Unix, Linux and C Programming	**LABORATORY**	**ONE**
OBJECTIVES		
1. Accessing Unix / Linux 2. Compiling and Running C Programs 3. Using Unix / Linux 4. C Programming		

Accessing Unix / Linux

In Department of Computing, your Unix / Linux account shares the *same username* and *password* as the PC account. These accounts are different from the university account for checking emails. Moreover, these departmental accounts share the *same storage space*. All the Unix / Linux files can be found in a special network drive, i.e., **J:** drive. So you could directly store a file in Unix / Linux by copying it to **J:** drive. ***Caution****: DO NOT delete those files starting with "." in Linux. Those are* ***hidden files*** *on Linux platforms. Also, avoid creating any file or folder with Chinese names in* ***J:*** *drive. They become garbage or inaccessible to Linux and can cause very strange behavior when shell scripts are executed (in the next lab).*

If you are using a Windows-based PC in the lab, you will still have to connect to the Linux system. Under the SSH Secure Shell on a PC, select Secure Shell Client. Then you will see a window asking you to hit **`Enter`** or **`Space`** to continue the connection. This program is called ***s****ecure* ***sh****ell* (**`ssh`**). Type in the name of a Linux machine (e.g., **apollo** or **apollo2**) and your username, then choose **`connect`**. After typing in your password, you will see the Linux *prompt*; treat it as like a DOS shell. You can type in *commands* for execution under the Linux *prompt*.

For example, to see the files you have created and maintained, type **`ls`**, which means "*list*" for historical reason. Then you will see the list of files under the current directory (i.e., folder). After you log in via a Mac, you will see files that you own, including the **`Desktop`** folder. If you want to delete a file, type **`rm`** followed by the filename. However, once you delete a file via **`rm`**, it ***cannot be recovered*** (i.e., there is *no trash bin* that you can look into). You could delete a file called **`hello.txt`** by typing:

```
rm hello.txt
```

The final few Unix server machines in the department, **rocket** and **rocket2** had been phased out. Unless you are working in some other companies, you cannot access Unix machines, but Linux machines in the lab. Anyway, the command line interfaces are almost identical between Unix and Linux. Sometimes, Unix and Linux are called ***nix** systems.

Note that in Unix / Linux, *upper case and lower case characters are treated* ***differently*** as in Python. For example, two files named **`ASSIGNMENT`** and **`assignment`** are ***not*** the same file by default. However, they might also be treated the same when being overwritten, but not the same when being compared on a Mac. Furthermore, on **apollo**, the current version (not previous version) of Linux (called CentOS) allows upper and lower case filenames to be considered equivalent. This is the server grade Linux from **Red Hat**, the most common OS running on web servers. Note that it is a *convention* in Unix / Linux that lower case letters are used normally for files and commands, whereas upper case letters are for directories. ***Always avoid*** special characters and space in file and directory names.

1. Login

- If you are on a Unix machine (just in case you have), just login the system and you will see the desktop environment.
- If you are using a PC, invoke Secure Shell (or putty) to **`connect`** to Linux workstation (**apollo**).

2. Logout

- When you are done, you should log out from the system. In Unix / Linux, type **`logout`** to terminate the SSH connection. You then log out of Windows in case you are using a Windows machine or log out of the Mac accordingly under the ***Apple icon*** on the upper left corner.

Using **pico** Editor

The primary importance of an OS is the ability to issue commands. The next important functionality is to create and modify files. Unix / Linux provides the common text editor called **`pico`**, which is similar to another one called **`nano`** (*nano* – 10^{-9} is "larger" than *pico* – 10^{-12}). This editor **`pico`** is similar to **`edit`** in DOS or **`notepad`** in Windows. Most Linux users are using the more powerful **`emacs`** and **`vi`** editors, but they are much harder to learn for beginners. Note that it is very useful to know how to use a text editor to modify a program file stored on **J:** drive, since this is the common mode of file modification when you are working at home, connecting to the department for the Linux system.

At the bottom, you will see a list of common commands that you could use. The normal commands used are <**Ctrl-X**> to exit the editor, <**Ctrl-O**> to save the current file (similar to the *SaveAs* command in Word), and <**Ctrl-R**> to read in a file. Try to use it for a while to get familiar with it.

