Course: DD2325 - Exercise Set 5

Exercise 1: *Basic Pointers*

Why do pointers exist?

Exercise 2: *Local Memory I*

Variables represent storage space in the computer's memory. Behind the scenes at runtime, each variable uses an area of the computer's memory to store its value. It is not the case that every variable in a program has a permanently assigned area of memory. Instead, modern languages are smart about giving memory to a variable only when necessary. The terminology is that a variable is allocated when it is given an area of memory to store its value. While the variable is allocated, it can operate as a variable in the usual way to hold a value. A variable is deallocated when the system reclaims the memory from the variable, so it no longer has an area to store its value. For a variable, the period of time from its allocation until its deallocation is called its **lifetime**.

Local Memory The most common variables you use are *local* variables within functions. All of the local variables and parameters taken together are called its *local storage* or just its *locals*. The variables are called *local* to capture the idea that their lifetime is tied to the function where they are declared. Whenever the function runs, its local variables are allocated. When the function exits, its locals are deallocated.

Here is an example which shows how the simple rule *the locals are allocated when their function begins running and are deallocated when it exits.* Sketch local memory at the points requested.

```
void X() {
   int a = 1;
   int b = 2
   /* Sketch local memory at this point:  T1*/
   Y(a);
   /* Sketch local memory at this point:  T3*/
   Y(b);
   /* Sketch local memory at this point:  T5*/
}

void Y(int p) {
   int q;
```

```
   q = p + 2;
   /* Sketch local memory at this point (T2 (T4) 1st (2nd) time)*/
}
```

Exercise 3: *Local Memory II*

What is wrong with the following code where the function `Victim()` calls the function `TAB()` ?

```
int* TAB() {
   int temp;
   return(&temp);
}

void Victim() {
   int* ptr;
   ptr = TAB();
   *ptr = 42;
}
```

Remember: Local Memory
Locals are very convenient for what they do - providing convenient and efficient memory for a function which exists only so long as the function is executing. Locals have two deficiencies which we will address in the following sections - how a function can communicate back to its caller, and how a function can allocate separate memory with a less constrained lifetime.

Remember Reference Parameter Passing by value (copying) does not allow the callee to communicate back to its caller and has also has the usual disadvantages of making copies. Pass by reference uses pointers to avoid copying the value of interest, and allow the callee to communicate back to the caller.

For pass by reference, there is only one copy of the value of interest, and pointers to that one copy are passed. So if the value of interest is an `int`, its reference parameter is an `int*`. If the value of interest is a `struct fraction*`, its reference parameters is a `struct fraction**`. Functions use the dereference operator `(*)` on the reference parameter to see or change the value of interest.

Exercise 4: *Heap Memory*

Heap memory, aka *dynamic* memory, is an alternative to local stack memory. Local memory is automatic - it is allocated automatically on function call and it is deallocated automatically when a function exits. Heap memory is

different in every way. The programmer explicitly requests the allocation of a memory *block* of a particular size, and the block continues to be allocated until the programmer explicitly requests that it be deallocated, using `malloc`, `free` etc... Nothing happens automatically. So the programmer has much greater control of memory, but with greater responsibility since the memory must now be actively managed.

Advantages	**Disadvantages**
Lifetime programmer controlled	*More work*
Size programmer controlled	*More bugs*

Draw the state of memory at three different times during the execution of the following code.

```
void Heap1() {
   int* intPtr;
   intPtr = malloc(sizeof(int));
   *intPtr = 42;
   free(intPtr);
}
```

Exercise 5: *Queues*

Draw a graphical representation of a `queue` implemented using a linked list.

Exercise 6: *Queues*

Consider the following situation. There is a bus stop at the end of your road. The first bus of the morning arrives at this bus stop at 08:10. From then until 10:00 a bus arrives at the bus stop every 10 minutes. People start queueing at this bus stop from 08:00. Say that one person arrives at the bus stop and joins the queue every minute with probability .9. Each arriving bus will allow 5 to 15 people board, depending on how full it is. How can you use a `queue` structure to record the number of people in the queue at any time and their waiting times for a bus once they board a bus from a program simulating this process?

Note: a pseudo-random integer can be generated in `C` using the command `rand()` from `stdlib.h`. An integer between `0` and `N` can be generated with the following code:

```
(int)(rand() / (((double)RAND_MAX + 1)/(N+1)));
```

Exercise 7: *Trees 0*

On the board we will review some of the terminology associated with trees.

Exercise 8: *Trees I*

A binary tree is a tree in which every element has at most 2 children. Draw a graphical representation of a binary `tree`.

Exercise 9: *Trees II*

A binary tree is *full* if every node bar the leaves has 2 children. How many nodes does a full binary tree of height `h` have?