# CS 2110 Lab 16
# Pointers and Arrays

The TAs :)

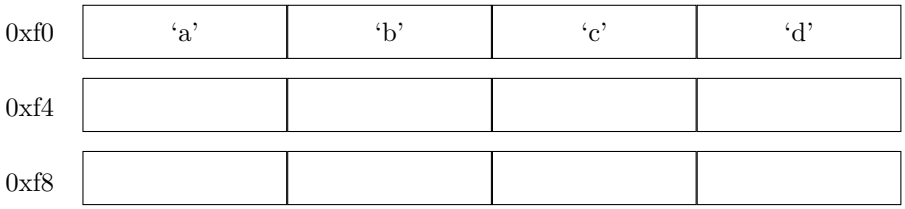
10/21/2018

## 1 Recitation Outline

1. Helpful Things to Know
2. Arrays
3. Pointers
4. Pointers and Structs
5. Pointers and Malloc

## 2 Useful Tidbits

Below is a typical picture of how memory is laid out. Each memory address represents one byte of memory, therefore the memory address 0xf0 points to memory location (a), address 0xf1 points to memory location (b), address 0xf2 points to memory location (c) etc.




For c, our basic data types are char, short, and int. A char is one byte, a short is generally two bytes, and an int is generally four bytes.

**Aside: Different architectures have different sizes for data types. Luckily, there is a function - sizeof() - that will give you the correct size of any data type on any architecture. When you are programming, DO NOT ASSUME SIZES, use sizeof instead. But, for the purpose of learning in this recitation, we will assume that a char is 1 byte, short is 2 bytes, and an int is 4 bytes. Please don't assume this later. :)**

# 3 Pointers

Pointers are variables that contain a memory address as well as the type of the data that's expected to be found at that address. A special case is a void pointer, which is a pointer that contains a memory address but not the type of the data found at the address (e.g. for use in cases where the type is not relevant).

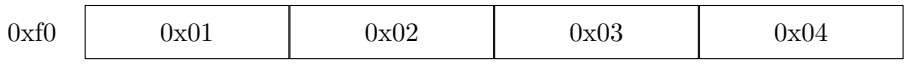
1. An * after a type denotes a pointer.
   Example:
   `char *x;` declares that x is a pointer to a char.
   `char **y;` declares that y is a pointer to a pointer to a char.

2. An & in front of something with a memory address (such as a variable or a function) gives its address, which can be stored as a pointer.
   Example:

   ```
   char i = 1;      // i stores the value 1
   char *x = &i;    // x stores the address of the variable i
   ```

   This block of code denotes that x is a char pointer, and holds the memory address of i. It may be helpful to remember the & operator as the "address of" operator.

3. Finally, a * in front of an expression with a pointer type dereferences that pointer, or gets the value at that location in memory. In this context, it is helpful to remember the * as the "value at" operator, since it will give you the value at a memory address (read: pointer).




```
char i = 1;      // i stores the value 1, and we store this value at 0xf0
char *x = &i;    // x stores the "address of" i, which is 0xf0
char y = *x;     // y stores the "value at" the "address of" i (read: x)
```

   As with the picture above, i equals the character '1'. Since x is a pointer to i, or holds the address of i, the value at x would be the memory address 0xf0. Finally, y is the dereferenced value of the memory address x. This means that y will equal the value at the memory address x = '1'.

### 3.0.1 A Helpful(?) Mnemonic

Why is the syntax for pointers so weird in C? Because the designers wanted you to declare pointers the same way you use them.
For example, to get a char out of a char pointer `cp`, you do `*cp`. Consequently, you declare `cp` as

```
char *cp;
```

Now suppose we have `cpa`, which is an array of character pointers (similar to the argv parameter in your main function). To get a char out of cpa, we need to do `*cpa[i]` for some index $i$. So we declare `cpa` as (if we want it to hold 420 character pointers)

```
char *cpa[420];
```

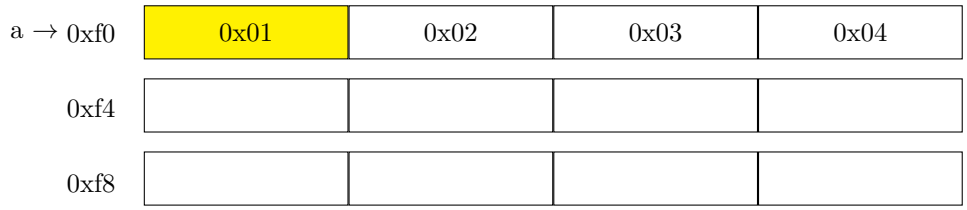
For a triple int pointer, this still works, since you'd write `***holy_moly` to get an int out of it, and the declaration looks like

```
int ***holy_moly;
```

When you get into more hairy pointer territory, such as function pointers (which you'll learn about later) this mnemonic is lit

## 3.1 Pointer Arithmetic

Like arrays, pointers can also be used to grab values at particular memory locations.