Try to create your own file **`hello.c`** with some content, e.g., *good morning, hello world*:

```
pico hello.c
```

Compiling a C Program

In C, all programs are named with extension **`.c`** (like Java programs with extension **`.java`** and Python programs with extension **`.py`**). You must *compile* a C program before running it, similar to what you have done with a Java program, though you do not need to compile a Python program in order to run it. To compile a program called **`hello.c`**, use **`gcc`** compiler (**`cc`** or **`gcc`**). On **apollo**, the two actually refer to the same compiler. Note that you should not compile a C program on **apollo** and run on a Mac, or vice versa, since their executable codes are different (for different machines).

- Type **`cc hello.c`**
- A file (executable program) called **`a.out`** is created, which can be verified by typing **`ls`**.
- If you want to give the executable the name **`hello`** instead of the *default* name **`a.out`**, you could type **`cc hello.c –o hello`**
- To run the program called **`a.out`**, just type **`./a.out`**

You could create C programs using *any editor* such as **`nano`** or **`vi/emacs`** or create it using a PC using **`notepad`** and then *copy* to **J:** or use **`scp`** (**secure copy**) to transfer the file to the Linux machine for compilation. Note that PC (actually Windows) terminates a text line by <LF><CR> but Linux terminates a text line only by <LF>. If the program **`hello.c`** that you created on a PC ***does not compile*** on Unix / Linux, try to ***convert*** your file by removing the excessive <CR> at the end of a line. Note that this is a very common problem when a **`shell`** program does not run (you would likely encounter it when we cover *shell programming* in next lab). The conversion command is:

```
dos2unix hello.c
```

Just in case that the command **`dos2unix`** is not available on the system (and it is not there on Mac), you can use the following more clumsy command **`tr`**, where **`helloPC.c`** is an existing file and **`helloLinux.c`** is a new file:

```
tr –d '\r' < helloPC.c > helloLinux.c
```

Try typing in your first C program (**`hello.c`**), compile and then run it.

```
// lab 1A
#include <stdio.h>

int main()
{
   printf("Hello World\n");
}
```

Using Unix / Linux

Under Unix / Linux, all files are organized into a hierarchical structure, called *directories*, in a similar way as a set of Windows *folders*. Some commonly used commands in Unix / Linux:

Linux/Unix command	***Description***	***Example***	***Approximate DOS command***
ls	list files in the current directory (**ls** functions like **dir/w** and **ls** -l gives full information)	**ls** -l	**dir**
man	provide a help topic	**man** ls	**help**
cat	show the content of a file	**cat** test.c	**type**
cp	copy a file	**cp** test1 test2	**copy**
mv	rename a file	**mv** test sample	**rename**
rm	delete a file	**rm** sample	**del**
nano	edit a text file	**nano** test	**edit**
gcc	compile a C program using GNU C compiler	**gcc** hello.c	**tc** (Turbo C)
cc	linked to the same **gcc** compiler currently	**cc** hello.c	**tc** (Turbo C)
mkdir	make a new directory	**mkdir** comp2432	**md**
rmdir	remove an old directory	**rmdir** temp	**rd**
cd	change directory	**cd** comp2432	**cd**
pwd	show the current directory	**pwd**	-
chmod	change permission mode of a file/directory	**chmod** 700 12345678d	**attrib**
more	show content page by page	**more** test.c	**more**
alias	redefine the meaning of a command	**alias** rm='rm -i'	-
grep	search for text within files	**grep** printf hello.c	**find**
wc	count lines, words, characters	**wc** sample.txt	-
diff	compare the contents of two files	**diff** test1.c test2.c	**fc**
who	show existing users on the system	**who**	-
whoami	show the current user	**whoami**	-
id	show user id and group for current user	**id**	-

Some of you may wonder why there is a slight difference between the prompts that you see in the Unix / Linux system and the output from **ls** command:

```
[12345678d@apollo] /home/12345678d:>
apollo2 12345678d>
```

All these differences are related to system setups in Unix / Linux (controlled by configuration files, the most important ones being **.login** file and also **.bashrc** file or **.cshrc** file). The appearance and behavior of commands could be altered through proper changes in those two files. You can get *help* and *descriptions* about a command from Linux with **man** (not quite working for **apollo**) meaning *manual*. On a Linux machine like **apollo2**, you could also get help by **--help** (e.g., **ls --help**) for the *possible options* for the command **ls**.

