenames. In UNIX there is virtually no limit on the length of filenames or extensions -- the complete filename with extension must be no longer than 1023 characters, but this is more than enough for all practical purposes!

## The UNIX File System & Directories

Like all other modern operating systems UNIX allows users to keep their programs and data on-line in files that are immediately accessible. As there are often thousands of files on the computer system, for ease of access related files are usually stored together in a **directory** (or **folder**). A file containing a piece of text, a program or data is called a **regular file**, whereas a file containing the names and information on the location of other files is called a directory.

Each user on a UNIX system is given a **home directory** in which store their files. The home directory of a user is denoted by `~userid` where `userid` is the user's UNIX log-in name. When users log-in to a UNIX system, their home directory becomes their **current working directory**.

A directory may contain a mixture of regular files and other (sub-)directories. Thus the internal nodes of the file system tree are directories and 'leaf' nodes are regular files. Different branches of the file system tree may reside on different physical devices (usually magnetic disks). The device may not even be connected directly to the computer, but may belong to another machine on the same network, but owing to NFS (network file service) this is largely

transparent to the user. The system administrator **mounts** physical devices containing file systems on nodes of the file system tree making them accessible to users.

The UNIX **file system** is a hierarchical tree structure with a root denoted by the character `/`. Each file or directory in the file system is identified by a character string known as its **absolute (or full) pathname**: this lists the names of all the directories on the path from the root node `/` to the file or directory required. These directories are separated by the character `/`.

For example `/usr/local/bin/emacs` specifies a file named `emacs` (the Emacs editor) which is in a sub-directory `bin` which itself is a sub-directory of the directory `local` which itself is a sub-directory of the directory `usr` which is a sub-directory of the root directory `/`.

The name of the home directory is usually the same as the user's UNIX userid. For Aston staff and students in the School of Engineering & Applied Science, the home directory is usually a 'descendant' directory of the top-level directory `/eas`. For example, on the file-server `helios` at Aston a typical user's home directory might be `/eas/d205/barnesa`.

To access a file in the current working directory one simply needs to specify the name of the file. For example if a file (`prog1.adb`, say) resides in the current working directory we could refer to it simply by its name and extension `prog1.adb`. However if the file is in another directory we need to specify both the name of the directory and the name of the file. For example if it resided in the home directory of another user (`csweb4` , say) we would need to specify the directory and the name as follows: `~csweb4/prog1.adb`. If instead the file resided in the directory

```
/usr/local/staffstore/CSAdaLib
```

we would specify it as

```
/usr/local/staffstore/CSAdaLib/prog1.adb
```

and so on.

Since absolute pathnames can be rather long, UNIX allows users to refer to a file by specifying its **relative pathname**, that is by specifying its location relative to the current working directory (cwd). **Note that absolute pathnames start with a `/` whereas relative pathnames do not**.

For example assuming the current working directory is `/eas/d205/barnesa`, a user could refer to the file `/eas/d205/barnesa/.login` by using the simple file name `.login`. Similarly the file `unit4.adb` in the sub-directory `ISP/unit-programs` of `/eas/d205/barnesa` could be referred to by the relative pathname:

```
ISP/unit-programs/unit4.adb
```

which is clearly easier than using its full pathname namely

```
/eas/d205/barnesa/ISP/unit-programs/unit4.adb
```

## Basic UNIX commands

UNIX commands may be entered at the UNIX shell prompt in an active `xterm` window.

### Listing the contents of a directory

In order to see the names of files contained in a directory the UNIX command `ls` (short for **ls**t) is used.

| | |
|---|---|
| `ls` | list the names of files in the current working directory |
| `ls` *`mydir`* | list the names of files in the directory *`mydir`* |

For example

| | |
|---|---|
| `ls /usr/local/bin` | list the names of the files in the system directory `/usr/local/bin` |
| `ls ~csweb5` | list the names of the files in the home directory of user `csweb5` |

`ls ~` — list the names of the files in your home-directory (even if it is not the current working directory). Note that ~ is UNIX shorthand for the home directory of the current user.

**Displaying the contents of a text file**
UNIX provides a command `cat` (short for con**cat**enate) for this:

`cat` *`file`* — display the contents of the specified *`file`* on standard output (usually the terminal window). The whole file is displayed 'in one go'.

To display long files it is usually more convenient to use a **pager** program which displays one screen-full (often referred to as one page) at a time. There are usually several pager programs available on UNIX systems for example `page`, `more` and `less` (`less` is probably the best of these).