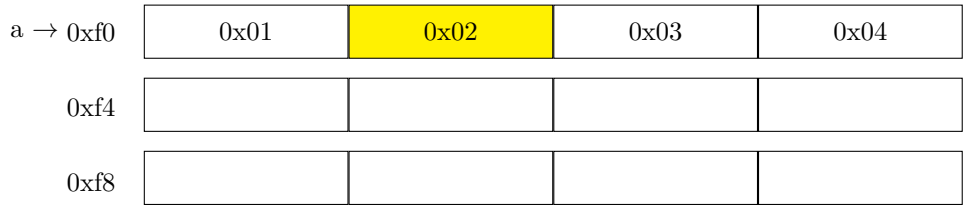



In this example, a is a char pointer and holds the memory address 0xf0. When dereferenced, a char — one byte — will be grabbed.

```
char x = *a; // x = 1
```

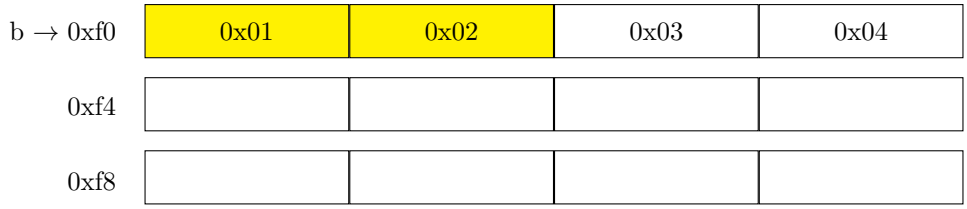
Therefore, x will be equal to 1. Now, if 1 is added to the char pointer, a would change from 0xf0 to 0xf1, and when dereferenced, x will equal 2.

```
char x = *(a + 1); // x = 2
```



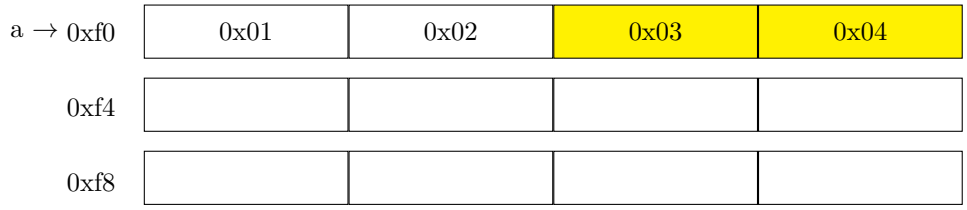

What if b is a short pointer? This means that b points to two bytes, not just one. When a short pointer gets dereferenced, two bytes will be grabbed from memory, not just one.

```
short xyz = *b; // gives 0x0102 = 258
```




Therefore xyz = 0x0102 = 258. If we add one to the memory address b, we add the size of one short. So if we do `b=b+1`, then b will now hold the address 0xf2 and when dereferenced, xyz will equal 0x0304 = 772.

```
short xyz = *(b+1); // gives 0x0304 = 772
```




This process of multiplying the offset by the size first is performed by the compiler at the compilation step using the type of the pointer:

```
0xf0 + (2 * sizeof(short)) = 0xf4
```

Where `0xf0` is our starting address, and we want the second element. Be very careful with pointer arithmetic as this concept is extremely tricky!

### 3.1.1 The Bracket Operator

In C, the bracket operator `x[y]` is used as a shorthand notation for accessing memory items from a pointer. This operator is simply syntactic sugar for pointer arithmetic and dereferencing: `x[y] = *(x + y)`. Note that while this operation is commutative, i.e. `x[y] == y[x]`, it is customary to keep the pointer outside the brackets and the index inside. This operator is widely used to index into arrays.
For example, let:

```
short* myPointer;
```

To get the 2nd element from memory starting at the pointer, all of the following are equivalent:

```
short secondElemFromPointer = myPointer[1];
short secondElemFromPointer = *(myPointer + 1);
short secondElemFromPointer = *((short *)((char*)myPointer + 1 * sizeof(short)));
```

## 3.2 Pointers and Structs

You can have a pointer to a struct as well! Remember the general form of structs:

```
struct dog {
    int number;
    char letter;
}
```

When you want to access a member of a struct - you use the '.' notation:

```
struct dog labrador;
int myNumber = labrador.number
```

Pointers are very similar! You just need to dereference the pointer to the struct using the '*' notation, then access a member of a struct using the '.' notation.

```
struct dog *dogPointer = &labrador
int myNumber = (*dogPointer).number
```

Getting a member from a struct pointer is such a common thing to do that it has its own notation '$->$'

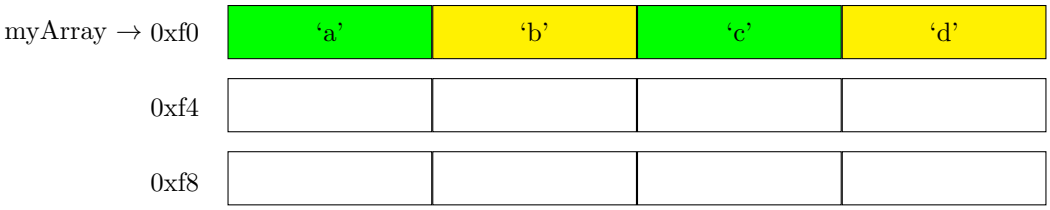
```
struct dog *dogPointer = &labrador
int myNumber = dogPointer -> number
```