Try to type **alias ls='ls -l'** and you will see more information about all the files when you use **ls** now. The **alias** command assigns a new meaning to **ls**, so that now **ls** means **ls -l** (i.e., produce the list of files in *long listing format*). This is the most common way a user would like to use with **ls**. You can change it *according to your need*. Most of you may find out that when using **rm** to delete a file, you will be prompted to *confirm*, due to the alias being set to **rm** command. In case you do

not get prompted, type **`alias rm='rm -i'`** (*interactive prompt*), perhaps also **`alias del='rm -i'`**. This will help avoiding accident file deletion (there is *no trash bin*).

The name **`/home/12345678d`** is the *absolute path* of your *home directory*. The files for each user are stored under different storage space. To show the current directory, you can type **`pwd`** and it will be displayed. The current directory is also referred to by ".". That is the reason why when you run **`a.out`**, you would type **`./a.out`** (it means to run **`a.out`** in the *current directory*). If you want to avoid typing the current directory "." (**`./`**) when executing a program, you could include the current directory in the *search path*, by typing **`export PATH="$PATH:."`** and this command changes the variable **`PATH`** containing the command search path. After doing this, you just need to type **`a.out`** to run the program instead of **`./a.out`**.

You could create subdirectory (**`mkdir comp2432`**), remove subdirectory (**`rmdir comp2011`**) and go around directories and subdirectories (**`cd comp2432`**). Try to organize your files under proper directories. You could go up one level of directory by typing **`cd ..`** ("**`..`**" means one level above). If you get lost in going around within directories, just type **`cd`** and you will return home.

You may want to change your password on Unix / Linux. The department has unified the password management system for passwords on PC and Unix / Linux so that you only have to remember a single password. Make sure that your password contains a mixture of upper case, lower case, digit and symbol to make it more secure. You could modify your password through a *password server* (**`pwsvr`**) accessible via a secure web connection: **`https://pwsvr.comp.polyu.edu.hk`**

Working from Home

If you work *at home* and want to have access to Linux and Unix, you could access the departmental machines. If you are using a PC, you have to use **`ssh`** or **`putty`** to connect to them. The departmental machines are *protected* against arbitrary direct access from outside PolyU. That will provide better protection against hackers. To connect to a machine in the department, say, **apollo**, you need to connect first to the *gateway machine*, called **csdoor**. Once you are in **csdoor**, you could connect to other machines in the department.

- To connect from home, execute **`ssh 12345678d@csdoor.comp.polyu.edu.hk`**
- Type your password and answer ***yes*** when you are prompted about the RSA key fingerprint
- After logging in to **csdoor**, type the name of the machine to connect to, for example, **apollo**
- You now enter your password to login the machine

If you do not have **`ssh`** installed on your PC, you could download, install and then execute **`putty`** from **`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`** followed by typing **`csdoor.comp.polyu.edu.hk`** as *hostname*. Select *port* **22** and *connection type* **`SSH`**.

After you have logged in to **csdoor**, you could also type **`load`** to check for the loading of the list of available machines before you decide to select a machine with fewer users to connect to. There are currently two accessible Linux machines, namely, **apollo** and **apollo2**. If you are using a Mac, you could access the departmental machines in a similar way.

If you are using a Mac, you are actually already running Mac OS/X, which is built based on a variant of Unix called Mach, compatible with the standard Unix, i.e., *BSD-compliant*. Simply start a *terminal* on your Mac and you could type in Unix commands accordingly.

A Fast Recap on C Programming

Recall that C programs have a similar control structure to Java programs and are more complicated than Python programs. C language provides the simple flow control (if-then-else, for-loop, while-loop) and procedure calls. There is no class, instance, inheritance, and polymorphism. All C programs have a *main*

procedure that contains the main program body. In the **`hello.c`** program, the first line is to declare the use of standard I/O routines (we used the output routine printf in this case).

