CS2106: Operating Systems
Lab 1 – Advanced C Programming

Important:

- **The deadline of submission through IVLE is 7th September, 5pm**
- The total weightage is 5%:
    - Exercise 1: 1 % **[Lab demo exercise]**
    - Exercise 2: 2 %
    - Exercise 3: 2 %
- **You must ensure the exercises work properly on the lab machine (i.e. Linux on x86)**

Section 1. General Information

Here are some simple guidelines that will come in handy for all future labs.

1.1. Lab Assignment Duration & Lab Demonstration

Each lab assignment spans **two weeks** and consists of multiple exercises. One of the exercises is chosen to be the "lab demo exercise" which you need to demonstrate to your lab TA. For example, in this lab, you will need to demo exercise 1. The demonstration serves as a way to "kick start" your effort as well as a way to mark your **lab attendance**. You are **strongly encouraged to** finish the demo exercise before coming to the lab.

The remaining lab exercises are usually quite intensive, please do not expect to finish the exercise during the allocated lab session. **The main purpose of the lab session is to clarify doubts with the lab TAs and ensure your exercises work properly under Linux.**

1.2. Setting up the exercise(s)

For every lab, we will release two files in IVLE "Labs" workbin:

- **`labX.pdf`**: A document to describe the lab question, including the specification and the expected output for all the exercises.
- **`labX.tar.gz`** : An archive for setting up the directories and skeleton files given for the lab.

For unpacking the archive:

1. Copy the archive **`labX.tar.gz`** into your account.
2. Enter the following command:
   ```
   gunzip -c labX.tar.gz | tar xvf –
   ```
   Remember to replace the ***X*** with the actual lab number.
3. The above command should setup the files in the following structure:

```
LX/                 topmost directory for lab X
   ex1/             subdirectory for exercise 1
         ex1.c            skeleton file for exercise 1
         testY.in   sample test inputs, Y= 1, 2, 3, ...
         testZ.out  sample outputs, Z = 1, 2, 3, …
   ex2/
         ...        Similar to ex1
   ex3/             Similar to ex1
   ...
```

1.3. Testing using the sample test cases

For most of the exercises, a number of sample input/output are given. The sample input/output are usually simple text file, which you can view them using any editor or pager program.

You can opt to manually type in the sample input and check the output with the standard answer. However, a more efficient and less tedious way is to make use of **redirection** in Unix.

Let us assume that you have produced the executable **`answer.exe`** for a particular exercise. You can make use of the sample test case with the input redirection:

```
answer.exe < test1.in
```

The above "tricks" the executable **`answer.exe`** to read from **`test1.in`** **as if it was the keyboard input**. The output is shown on the screen, which you can manually check with **`test1.out`**.

Similarly, make use of output redirection to store the output in a file to facilitate comparison.

```
answer.exe < test1.in > myOut1.txt
```

The effect of the above command is:

- Take **`test1.in`** as if it is the standard input device
- Store all output to **`myOut1.txt`** as if it is the standard output device
  - Obviously, any other filename can be used
  - Just be careful not to overwrite an existing file

With the output store in **`myOut1.txt`**, you can utilize the **`diff`** command in unix to compare two files easily:

```
        diff test1.out myOut1.txt
```

which compares the output produced by your program with the sample output. Do a "**`man diff`**" to understand more about the result you see on screen.

You are **strong encouraged** to check the output in this fashion. By making sure your answer follow the standard answer (especially the format), you free up more time for the Lab TA to give better comments and feedback on your program.

Note that we will use **additional test cases** during marking to further stress test your submission. It is also part of your training to come up with "killer tests" for your own code. ☺

Section 2. Exercises in Lab 1

There are **three exercises** in this lab. Although the main motivation for this lab is to familiarize you with some advanced aspects of C programming, the **techniques** used in these exercises are quite commonly used in OS related topics.

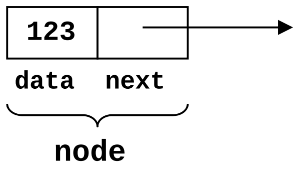
The first two exercises focus on **linked list**. The linked list is a “simple” yet powerful data structure that allows elements of a list to be stored in non-consecutive memory locations (as opposed to array). Operating System frequently make use of variations of linked list to keep track of important information, e.g. the process list, the free memory lists, file representation etc.

The last exercise is on **function pointer**. Unlike normal pointer, which points to memory location for **data storage,** a function pointer **points to a piece of code (function)**. By dereferencing a function pointer, we **invoke the function** that is referred by that pointer. This technique is commonly used in **system call / interrupt handlers**.

2.1 Exercise 1 [Lab Demo Exercise]

Linked list in C is based on pointers of structure. In ex1, the structure is as follows:

```
typedef struct NODE{
    int data;
    struct NODE* next;
} node;
```



A linked list is basically a series of node structure hooked up by the next pointer.

