CS2106: Operating Systems
Lab 1 – Advanced C Programming

Important:

- **Read "Introduction to OS Lab" document before this.**
- **The deadline of submission is 5th February, 2359**
- The total weightage is 5%:
  - Exercise 1: 1 % **[Lab demo exercise]**
  - Exercise 2: 1.5 %
  - Exercise 3: 1.5 %
  - Exercise 4: 1 %
  -

Section 1. Lab Setup

Please read through the "Introduction to OS Labs" document **first**. Follow section 3 in that document to unpack the lab archive accordingly.

1.1.Group Submission Setup

If you have decided to tackle CS2106 with a buddy. Please use the following google form to register your team:

https://forms.gle/2XGm3zFYpXdFTgpc6

Select your name and your buddy's name from the dropdown list to form a team. Your team will be in effect from this lab onward.

Section 2. Exercises in Lab 1

There are **four 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 third 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

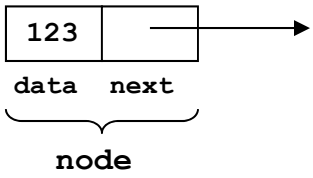
referred by that pointer. This technique is commonly used in **system call / interrupt handlers**.

The last exercise introduces a useful tool to catch memory bug in program. This can prove to be very useful to debug program with frequent memory allocation / free (like exercise 1 and 2!).

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: ○ Make a new **`node`** to store **`newValue`** ○ Make this new **`node`** to be the **first node** (head) of **`list`** ○ 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. In addition, exercise 4 introduces a useful tool for you to catch memory bugs.

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, N^{th} = 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, the new value and number of copy
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**. [Clarification: You can of course use a loop for the main input loop. Selection statement / repetition is banned for the spell casting part only.]
- Each "spell" must be self-contained as a function. Printing (i.e. for the spell name) should **not** be performed in the function.

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
sonorus 243
lumos 20
```

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.

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.

2.3 Exercise 4 – Checking for Memory Error

Catching memory bugs (e.g. memory leak, out of bound error, etc) is probably the top nightmare for programmers. Fortunately, there are tools nowadays to somewhat mitigate the pain.

For this exercise, we introduces one such tools, **valgrind**. This tool works by "attaching" itself to an executable and observe the memory usage throughout the execution of that executable. Valgrind will then give you a report at the end of the execution.

We have provided a quirky program **`ex4.c`** in the **`ex4`** folder. Compiles the program using the standard command:

```
$ gcc -std=c99 -Wall -Wextra ex4.c -o ex4
```

This program takes in two command line arguments **`N`** and **`M`** (with the assumption that **`M`** <= **`N`**). During execution, the program will attempt to allocate **`N`** integers, and only free up M of them at the end. (i.e. you can easily create a memory leak example when **`M`** < **`N`**).

Here's a sample output from valgrind with **`N`** = 10, **`M`** = 7 by using the command:

```
$ valgrind ./ex4 10 7
```

```
==3247239== Memcheck, a memory error detector
==3247239== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3247239== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3247239== Command: ./ex4 10 7
==3247239==
==3247239==
==3247239== HEAP SUMMARY:
==3247239==     in use at exit: 12 bytes in 3 blocks
==3247239==   total heap usage: 11 allocs, 8 frees, 120 bytes allocated
==3247239==
==3247239== LEAK SUMMARY:
==3247239==    definitely lost: 12 bytes in 3 blocks
==3247239==    indirectly lost: 0 bytes in 0 blocks
==3247239==      possibly lost: 0 bytes in 0 blocks
==3247239==    still reachable: 0 bytes in 0 blocks
==3247239==         suppressed: 0 bytes in 0 blocks
==3247239== Rerun with --leak-check=full to see details of leaked memory
==3247239==
==3247239== For lists of detected and suppressed errors, rerun with: -s
==3247239== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

From the highlighted portion, you can see the value of such tools! **`Valgrind`** is able to capture the number of memory allocation (**`malloc`**() in our case), the number of de-allocation (**`free`**()). More importantly, it can tell you about memory leaks (we allocated 10 integers but only free 7 of them, so we lost 3 x 4 bytes = 12 bytes!).

Here's another with **N** = **M** = 10 **$ valgrind ./ex4 10 10**

```
==3247567== Memcheck, a memory error detector
==3247567== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3247567== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3247567== Command: ./ex4 10 10
==3247567==
==3247567==
==3247567== HEAP SUMMARY:
==3247567==     in use at exit: 0 bytes in 0 blocks
==3247567==   total heap usage: 11 allocs, 11 frees, 120 bytes allocated
==3247567==
==3247567== All heap blocks were freed -- no leaks are possible
==3247567==
==3247567== For lists of detected and suppressed errors, rerun with: -s
==3247567== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Try different values of **N** and **M** to verify your understanding of **valgrind**'s capability.

Now, use **valgrind** to ensure your **ex2** is free from memory leak errors. i.e., this exercise essentially give 1 **bonus point** for ex2 if they are free from memory bugs.

Section 3. Submission Procedure

Zip the following folders and files as A0123456X.zip (**use your student id with "A" prefix and the alphabet suffix).** One simple way is to perform the following command **at the parent folders of the ex?/** folders (i.e. at the **L1/** folder):

```
zip A0123456X.zip ex2/ex2.c ex3/ex3.c
```

Remember to modify the archive name and the content accordingly as needed.

A0123456X.zip should contains 2 folders with the following content:

ex2/
    ex2.c
ex3/
    ex3.c

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

3.1 Do a quick self checking

We have provided a self checking shell script to help validating your zip archive. The script checks the following:

a. The name or the archive you provide matches the naming convention mentioned above.
b. Your zip file can be unarchived, and the folder structure follows the structure described above.
c. All files for each exercise with the required names are present.
d. Each exercise can be compiled and/or executed.
e. The output your exercise produces using our sample input matches the expected output.

Once you have produce the zip file, you will be able to check it by performing the following steps at the **L1/** folder:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456X.zip (replace with your zip file name)
```

The check_zip script will print out the status of each checks. Successfully passing checks enable the lab TA to focus on grading your assignment instead of performing lots of clerical tasks (checking filename, folder structure etc). Please note that **points might be deducted if you fail these checks.**

Expected Successful Output
Checking zip file.... Unzipping file: A0123456X.zip Transferring necessary skeleton files ex2: Success ex3: Success

3.2 Upload to Luminus before deadline!

After verifying your zip archive, please upload the zip file to the "Student Submission→Lab 1" file folder on Luminus. Please note that **late penalty is quite steep,** so submit early to avoid last minute issues.

Also, remember that if you are in a team, **only one member need to submit into Luminus. The archive name can be any of the team members, we will do "background processing" to match with your team members automatically. Both team members will receive the same score.**

~~~ Have Fun! ~~~