You may use void main or int main for the type of the main procedure. Here void means that no value is returned from main (as a procedure) and int means that an integer return value could be resulted from the execution of main (as a function). Note that void may not work for some Linux systems. In C, functions and procedures take the *same format*, with only a difference in void versus a return data type. Note that there is no boolean (normally we use char), or byte (we use unsigned char) type in C.

Since the basic concepts in Java are derived from C++ (which is derived from C), it is not surprising that C shares a lot of similarity with Java. As a result, it is very easy to write a first C program if you know Java. It is also not too difficult if you can program in Python. The first lesson in writing C programs for a Python programmer, just in case you are one, is perhaps the use of I/O formatting statements and then the data types, since Python is so tolerant in data types and I/O. Note that in C, for variable input (using scanf), it is not sufficient to give the name of the variable, but one must give the *address* of the variable. The *address* of variable **`data`** is **`&data`**. This is a very common error for a non-C programmer.

- printf() is a function for output in C program. It has at least two arguments: the first is a string for formatting, followed by the list of variables to be output.
- The format string is a string containing argument formatting specification. Each format specifier begins with the percent sign (**%**), followed by the actual format. Each format specifier is paired up with the corresponding argument sequentially.
- Common specifiers: **%d** for integer (including short and long), **%f** for floating point, **%lf** for double, **%c** for character, **%s** for string, **%x** or **%X** for integer in hexadecimal format (lower case and upper case), **%e** for exponential format of floating point.
- Special formatting strings: **n** means new line, **t** means tab, **"** for double quote, **a** to generate a beep sound, **%%** for percentage sign.
- You could further control the formatting of integers with **%d** etc. For example, **%3d** means space of at least 3, **%7.3f** means space of at least 7 to a floating point, with 3 decimal spaces.
- scanf() is a function for input in C program. It has a formatting string followed by *addresses* of variables for input.
- Common specifiers: **%d** for integer (including short and long), **%f** for floating point (including double), **%c** for character, **%s** for string.
- **Do not forget** to pass in the *address* in the argument list (prefixing with & to the variable). The name of an array serves the purpose of an address (no & is needed).
- A better alternative is to read the whole line in a buffer and use sscanf() to process the string buffer.

A key concept in C language that is different from Java and Python is the widespread use of ***pointers***. In C, all variables are just the names to different blocks of storage in memory. An integer is the name to a block of 4 bytes and a character array of size 10 is the name to a block of 10 bytes. A pointer refers to the *address* of a variable. A *pointer variable* is a variable that stores ***not*** an integer or a character, but the ***address*** to another variable, which is one of the hardest things for beginners.

Warning: once you get a pointer in C, you could access to *anywhere* around the pointer. In Java, accessing outside an array will throw an exception, but *no checking* is done in C. It will be very **dangerous** because you could end up accessing (reading or writing) to the space of other variables, or even outside the program. In Linux, this will cause a program to crash (and a *core dump*). In older Windows, that will cause unpredictable damages. Mostly, the system would crash or hang up, but if the damage is to change the OS behavior (a *virus* is actually doing that), no one will know what the consequence could be (perhaps erasing all your data). On the useful side, this is a ***necessary feature*** when implementing an OS and other low level functions, because we can use the pointers (if used carefully) to do *almost anything*. A comparison of a C program with a program written in most other *conventional languages* (C++, Java, C#, Pascal) to achieve the same functionality will demonstrate that C programs are usually among the *smallest*.