This exercise requires you to write **two functions**:

`node* addToHead(node* list, int newValue);`
This function takes an existing linked list `list` and an integer value `newValue`. The function will: o Make a new `node` to store `newValue` o Make this new `node` to be the **first node** (head) of `list` o Return the modified list

`void destroyList(node* list);`
This function takes an existing linked list **`list`** and **destroy every node** in the linked list by return the memory to system.

The program should:

a. Read a new integer value
b. Insert this integer value to the head position of linked list
c. Go to step a, until user terminates input by pressing **`Ctrl-D`**
    - **`Ctrl-D`** is the "end-of-file" signal
d. Print out the whole list
e. Destroy the list
f. Print out the list (should be empty!)

The skeleton file **`ex1.c`** has the following:

- The input/output code is already written.
- A useful function **`printList()`** is written to print out a correctly constructed linked list.

Sample Input:

```
1               //List =  1
2               //List = 2→1
3               //List = 3→2→1
4               //List = 4→3→2→1
[Ctrl-D]
```

Sample Output:

```
My List:
4 3 2 1
My List After Destroy:
                              //Should be empty
```

Important Note:

The correctness of your program cannot be entirely verified just by the output. For example, the linked list can be freed in the wrong way (*hint!*) but still seems to work. So, as a programmer, you need to scrutinize the code instead of just relying on the output.

This marks an important progression of your skill and understanding ☺. Of course, it is also a source of major pain for you as there is now no "simple way" to ensure your code is 100% correct. Just a friendly warning: most, if not all, lab exercises in CS2106 share this characteristic.

2.2 Exercise 2

This exercise is the same as exercise 1 except the addition to linked list can be **at any position**. To indicate a position, an **integer index is used**. The index is similar to those in array, i.e. first position = 0, second position = 1, Nth = N-1. The new value is to be inserted "before" the supplied index. If the index supplied is >= N (when the linked list has N nodes), then the new value is added at the end of linked list.

In addition, user can insert **multiple copies** of the same value in one call. You can assume the number of copy is ≥ 1.

You need to write at least **two functions** for this exercise:

```
node* insertAt(node* list, int position, int copies,
               int newValue);
```

Insert **`copies`** number of the integer **`newValue`** and **before** the index **`position`** in the linked list **`list`**.

```
void destroyList(node* list);
```

This function takes an existing linked list **`list`** and **destroy every node** in the linked list by returning the memory to system. You can use the same implementation as in ex1.

You are allowed to write additional functions if needed.

The program should:

a. Read three integer values:
   - The position, number of copy and the new value
b. Insert this integer value accordingly
c. Go to step a, until user terminates input by pressing **`Ctrl-D`**
   - **`Ctrl-D`** is the "end-of-file" signal
d. Print out the whole list
e. Destroy the list
f. Print out the list (should be empty!)

Sample Input:

```
0 33 1      //insert one 33 before index 0. List = 33
0 11 2      //insert two 11 before index 0. List = 11→11→33
1 22 1      //insert one 22 before index 1. List = 11→22→11→33
5 44 2      //insert two 44 before index 5. List = 11→22→11→33→44→44
[Ctrl-D]
```

Sample Output:

```
My List:
11 22 11 33 44 44
My List After Destroy:
                                   //Should be empty
```

2.3 Exercise 3

Suppose we have two functions:

```
void f( int x );

void g( int y );
```

They are considered as the same "type" of functions because **the number and datatype of parameters, as well as the return type is the same**. (More accurately, we say the **function signature** of the two functions are the same).

In C, it is possible to define a **function pointer** to refer to a function. For example:

```
void (*fptr) ( int );
```

To understand this declaration, imagine if you replace `(*fptr)` as `F`, then you have:

```
void F( int );
```

So, `F` is "a function that takes an integer as input, and return nothing (void)".

Now, since `(*fptr)` is `F`, `fptr` is "**a pointer to** a function that takes an integer as input, and return nothing (void)". Simple eh? ☺

Try your understanding of the following declarations:

Declaration	Meaning
`int (*fp) ();`	`fp` is a pointer to a function that takes no parameter and returns integer.
`int (*fp) (int, double);`	`fp` is a pointer to a function that takes integer and double values and returns integer.
`int* (*fp) ( );`	`fp` is a pointer to a function that takes no parameter and returns integer pointer.

Just like a normal pointer, you need to "point" it to a correct location before you can perform dereferencing later. Using the ***`f()`*** and ***`g()`*** functions at the beginning of this section, we can:

```
fptr = f;            //fptr points to function f
```

OR

```
fptr = g;            //fptr points to function g
```

Both of the above assignments are valid, as `fptr` must point to a function that takes an integer as input, and return nothing. Both `f()` and `g()` fit the type restriction.