`less` *`file`* — display the contents of the specified *`file`* one screen-full at a time Press the space-bar (or the `'f'` key) to display the next 'page', press the `'b'` key to move back one page or press the `'q'` key to terminate the display of the file. `'g'` and `'G'` move to the start and the end of the file respectively. For help on more advanced features of `less`, press the `'h'` key followed by `<Return>`.

The program `more` behaves in a more or less similar fashion to `less`, but the program terminates automatically whenever the end of the file is reached. Incidentally the reason (such as it is) for the bizarre names is that pager programs often display a prompt at the bottom of the page displayed of the form "More 25%" to indicate that there is more of the file than appears on the display and that the currently displayed page is about one quarter of the way through the file. `less` is a rather ironic name for a program with more features than `more`!

**An alternative way of viewing a text file is to open the file in Emacs.**

Generally one should only display **text files** with utilities such as `cat`, `less`, `more` etc.. Never attempt to display binary files such as executable programs with these utilities --if you do attempt to display binary files, certain 'control characters' in these files are likely to crash your terminal window making it unusable; then you may need to kill the `xterm` window. Actually if you really want to see what an binary file looks like, it is safe to open it with Emacs -- however it simply looks like gobble-de-gook.

Often you can guess the type of file from its extension. For example typical text files might have the extension `.txt` (plain text files), `.adb` & `.ads` (Ada program text files) `.c` (C program source files) whereas typical binary files might have the extensions `.o` (relocatable binary) `.a` (UNIX archive file) and no extension (executable binary). However in cases of doubt you should use the UNIX utility `file` to check that file-type is text before attempting to display a file. Here are a few examples of the use of the command `file` and the output it produces:

```
file /usr/local/bin/emacs
ELF 32-bit MSB executable SPARC Version 1

file /usr/local/staffstore/CSAdaLib/cs_int_io.o
ELF 32-bit MSB relocatable SPARC Version 1

file /usr/local/staffstore/CSAdaLib/cs_file_io.adb
ascii text

file /usr/local/staffstore/CSAdaLib/cs_file_io.ali
ascii text
```

In the first two examples the files are binary files (of various kinds); only the third and fourth are text-files.

**Changing the current directory**
As we have said, when a simple file name is given to a UNIX command the file is assumed to be in the current working directory. In order to specify a file in another directory we must give both the directory and the file name -- and since UNIX directory names are often rather long

this can involve a considerable amount of typing! To avoid this hassle, users may change the current working directory using the command `cd`. Note also as you change directory, the TC-shell prompt alters automatically to display the current working directory.

| | |
|---|---|
| `cd` *`directory`* | make the specified *`directory`* into the current working directory |

For example

`cd /usr/local/gnat312p/gnat-3.12p-docs`
make the system directory `/usr/local/gnat312p/gnat-3.12p-docs` into the current working directory (`/usr/local/gnat312p` contains miscellaneous files relating to the Ada system you will be using in the ISP course).

`less gnat_ug.txt` then display the contents of file `gnat_ug.txt` in that directory. This contains documentation regarding the gnat Ada system. Without the prior `cd` command one would need to specify the name of the directory **and** the file:

```
less /usr/local/gnat312p/gnat-3.12p-docs/gnat_ug.txt
```

and if you were going to do a lot of work involving files in this directory, this would soon become irksome.

| | |
|---|---|
| `cd ~csweb5`<br>`ls` | make the home directory of user `csweb5` into the current working directory and then list the files that it contains |
| `cd CSLib`<br>`ls` | make the sub-directory `CSLib` of the current working directory (i.e. the `~csweb5/CSLib`) into the current working directory and then list its contents |
| `cd` (or `cd ~`) | 'return' to your home directory, i.e. make the user's home directory into the current working directory. |

**Keeping track of the current working directory**
To find out what the current working directory is at any time, use the UNIX command

```
pwd
```

This displays the full name of the current working directory. `pwd` stands for **p**rint **w**orking **d**irectory. `pwd` is useful if you forget where you are when 'navigating' around the UNIX file-system with `cd`.

**Manipulating Files:**

**Copying**

| | |
|---|---|
| `cp` *`file1 file2`* | copy the contents of the file *`file1`* to another file called *`file2`*. If *`file2`* already exists, its original contents are overwritten; otherwise a new file is created. |

**Renaming**

| | |
|---|---|
| `mv` *`file1 file2`* | rename (or move) file *`file1`* to have the new name *`file2`*. If a file with the name *`file2`* already exists, its original contents are overwritten. |

**Deleting**

| | |
|---|---|
| `rm` *`file`* | remove (that is delete) the specified file. |

These three commands should be used carefully since otherwise one may lose valuable information by accidentally deleting or overwriting it. Once a file has been overwritten or deleted its original contents are lost for ever[1].