# 4 Pointers and Arrays

Now that we know what pointers are, we can discuss arrays.
Arrays are contiguously stored sequences of items in memory. An array declaration allocates room for the array on the stack, then stores a pointer to the first element as the variable. From then on, the array variable can be used as a pointer.
An array in C has a fixed size. After the allocation is complete, the array variable can be used as a pointer to the first element in the array. As a result, pointer arithmetic and the box operator can be used to index into the array to get individual elements.
Note that the box operator for arrays is NOT special! It is the same stuff as the pointer arithmetic box operator and thus will not prevent you from indexing out of the bounds of the array.
Example:

```
char myArray[4];
```

This means that myArray will be an array containing four chars.



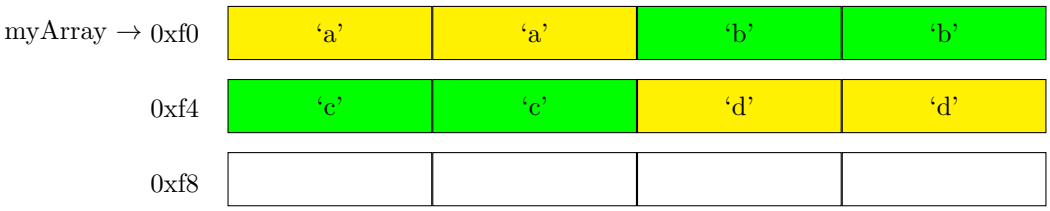

myArray is an array that begins at memory address `0xf0` and contains four elements (at memory addresses `0xf0`, `0xf1`, `0xf2`, `0xf3` respectively). Indexing into the array grabs the value at position x in the array. Example:

```
char xyz = myArray[2]; // xyz will now contain the value at position 2 in the array, which is 'c'
```

What about an array of shorts?

```
short myShortArray[4];
```

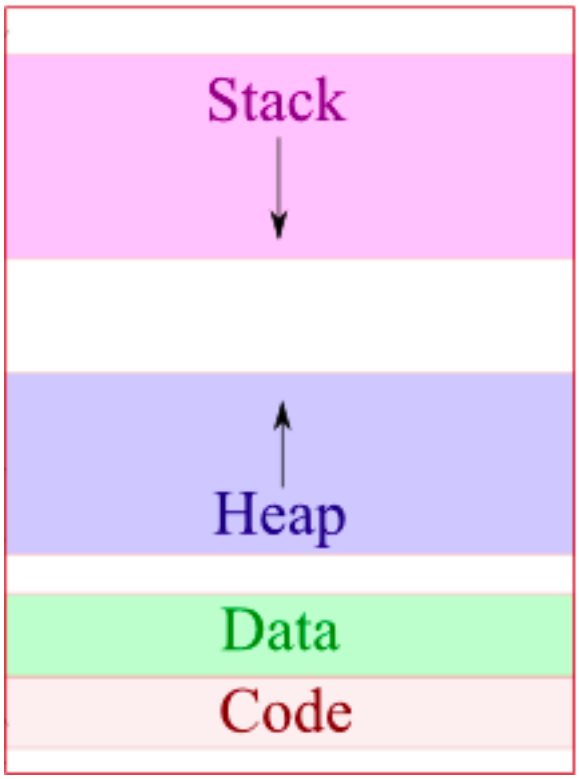
With an array of shorts, each position in the array has two bytes in memory:




Now, myShortArray is an array beginning at memory address `0xf0` and containing four shorts (at memory addresses `0xf0`, `0xf2`, `0xf4`, and `0xf6` respectively. Indexing into this array will return to the user a short, or two bytes:

```
short a = myShortArray[2];
short a = *(myShortArray + 2);
```

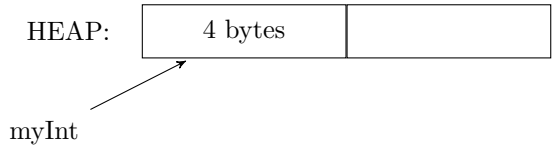
# 5 Pointers and Malloc




Now that we have discussed both pointers and arrays, where are they located in memory? Arrays can be located in the stack portion of memory (if it is a local variable) or the static portion of memory (if the array is a global variable). Pointers, on the other hand, are located in the stack or static portion of memory, but point to variables located in the stack, static, or heap portions of memory.

The heap is used for dynamic allocation of memory, and pieces of the heap can be allocated to the user using a function called `malloc()`. Malloc takes in an argument depicting how many bytes to allocate from the heap and returns a pointer to this area of memory. Example:


```
int* myInt = malloc(sizeof(int)); //Assume int is 4 bytes here
```




Now we have this area of memory to do things with! When we are done with this memory, we need to give it back to the heap so that it can be allocated in another place.

```
free(myInt);
```

Because free() tells C that it can give the memory to other code, which may use the memory for other purposes, make sure to free() a block only when you're *completely* done with it. And free() a given block only once per malloc(), or it might get given to two different people!