Lab 05 – Dynamic Memory

This lab explores the C language dynamic memory functions. Throughout this lab, you will review some programs in C as well as compile, test, debug and fix them.

By the end of this lab, you will be able to:

- Use the C language to express code solutions to simple problem specifications;
- Use C constructs to allocate and deallocate dynamic memory;
- Test, debug and fix source code in C.

Copy the lab files

Create a Lab5 directory, go to it, and copy over the files from the source, which are located in the lab environment in the **`/home/rbittencourt/seng265/lab-05/`** directory.

example.c

This program attempts to reserve enough dynamic memory for 10 doubles, and then prints and modifies their values. It currently contains several errors and omissions. As you correct these issues, keep in mind the following hints:

• We can obtain a pointer to a region of heap memory with the **`malloc()`** function.

• **`malloc()`** needs to be told the number of bytes to reserve.

• we can use the **`sizeof()`** operator to figure out the size of a type. For example, **`sizeof(int)`** will report that an **`int`** needs 4 bytes of memory.

• **`malloc()`** does not initialize the reserved memory – if you want to set it to zero, used **`calloc()`** instead.

• On success, **`malloc()`** returns a pointer to the first location in the reserved memory.

• On failure, **`malloc()`** will return a null pointer.

• If we get a pointer to dynamic memory, we must use **`free()`** on that pointer before the end of the program.

intblock_start.c

This program takes two numbers as command line arguments. The first is the 'number of integers to print' and the second is a seed for the random number generator. Right now the program simply generates a series of random numbers and prints them.

You need to modify the program by replacing the current for-loop with code that places random numbers into an array. Use dynamic memory to create the array. Once the array is full, use **`qsort()`** to sort the array, and then print out the resulting array.

Recall the signature for **`qsort()`**:

```
qsort(array_name, number_of_elements, size_of_each_element,
sorting_function_name)
```

A sorting function named compare_ints has been provided for you; you don't need to modify it.

Remember to free memory once you're done with it.

sorty_start.c

A file named **`words.txt`** has been provided - it contains several hundred words, one per line. Our goal is to print out the words in this file, in sorted order.

• Note that we have some global variables already defined for us.

• **`getline()`** is a useful function that allows us to read a line of input. Recall its syntax: **`getline(&buffer, &buffer_len, infile)`** will put a line from **`infile`** into **`buffer`**, its default allocated line size into **`buffer_len`**, and will return the string length of the line just read (or will return **`-1`** if we have run out of input).

• **`getline()`** isn't available by default, since it's a GNU extension to C11's **`stdio.h`**. To use it, we need to add a **`#DEFINE`** directive to our file prior to where we include **`stdio.h`**:

```
#define _GNU_SOURCE
```

• Remember that **`getline()`** gives us a string with the newline at the end included. We probably don't want that - overwriting the newline with a null terminator will solve this problem.

• Alternatively, you can use **`fgets()`** to read a string from the file.

• We need dynamic memory! Our **`lines[]`** array just holds pointers to strings - it doesn't hold the strings themselves. Therefore we can't just have it point to **`buffer`**, since buffer will change every time we call **`getline`**. We'll need to create a chunk of memory, copy the buffer into it, and then store a pointer to that memory in **`lines[]`**.

• Our **`compare()`** function is all ready for **`qsort()`**; we don't need to modify it at all.

Present your results to your TA

Take a few minutes to present the results of your code changes and creation to your TA. That is how you will be assessed in this lab.