

More Programming in C	LABORATORY	TWO
<b>OBJECTIVES</b>		
1. More on Unix and Linux 2. Connecting to Unix and Linux Machines 3. More C Programming		

### More Unix and Linux Commands

Some of you may wonder the slight difference between the prompt you see in the Unix/Linux system and the output from `ls` command:

```
[lab216:~] /Network/Servers/nwc1h1/home/11234567d %
[11234567d@apollo2] /home/11234567d:>
apollo2 11234567d>
```

All these differences are related to system setups in Unix/Linux (controlled by configuration files, the most important ones being `.login` and `.cshrc` files). The appearance and behavior of commands could be altered through proper changes in those two files. Recall that you can get *help* and *descriptions* about a command from Unix or Linux with `man`, 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. Try to type `alias ls ls -l` and you will no longer see those hidden files starting with “.”. The `alias` command assigns a new meaning to `ls`, 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. Type `cat .cshrc` and you can verify this. In case you do not get prompted, type `alias rm rm -i` (*interactive prompt*).

To define many aliases, you may want to put them into a “batch” file and execute it (like `.bat` file in MS-DOS). Then you do not need to type in the alias commands every time you login the system. To execute a file `myalias`, for example, containing useful aliases, type `source myalias`. You are encouraged to create your own alias file to customize your Unix/Linux.

The name `/home/11234567d` is the *absolute path* of your *home directory* (note that some students may see a slightly different form of the path, e.g., second year students may see something like `/home/student1/9/10987654d`). The files for each user are stored under different space. The shaded part of the path `/Network/Servers/nwc1h1` is generated because of the connection from Mac to the actual file system (i.e., **J:** drive). 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` (to run `a.out` in the current directory).

You could create subdirectory (`mkdir comp304`), remove subdirectory (`rmdir comp305`) and go around directories and subdirectories (`cd comp304`). Try to organize your files under proper directories. You could go up one level of directory by typing `cd ..` (“..” means one level above). When you are in `/home/11234567d`, typing `cd ..` will lead you to `/home`, which is the directory for some other people. Type `ls` and you could see your friends with a Unix/Linux account in the same group as you under *standard* Unix/Linux setup. However, this “viewing” of other students does not work on Mac Unix. If you get lost in going around within directories, just type `cd` and you will return home.

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 `set path = ( . $path)` and just `a.out` would work.

When you type `ls`, you will see the permission mode about your files. You are recommended to change the permission mode for your home directory from `drwx--x--x` (everyone can “execute” your directory, i.e., can access the content inside it) to `drwx-----` (no one else could access your files). To do that, type `chmod 700 11234567d` when you are at one level above your home directory (with `cd ..`).

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 have only 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>

<i>Linux command</i>	<i>Description</i>	<i>Example</i>	<i>Approximate DOS command</i>
<b>uname</b>	show the machine and system information, e.g. machine name, CPU, OS version, etc	<b>uname -a</b>	-
<b>mkdir</b>	make a new directory	<b>mkdir comp304</b>	md
<b>rmdir</b>	remove an old directory	<b>rmdir temp</b>	rd
<b>chmod</b>	change permission mode of a file/directory	<b>chmod 700 11234567d</b>	attrib
<b>alias</b>	redefine the meaning of a command	<b>alias rm rm -i</b>	-
<b>source</b>	execute commands in a file	<b>source setup</b>	setup(.bat)
<b>more</b>	show content page by page	<b>more test.c</b>	more
<b>less</b>	show content page by page	<b>less test.c</b>	-
<b>clear</b>	clear the screen	<b>clear</b>	cls
<b>grep</b>	search for text within files	<b>grep printf hello.c</b>	find
<b>spell</b>	perform a spell check	<b>spell sample.txt</b>	-
<b>wc</b>	count lines, words, characters	<b>wc sample.txt</b>	-
<b>diff</b>	compare the contents of two files	<b>diff test1.c test2.c</b>	fc
<b>who</b>	show existing users on the system	<b>who</b>	-
<b>whoami</b>	show the current user	<b>whoami</b>	-
<b>id</b>	show user id and group for current user	<b>id</b>	-
<b>ps</b>	show the processes in the computer	<b>ps -ef</b>	mem

## Connecting to Unix and Linux Machines

When you are using a Windows-based PC (in most other labs, e.g., PQ 604A and 604B), you will have to connect to the Unix/Linux system, just like what you had done at winter workshop. 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 *secure shell* (**ssh**). Type in a Linux machine name (e.g. **apollo2** or **apollo**) and your username then choose **connect**. After typing in your password, you will be in the Linux prompt; treat it as like a DOS shell. You may also connect to a Unix machine, e.g. **rocket2**.

If you are using a terminal on a Mac, you can use **ssh** as well, by typing **ssh apollo2** to connect to **apollo2**. The Linux and Unix command line interfaces are almost identical. Sometimes, Unix and Linux are called **\*nix** systems. Another common software similar to **ssh** is **putty**.

When you are *at home*, you have to use **ssh** or **putty** to connect to machines in Department of Computing. The machine and other machines in Department of Computing 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** or **apollo2**, 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 11234567d@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, **apollo2**
- 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 possible machines to select a machine with fewer users to connect to. Machines of name **b5x-180** to **b5x-183** besides **rocket** and **rocket2** are other Unix machines that you could use and they are similar to Linux machines. Note that you should use **jpico** on Linux machines but use **pico** on Unix machines.