[1] A back-up of user files onto magnetic tape is made regularly each night and it is possible for the system administrator to retrieve deleted files from the back-up tape (provided that the file was created before the last back-up!). However retrieval is a time-consuming process (and embarassing for the person asking for the file to be restored!).

**Examples**
Make a (back-up) copy of `prog1.adb` called `prog1.bak`

```
cp prog1.adb prog1.bak
```

Rename the file `prog1` (perhaps an executable file produced by the Ada compiler) to something more meaningful

```
mv prog1 hello_world
```

**Working with Directories**
After using UNIX for some time a typical user might have several hundred files in his home directory. In order to manage a large number of files it is better to place related files together in sub-directories (or folders) of the user's home directory. To create a new directory the command `mkdir` is used, for example you would create a sub-directory in which to store your Ada programs as follows:

**Creating a folder (or directory)**

| | |
|---|---|
| `mkdir Ada` | create a new folder with the name `Ada` in the current working directory. |

Normally users are only allowed to create new directories and files in their own home directory (and in sub-directories of their home directory) -- so make sure you do a `cd` to your home directory before doing a `mkdir`.

**Removing a folder**

| | |
|---|---|
| `rmdir tmp` | delete (i.e. remove) the directory `tmp` in the current working directory. |

Normally the directory to be deleted must be empty (i.e. all files must first be deleted from it using `rm`). However it is possible to remove a folder and all the files (and sub-folders that it contains) by using the `rm` command with the `-r` option (`-r` for **r**ecursive delete). Obviously this command should be used with extreme caution!

| | |
|---|---|
| `rm -r tmp` | remove all files and sub-directories of the directory `tmp` of the current working directory and then delete `tmp` folder itself |

**More on Copying and Renaming**
The second argument of the commands `cp` and `mv` may be a directory instead of a file. In this case the commands behave slightly differently: assuming `Ada` is a sub-directory of the current working directory then the command

```
cp ~csweb5/prog1.adb Ada
```

creates a copy the file `prog1.adb` from the home directory of user `csweb5` in the sub-directory `Ada` of the current working directory. The copy of the file is also called `prog1.adb`.

Similarly the command

```
mv prog2.adb Ada
```

would move the file `prog2.adb` (from the current working directory) into the sub-directory `Ada`. Thus we see the reason for the naming of the `mv` command -- it is short for move.

What if the first argument of `mv` is also a directory? For example:

```
mv dir1 dir2
```

This behaves differently depending on whether `dir2` already exists or not. If `dir2` exists, `mv` moves directory `dir1` (and all its contents) making it a sub-directory of directory `dir2`. If `dir2` does not already exist, then `dir1` is renamed to be directory `dir2`. In the command

```
mv dir1 file2
```

If `file2` exists and is a regular file (rather than a directory), then an error message is output; we can't change a file into a directory with `mv`.

**Hidden Files**
Files (and directories) whose names begin with the full-stop character are not displayed by the basic `ls` command. These files are said to be 'hidden' or 'invisible' files. Note that the full-stop is usually pronounced as 'dot' in UNIX. Conventionally files whose names begin with a dot are not used for everyday working. Instead they are used to customise a user's environment. For example if the files: `.login`, `.tcshrc`, `.dt`, `.netscape` and `.emacs` exist in a user's home directory, they are read automatically when a user logs in, starts a new TC-shell, starts the Window system, launch the Netscape web browser or invokes the Emacs editor respectively.

**Do not change or delete any of these files or you may corrupt your UNIX environment.**

All files in a directory including hidden ones can be displayed by using the `ls` command with the option `-a`.

| | |
|---|---|
| `ls -a` | list the names of all files including hidden ones in the current working directory |

**The . and .. directories**
Every directory contains two **irregular** hidden entries:

| | |
|---|---|
| `.` | a pointer to the directory itself |
| `..` | a pointer to the **parent** directory (i.e. the directory immediately above it in the file system tree). |

These pointers can be used in pathnames, for example:

| | |
|---|---|
| `ls ..` | list the names of all 'visible' files in the parent directory of cwd |
| `ls -a ..` | list the names of all files (both visible & hidden) in parent directory |
| `cd ..` | change working directory to the parent of the cwd |
| `cd ../..` | move two levels up in the file system hierarchy |
| `less ../Ada/prog1.adb` | list the contents of the file `prog1.adb` in the sub-directory `Ada` of the parent directory of the cwd. |
| `cp /etc/motd .` | copy the contents of the message of the day file to the current working directory (note the final 'dot') |