We can now dereference **`fptr`**. When a function pointer is dereferenced, the function it is pointing to get invoked (called). E.g.

```
fptr = f;
(*fptr) ( 3 );                //Exactly the same as f( 3 );
```

OR

```
fptr = g;
(*fptr) ( 3 );                //Exactly the same as g( 3 );
```

If you look closely, you can see that we can invoke either **`f()`** or **`g()`** just by changing the pointer as the dereferencing statement is the same **"`(*fptr) (3);`"**

The function pointer can be applied in various interesting problems. However, you must agree that declaring a function pointer is quite troublesome. For example, to have 3 function pointers like **`fptr`**, we need to write:

```
void (*fp1) ( int );
void (*fp2) ( int );
void (*fp3) ( int );
```

To simplify the declaration, we can use **`typedef`** as follows:

```
typedef void (*funcPtrType) ( int );
```

The above creates a **new datatype** named "**`funcPtrType`**", which is a **function pointer that points to a function.**

With the help of this new datatype, we can now declare function pointer variables easily:

```
funcPtrType fp1, fp2, fp3;          //Quite an improvement!
```

Remember that the above is just a shortcut for the declaration, it **does not** affect the usage/meaning of the function pointer in any way. E.g.

```
fp1 = f;                //points fp1 to f()
(*fp1) (123);           //Same as f(123);
```

Now we are ready to tackle exercise 3. The idea of this lab is quite simple, we are going to simulate the spell repertoire of the famous wizard H. Potter. Each of his spell requires 3 integers: <Spell Number> <X> <Y> and produce two effects: <Name of the spell> and <Z>, where <Z> is the result of a mythical (I mean ***Math***ematical) combination of <X> and <Y>.

Below is the summary of the spells to be simulated:

Spell Number	Spell Name	Result
1	*lumos*	X is guaranteed to be smaller or equal to Y Z = X + (X+1) + (X+2) + …. + Y
2	*alohomora*	Z = GCD(X, Y)
3	*expelliarmus*	If X can be expressed as Y^n*Q, then Z = Q i.e. remove factor Y from the number X, such that Z no longer has Y as a factor.
4	*sonorus*	$Z = X^Y$

Your program should print out the **<Spell Name>** and the **<Z>** based on the user supplied <Spell Number> <X> <Y>. The program will continue to receive input until the user press Ctrl-D (a.k.a. end-of-file signal) to terminate.

Now the fun part: Your code **must conform to the following restrictions:**

- You are **not allowed to use selection statement nor repetition statement**, i.e no "**`if-else`**", "**`switch-case`**", "**`while`**", "**`for`**" etc **to determine which spell to cast**.
- Each "spell" must be self-contained as a function. Printing (i.e. for the spell name) should **not** be performed in the function.

Hint:

- You'll need function pointers (duh!) and **array**……
- You can assume that the **input are always valid** and will not result in any overflow / underflow in calculation, i.e. you do not need to validate input.

Sample Input:

```
2 13 7                  // Spell 2, GCD(13, 7)
1 1 100                 // Spell 1, 1 + 2 + 3 + ….. + 100
3 2835 3                // Spell 3, removes factor 3 from 2835
2 30 105                // Spell 2, GCD(30, 105)
4 3 5                   // Spell 4, 3^5
1 20 20                 // Spell 1, 20
[Ctrl-D]
```

Sample Output:

```
alohomora 1       //Output format:  Spell_Name<space>Z<newline>
lumos 5050
expelliarmus 35
alohomora 15
sonorous 243
lumos 20
```

Note that the skeleton file **`ex3.c`** contains demo code to show you the idea behind function pointer. You need to replace the code for the actual exercise.

For your pondering:

Function pointer, though seems like a weird C language quirks, is actually a very powerful programming technique. Consider the following:

- How can we quickly change the mapping of the functions (say 1=sonorus, 2=lumos, etc in this exercise)?
- How can we easily add a new spell without disturbing the current code structure?
- How can we modify the functionalities without affecting other part of the code (say lumos calculates the prime factorization now instead)?
- Is the function pointer approach faster / easier to write than selection statement (think about when you have many more options, say 100+)?

This mechanism is available in many other programming languages, albeit with a different name, e.g. JavaScript allows you to bind function to a variable, functional programming languages, e.g. Python allows you to pass function around as a "value" (more formally known as **function as first class citizen**), etc.

Section 3. Submission through IVLE

Zip the following files as E0123456.zip (**use your student id and NOT matric number, use capital 'E' as prefix**):

a. **`ex2.c`**
b. **`ex3.c`**

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "**`lab1\`**" subfolder etc.

Upload the zip file to the "Student Submission→Lab 1" workbin folder on IVLE. Note the deadline for the submission is **7th September, 5pm**.

Please ensure you follow the instructions carefully (output format, how to zip the files etc). Deviations will cause unnecessary delays in the marking of your submission.

**~~~ Have Fun! ~~~**