In C, *string* is ***not*** a special data type. It is just an *array of characters*. Thus, a string "**Hello**" in C is just a character array of length **6**. The extra character is the **NULL** character (ASCII code 0) that *terminates* a string in C (i.e., signals when a string ends like an end-of-data marker). In other words, C strings are *null-terminated*. To see this, try the following program **lab1B.c**.

```
// lab 1B
#include <stdio.h>

int main()
{
  char firstStr[6] = "Hello";
  char secondStr[6];
  int  i;

  secondStr[0] = 'H'; secondStr[1] = 'e';
  secondStr[2] = 'l'; secondStr[3] = 'l';
  secondStr[4] = 'o'; secondStr[5] = 0; // 0 is the same as NULL
  printf("String 1 is %s\n", firstStr);
  printf("String 2 is %s\n", secondStr);
  for (i = 0; i < 6; i++)
      if (firstStr[i] == secondStr[i])
         printf("Position %d is same for the two strings: %c\n",i,firstStr[i]);
}
```

Now, try to change the last assignment statement into **secondStr[5] = '!'** (try making it "**Hello!**"). What do you see? This tells you that you *must terminate* a string in C by **0** or **NULL**. Otherwise, the string will only terminate *until a zero* is reached (and that *zero* could be very *far away*).

Actually, the assignment of "**Hello**" to **firstStr** is a fast way in C to initialize an array of characters. It could also be an array of integer. For example, the following initialize two arrays with appropriate values:

```
int month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
char firstStr2[6] = {'H', 'e', 'l', 'l', 'o'}; /* rest of array filled with 0 */
```

If you do not want to count, C allows you to omit the number and it will *count automatically* for you. So you could use these as well:

```
int month2[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
          /* array size=12 */
char firstStr3[] = "Hello";
          /* array size=6, a null automatically added to terminate the string */
```

However, you should *watch out* for declaration like this, since it is problematic:

```
char firstStr4[] = {'H','e','l','l','o'};
```

Can you tell what the problem is for the declaration **firstStr4** above? How could you *correct* it?

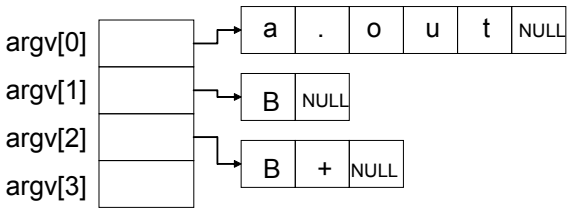
C does not distinguish very seriously between an *array* and a *pointer* in terms of data access. Internally, an array is just considered as *a pointer to a table* (array). The difference between an array and a pointer is that an array is made up of *memory space already allocated* to the program, but a pointer has *no space allocated* for where it points to. You either use **malloc()** to *allocate* the memory, or use the pointer to point to another *existing data structure*. Also, C does not care too much about the size of an array, so you must be very careful to avoid index overflow or out-of-bound error. Unlike similar problems in Java programs giving an exception, C programs just *crash* at some inappropriate moment.

For those who are curious and exploratory, can you guess what the following piece of C code with a while loop is trying to do? Try to fill in some lines in the program segment and test it out. Using C language, one can really do a lot of things with a small program, but you need to be *smart* and *careful*.

```
char *p, *q;

// some program code here
while (*p++ = *q++);
```

The use of pointers to express a string is very useful, particularly for command line argument processing. In C and Java, you can access arguments on the *command line*, via arguments inside **`main()`**, where **`argc`** returns *the number of arguments* in the command line including the program itself and **`argv`** is *an array of pointers* to the arguments themselves, as illustrated below.



This array contains *n* elements, where *n* is the value of **`argc`**, and each pointer element in **`argv`** is a pointer for that particular argument (the string for the argument). Consider the following C program to compute the GPA from a list of grades. This is quite a typical technique. You can see an efficient way of processing the *argument list*. You do not even need to count them!

```
// lab 1C: please fix the program
#include <stdio.h>

int main(int argc, char *argv[])
{
  int   num_subj;
  float in_gp, sum_gp = 0.0;
  char  in_grade;
  int   i;
  // argv[0] is the name of the program
  printf("This program is %s\n",argv[0]);
  num_subj = argc-1;
  printf("There are %d subjects\n",num_subj);
  for (i = 1; i <= num_subj; i++) {
      in_grade = argv[i][0]; // get the first character
      switch (in_grade) {
        case 'A': in_gp = 4.0; break;
        case 'B': in_gp = 3.0; break;
        case 'C': in_gp = 2.0; break;
        case 'D': in_gp = 1.0; break;
        case 'F': in_gp = 0.0; break;
        default: printf("Wrong grade %s\n",argv[i]);
      }
      if (argv[i][1] == '+') in_gp = in_gp + 0.3;
      if (argv[i][1] == '-') in_gp = in_gp - 0.3;
      sum_gp = sum_gp + in_gp;
    }
  printf("Your GPA for %d subjects is %5.2f\n",num_subj,sum_gp/num_subj);
}
```

Note that there is a serious bug in the program **lab1C.c** above. Whether it shows up depends on the input data. Do you know what it is, and how it occurs?