## More C Programming

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. In other words, C strings are *null-terminated*. To see this, try the following program.

```

/* lab 2 A */
#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. 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 will be filled by 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 mon[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; /* size=12 */
char firstStr3[] = "Hello"; /* size=6 */

```

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

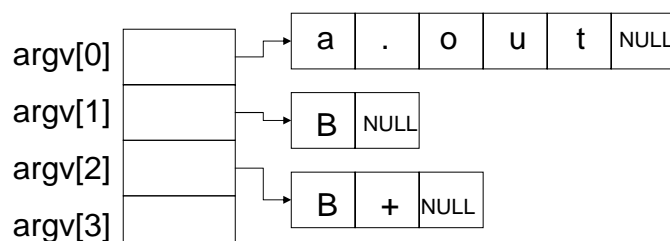
```

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

```

Can you tell what the problem is for the declaration **firstStr4** above?

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 in **Lab 1C**, you could *access anywhere outside an array* without any checking made by C.



The use of pointers to express a string is actually very useful. This is particularly useful for command line argument processing. In C and Java, you can access arguments on the *command line*, by means of the 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. The 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. You can see an efficient processing to the *argument list*. You do not even need to count them!

```

/* lab 2 B */
#include <stdio.h>

/* correct the program */
int main(int argc, char *argv[])
{
    int    num_subj;
    float  in_gp, sum_gp = 0.0;
    char   in_grade, grade[10];
    int    i;

    /* argv[0] is the name of the program */
    printf("This program name 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]; /* read 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 + 0.5;
        sum_gp = sum_gp + in_gp;
    }

    printf("Your GPA for %d subjects is %5.2f\n", num_subj, sum_gp/num_subj);
}

```

You can move the conversion of the grade point for a grade into a *function* in C. A *function* and a *procedure* are similar in C. For example, `void main()` is a procedure definition and `int main()` is a function definition. A function has a *return value*, but a procedure *does not*. Try to modify the program to make the grade point conversion a *function call*, based on the following.

```

/* function to return the GP of a grade */
float computeGP(char *grade)
{
    float value;

    /* convert the grade point and store it in value */
    return value;
}

/* in main program */
int main(int argc, char *argv[])
{
    . . .
    for (i = 1; i <= num_subj; i++) {
        in_gp = computeGP(argv[i]); /* make a function call */
        sum_gp = sum_gp + in_gp;
    }
    . . .
}

```

Recall that 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);
```

### Laboratory Exercise

Modify and correct `lab2B.c` to *print out* the grade and print out also the grade point for the subjects *in that order*. Do not forget to *check* your program and make sure that your program can be *compiled* with `gcc` compiler on a Linux machine or Mac. Programs that *cannot be compiled* on Linux or Mac with `gcc` will automatically be considered *incomplete*. Please provide *appropriate comments* in your program, including your *Name* and *Student No*.

```
Grade for subject 1 is A, GP 4.0
Grade for subject 2 is B+, GP 3.5
Your GPA for 2 subjects is 3.75 under SystemP
```

Write another grade point conversion function to compute the GPA for some institutes with this very confusing **Poor & Standard** mapping. A China-related subject with AA grade is mapped to a grade point of 4.0; a US-related subject with AA+ grade is mapped to 4.3; a Hong Kong-related subject with AAA grade is mapped to 4.5. Note that the grades BB or below are considered as “rubbish” under that rating scale.

AAA	AA+	AA	AA-	A+	A	A-	BBB+	BBB	BBB-	BB	B	C	D
4.5	4.3	4.0	3.7	3.3	3.0	2.7	2.3	2.0	1.7	1.5	1.0	0.5	0.0

#### Requirement:

1. Your program should compute the GPA for PolyU under the PolyU system, call it **SystemP**. PolyU GPA is *capped at 4.0*.
2. Your program should compute the GPA for the Poor & Standard institutes using the conversion table above, call it **SystemS**. The GPA is also *capped at 4.0*.
3. Grades are separated by spaces. Invalid grades should be *detected* and *discarded* from GPA calculation. For example, grade **C++** is considered to be invalid, so is **E** grade in both systems. Grade **A-** is invalid for **SystemP** and grade **D+** is invalid for **SystemS**. Grades are *case insensitive*, i.e. grade **a+** is considered to be equivalent to **A+** and is valid under **SystemP**.
4. There are *two functions*, one for each of the two university GPA systems.
5. Your program should compute the GPA for the two different systems, to *two decimal places*. Your program will recognize **-s** as the first parameter to indicate for the system under **SystemS**, by making the appropriate function call inside the main program.

For example, running

```
GPA A+ B C
```

will produce a GPA of **3.17** in **SystemP**, while running

```
GPA -s A+ B C
```

will produce a result GPA of **1.60** in **SystemS**.

Running

```
GPA Aa B a D
```

will produce a GPA of **2.67** in **SystemP**, with **AA** detected and discarded. Note this is valid in **SystemS**.

6. Complete your GPA program and **submit it via BlackBoard on or before 30 January**. Before submission, make sure that your program can *compile* with `gcc` and can *run* properly either under Mac or under **apollo2**. If your program is compiled and run under Mac, name your program **MacGPA.c** for submission. If your program is compiled and run under **apollo2**, name it **LinuxGPA.c** for submission.