**C Programming**

Basics Part 2

**Pointers**

A pointer is like a type of variable. It is a variable which stores an address in memory. Remember, C is a low-level language so you need to be careful when using pointers

Declaration

Int \*pointerInt;

Char \*letter;

Double \*price;

Int size = 15;

Int \*ptr;

Int = &size; // This stores the location in memory of the *size* variable

Printf(“address %x points to %d”, ptr, \*ptr);

// ptr = hex address, \*ptr = value associated with memory

Remember, all memory in the computer is labelled by hex numbers or binary. When you store a value in a variable, you are picking one of those hexadecimal labels of memory locations and storing some data to it. A pointer stores the location of that memory, not the value.

**Performing pointer arithmetic**

Int sizes[] = {15, 20, 30};

Int \*ptr = sizes; // pointer points to the start of the array

Ptr++; // points to the next value address of the array which could be a completely different address to the first

Ptr--; // points back to index 0

**Understanding relationships between pointers & array**

Arrays store data in a sequential block. Array notation is essentially pointer notation.

Pointers enable powerful access to memory.

Int main(void) {

Int data[] = {15, 20, 30, 45, 60, 80};

Int \*ptr = data;

Printf(“data[] starts at address %x with value %d\n”, ptr, \*ptr);

// calculate length of array

Int numElements = sizeof(data) / sizeof(int);

Printf(“There are %d elements in data\n”, numElements);

For (int I = 0; I < numElements; i++) {

Printf(“Address %x stores value %d\n”, ptr, \*ptr);

Ptr++;

}

// note, in for loop there is no use of the index of array

Return EXIT\_SUCCESS;

}

//sizeof() returns size of memory in bytes, an integer is 4 bytes, there are 6 elements in data array above, so 6(elements) \* 4(byte of 1 integer) = 24 bytes of total array. Divide by 4 to get the total length of array (for integers)

**Managing Memory using allocation and release**

Static arrays waste memory: int array[20]; guaranteed to allocate 20 integers. Whether you use 20 ints or not, that memory is set aside.

Dynamic arrays save memory by using a pointer:

Int \*array = malloc(20 \* sizeof(int)); // malloc = memory allocation

Dynamic arrays just point to the address, you don’t need to use it.

To release unused memory: *free(array);*

Int \*dynamicArray = malloc(20 \* sizeof(int));

\*dynamicArray = 10; // 0 index

\*dynamicArray[1] = 20; // 1 index

Printf(“The size of dynamic array: %d\n”, sizeof(dynamicArray));

// 8 – 2 elements \* 4. If this wasn’t a dynamic array, the size would be 80 bytes (20 \* 4)

Free(dynamicArray);

**Structures**

**Quiz**