C Review

CS 2200

C programming – Things to watch out for

- It is easy to write bad C code. Don’t.
- Watch out for implicit casting.
- Watch out for memory leaks.
- Watch out for undefined behavior.
- Macros and Pointer arithmetic can be tricky
- Kernighan Ritchie C programming Language is your best friend

Pointers

- What are **pointers** in C? – A variable that **holds the address** of another variable is a pointer.
- How do you instantiate a pointer? – The type followed by a '*'. For example:

```
int * ptr;          // ptr is a pointer to an integer data type
```

- Why are pointers useful? – They allow you to change a value of some variable inside a function; Allows you to link data structures together, aka linked list; Allows you to work with data allocated on the stack
- The address of a variable is taken using the '**&**' operator. What is the meaning of dereferencing a pointer – Reading the value of the variable that the pointer is pointing to. You dereference by using the '*' operator before the variable name:

```
int a = 5; ptr = &a; printf("%d", *ptr); // Will print value of a
```

- Keep in mind that a pointer to type 'a' references a block of sizeof(a) bytes.

Pointer Arithmetic

- You can add/subtract from an address to get a new address:
  - Results depend on pointer type
  - Do arithmetic only when necessary (i.e. with malloced memory)
- Let us say that A is a pointer = 0x1000, i is an int on a 64 bit machine
  - `int * A` : A + i = 0x1000 + sizeof(int) * i = 0x1000 + 4 * i
  - `char *A` : A + i = 0x1000 + sizeof(char) * i = 0x1000 + i
  - `int **A`: A + i = 0x1000 + sizeof(int *) * i = 0x1000 + 8 * i
- Rule of thumb, cast pointers explicitly to avoid confusion.
  - This means that do `(int*) A + i` vs `A + i`

Structs

- Collection of values placed under one name in a single block of memory
  - Can have arrays of structs or structs of arrays
- Given the struct instance, you can access the fields using the '.' operator
- Given a pointer to a struct, you can access values using the '->' operator

```
struct foo_t {
  int a;
  char b;
}
```

```
struct foo_t foo;
      foo.a = 100;
struct foo_t *ptr = &foo;
            ptr->b = 'a'; // Access using a pointer
```

Dynamic Memory Allocation

- `void * malloc (size_t size):`
  - Allocated a block of memory of size number of bytes
  - Memory is not initialized (unlike calloc which initializes memory to 0)
- `Void free(void* ptr):`
  - Frees the memory block that had been previously allocated using malloc, calloc or realloc and is pointed to by ptr.
  - Note: Use only once per allocated block
- The `size` argument of malloc should be computed using the `sizeof` operator. `sizeof()` is not a function, it is an operator.
- The **number of mallocs should be equal to number of frees** to ensure that there are no memory leaks nor double free errors.

Identify the Problem with this code

```
int foo(unsigned int u) {
        return (u > -1) ? 1 : 0;
}
```

```
int foo(unsigned int u) {
        return (u > -1) ? 1 : 0;
}

// Implicit casting of an signed number to unsigned. So what is
// happening is u > int_max (-1 == 0xFFFFFFFF), and this will always
//  return 0;
// Warning: Don't rely on implicit casting - Always add explicit cast
```

Identify the Problem with the below code:

```
int main() {
        int* a = malloc(100*sizeof(int));
        for (int i=0; i<100; i++) {
                a[i] = i / a[i];
        }
        free(a);
        return 0;
}
```

Using variable before assigning a value to it:

```
int main() {
        int* a = malloc(100*sizeof(int));
        for (int i=0; i<100; i++) {
                a[i] = i / a[i];
        }
        free(a);
        return 0;
}
// Using the value of a[i] before initializing it. Undefined program
// behavior
```

How to debug Segmentation Faults

- Use GDB!! - Micro Project 2 is designed to make you get a feel of GDB, and to explore its functionality.
- Show a demo of gdb using some piece of code.