**File Name Expansion**
Many UNIX commands involve specifying one or more files, UNIX makes it easier to specify file names by providing **file name expansion**. A **pattern** is used in place of a file name and the pattern is expanded to give a list of all files matching the pattern. Certain characters have a special meaning when used in a pattern:

| | |
|---|---|
| ? | a 'wildcard' character which matches any single character (except for / and leading full-stops) |
| * | a 'wildcard' character which matches any string of characters of length zero or longer (except for / and leading full-stops). |
| ~ | when used alone, it refers to the current user's home directory. When used with a userid, it refers to the home directory of the user specified. |
| […] | match any of the characters enclosed in the square brackets |

A few examples should make this clear.

| | |
|---|---|
| `rm *` | delete all (unhidden) files in the current working directory (use with extreme care!!) |

| | |
|---|---|
| `ls prog1.*` | list the names all files with the basename `prog1` in the current working directory (e.g.. `prog1.adb`, `prog1.o`, `prog1.ali`, etc.) |
| `ls test.?` | list the names of all the files with the basename `test` and a **single** character extension (e.g. `test.c`, `test.o`, `test.s` etc.) in the current working directory. |
| `rm unit[1234].adb` | delete the files `unit1.adb`, `unit2.adb`, `unit3.adb` and `unit4.adb` (if they exist in the current directory) |
| `ls *.adb` | list the names of files whose names have the extension `.adb` in the current working directory. Note that the wildcard characters `*` and `?` do not match a leading dot in a file name. So this command would not list any hidden files with the extension `.adb` |
| `ls .??*` | list the names of all hidden files in the current working directory whose names are at least three characters long including the dot |
| `cd ~csweb5` | 'move' to the home directory of user `csweb5` |
| `cd CSLib` | 'move' into a sub-directory of this called `ISP` |
| `cp cs_int_io.ads ~` | copy the file `cs_int_io.ads` from this sub-directory to your own home directory. |

**Access Control to Files**

A typical UNIX file-system contains many thousands of files owned by hundreds of different users. The UNIX operating system must provide mechanisms by which users can control access to their own files and so, for example, prevent unauthorised users from reading or modifying their files.

For each file UNIX records (among other things) the userid of the owner of the file, the groupid of the file, the type of the file (for example whether it is a directory or a regular file) and a protection code. The **protection code** governs access to a file by three classes of user: the owner of the file, a member of the owner's group and any other user on the system.

| | |
|---|---|
| `u` | the owner of the file or directory (referred to as the user) |
| `g` | members of the owner's group. |
| `o` | all other users |

The creator of a file (normally) becomes the owner of a file. Each user also belongs to a group and that group becomes the groupid of the file when it is created by a user.

The protection code consists of nine bits: the first three bits of the code specify the u-access (user's or owner's access), the next three the g-access (group access by users belonging to the same group as the owner) and the last three the o-access (others access). Each set of three bits controls the three basic types of access that are permitted by UNIX

| | | |
|---|---|---|
| `r` | read access | the contents of the file may be inspected (e.g. using `cat` or `less`) |
| `w` | write access | the contents of the file may be altered (e.g. using an editor) |
| `x` | execute access | the file may be executed (assuming it is an executable program) |

**Access Control to Directories**

The protection code for directories have a somewhat different meaning as compared to regular files:

| | | |
|---|---|---|
| `r` | read access | the contents of the directory may be listed (for example) using `ls`. |
| `w` | write access | allows a user to create and delete files in that directory. Note that a user can delete a file even though (s)he does not have write access to the file itself! |
| `x` | search[2] access | allows the directory to be searched to see if a file is present. Note that without `x` (search) access to a directory, a user has no access to any of the files within that directory. With `x` access to a directory, users can |

[2] Clearly 'executing a directory' is meaningless.

access a file subject to that file's individual access permissions but only if they know the exact name of the file (wildcard characters can only be used if you have both read and search access).

**Notes**

In order to have access to a file, a user must have search access `x` to **all the directories on the full pathname of the file**. Thus in particular a user must have search access to the directory in which the file resides.

Normally if a directory is to be made accessible to a certain class of users, then both read and search access (`rx`) is granted; then files in the directory may be accessed (subject to their individual file access permissions) and `ls` can be used to find out the contents of the directory and wildcard characters may be used to specify files.

Note also the difference between write access to a file and a directory. With only write access to a file one may alter the file's contents (for example with an editor), but one may not delete the file completely. With only write access to a directory one may delete completely any file in that directory, but one may not alter a file's contents in any other way.

It is quite common for users to allow read and execute (search) access to (some of) their files and directories so that information can be shared. However for fairly obvious reasons it is rare for users to grant write access to other users.