A dynamic array in C could be defined using **`malloc`**(*s*) for an array of size *s*. This would reduce wasted storage. It is a strongly recommended programming style to *release* the allocated memory after its use. This will avoid the leakage of memory in your program, especially for a long-running program. As a result, those **`malloc()`** and **`free()`** calls often come in pairs. To use **`malloc()`** and **`free()`**, you should *include* the appropriate library, **`stdlib.h`**. If you want to use *string processing functions* in C, you should *include* the library **`string.h`**.

```
#include <stdlib.h>
  float *gp;    /* an example of a dynamic array of floating point */
  gp = (float*) malloc(sizeof(float)*num_subj);
  gp[i] = computeGP(...); /* gp is a floating point array in this example */
  free(gp);
```

Modify and ***correct*** **lab1C.c** to *print out* the grade and print out also the grade point for the subjects *in that order*. For example, output produced when running

GPA A B+

Grade for subject 1 is A, GP 4.0
Grade for subject 2 is B+, GP 3.3
Your GPA for 2 subjects is 3.65

This GPA system was adopted by PolyU since 2020. As a comparison, some other universities are adopting some other GPA systems, e.g., some in Canada. Let us call them **PolyU System** and **Other System** respectively as follows:

Grade	A+	A	A-	B+	B	B-	C+	C	C-	D+	D	D-	F
PolyU System	4.3	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.3	1.0	-	0.0
Other System	12	11	10	9	8	7	6	5	4	3	2	1	0

Modify your program so that it can calculate the GPA under *both systems*. Invalid grades are discarded when GPA is being calculated. For example, grade **D-** is invalid for PolyU system. For simplicity, you can assume that there are no other invalid grades such as **C++** or **E** being entered, and there are *at most 10 grades* in the input list.

Feedback: After calculating the GPA, the program will offer feedback based on the calculated GPA, categorizing it into ranges (e.g., Excellent, Good, Average, Poor).

Feedback Based on GPA:

GPA >= 3.5:	"Excellent"
3.0 <= GPA < 3.5:	"Good"
2.0 <= GPA < 3.0:	"Average"
GPA < 2.0:	"Poor"

Make sure that your program can ***compile*** and ***run*** on **apollo** or **apollo2**. Submit your program with name **GPA.c** to **BlackBoard** on or before **02 February 2025** to claim *participation score*.

Output produced when running

```
GPA A B D+

PolyU System:
Grade for subject 1 is A, GP 4.0
Grade for subject 2 is B, GP 3.0
Grade for subject 3 is D+, GP 1.3
Your GPA for 3 valid subjects is 2.77
Feedback: Average
Other System:
Grade for subject 1 is A, GP 11
Grade for subject 2 is B, GP 8
Grade for subject 3 is D+, GP 3
Your GPA for 3 valid subjects is 7.33
Feedback: Excellent
```

Output produced when running

```
GPA A+ D- A F B+

PolyU System:
Grade for subject 1 is A+, GP 4.3
Grade for subject 2 is D-, invalid
Grade for subject 3 is A, GP 4.0
Grade for subject 4 is F, GP 0.0
Grade for subject 5 is B+, GP 3.3
Your GPA for 4 valid subjects is 2.90
Feedback: Average
Other System:
Grade for subject 1 is A+, GP 12
Grade for subject 2 is D-, GP 1
Grade for subject 3 is A, GP 11
Grade for subject 4 is F, GP 0
Grade for subject 5 is B+, GP 9
Your GPA for 5 valid subjects is 6.60
Feedback: Excellent
```