**Full directory listing**

A long directory listing may be obtained using the option `-l` with `ls`. This listing contains information on the size of the file, its owner, its date of creation, and its access permissions etc.

`ls -l` *`dir-name`* give a full directory listing of all visible files in *`dir-name`*.

Options may be combined (in any order), for example to get a full directory listing of all files

`ls -l -a` *`dir-name`* or equivalently `ls -la` *`dir-name`*

**Displaying a file's access permissions**

The output of the long form of the `ls` command (`ls -l`) represents the information about the file-type and its protection code symbolically as follows:

| | | | | |
|---|---|---|---|---|
| File-type (1st character) | `d` | directory | `-` | regular file |
| Owner read access (2nd character) | `r` | read access | `-` | no read access |
| Owner write access (3rd character) | `w` | write access | `-` | no write access |
| Owner execute access (4th character) | `x` | execute access | `-` | no execute access |
| Group read access (2nd character) | `r` | read access | `-` | no read access |
| Group write access (3rd character) | `w` | write access | `-` | no write access |
| Group execute access (4th character) | `x` | execute access | `-` | no execute access |
| Others read access (8th character) | `r` | read access | `-` | no read access |
| Others write access (9th character) | `w` | write access | `-` | no write access |
| Others execute access (10th character) | `x` | execute access | `-` | no execute access |

**Examples**

| | |
|---|---|
| `-rw-r--r--` | Regular file. Owner has read/write access. Members of owner's group and all other users have read-only access. |
| `-rwxr-x---` | Regular file. Owner has read/write/execute access. Members of owner's group have read/execute access. No access by Others |
| `drwxr-xr-x` | Directory file. Owner has read/write/search access. Group and Others have read/search access |
| `----r-x---` | Regular file with rather bizarre access permissions: all members of owner's group **except the owner** can read and execute the file[3]. |

[3] Note group permissions refer to everyone in the group except thre owner. Similarly other permissions refer to everyone except the owner and members of his/her group.

**SuperUsers**
There is a fourth class of users called `root` users (or super-users) to whom these file access restrictions do not apply. A user gets superuser status if (s)he logs in under the userid `root`. Of course only the system administrator and a few 'trusted' users should know the `root` password as they have unrestricted access to all files on the system.

**Getting more info on UNIX commands**
The behaviour of most UNIX commands may be modified by using command-line options (such as the options `-l` and `-a` for `ls` or the option `-r` for `rm`. It is not sensible in an introductory course such as this to describe all the available options for each UNIX command (since most self-respecting UNIX commands come with at least a dozen options!). However all this information is available on-line on any UNIX system by using the `man` (manual) command.

For example to display the UNIX manual pages for the commands `ls`, `cp` and `less` type

```
man ls                  man cp            man less
```

The required UNIX manual page is then displayed page by page on the VDU screen. This gives information on the purpose of the instruction, a synopsis of how it can be used together with a information on all the options available with this command, a list of related commands, known bugs etc..

**Emacs Automatic Back-up Files**
As a safety measure, Emacs retains the previous version of any file that you edit. This is useful if you completely mess up an edit and make the mistake of saving the changes; you still have a copy of the original version safely stored on disk. The original file is saved as an **automatic back-up**. The name of the back-up file is obtained by appending a tilde (~) to the original file name. For example if the original file was called `prog1.adb`, the back-up will be called `prog1.adb~`. Note if you completely mess up an edit, rename the backup file to be the original file name

```
mv prog1.adb~ prog1.adb
```

and then edit the file -- don't edit the automatic back-up file directly with Emacs.

**Emacs Auto-Save Files**
As an extra insurance, as you edit a file, Emacs always creates (and periodically updates every few minutes) an **auto-save file** so that if the system crashes in the middle of a long editing session you only lose a few minutes work. The name of the auto-save file is automatically generated by Emacs by adding the hash character `#` to the front and end of the filename. For example the auto-saved version of `prog1.adb` is called `#prog1.adb#`.

If Emacs exits normally the auto-save file is deleted, but if Emacs exits abnormally (due perhaps to a system crash or if you quit without saving your work) the auto-save file will be retained.

Again if you want recover a file after a system crash from the auto-save file, rename the auto-save file to have its original name

```
mv #prog1.adb#  prog1.adb
```

Then edit the file in the normal way. Beware editing the auto-save file directly with Emacs as this turns of the auto-save protection!

**Saving Disk Space**
Automatic back-up files and auto-save files created by Emacs take up valuable disk space. Periodically such files should be deleted. This can be done by issuing the UNIX command `clean` in the directory where the files are located. Remember this also cleans up object files and Ada Library Information files and bind files created by the Ada compiler.