Lab Week #3
CIS 314

October 16, 2016

How to declare array

Array declarations in C is (nearly) identical to how it was done in Java:

```c
//Declare an array named 'array' of length 5 which will hold integers
int array[5];
```

Once we have declared an array, we can populate it (and subsequently print the values) by using some form of iteration:

```c
int array[5];

//To be compliant with gcc, we need to forward declare our index counter
int i;
for(i=0; i<5; i++){
  array[i] = i;
}

//This function will be moved later... otherwise we could just print the values
//  in the loop above!
for(i=0; i<5; i++){
  printf("array[%d] = %d\n", i, array[i]);
}
```

What is an array?

Recall back to arrays in Java – a common issue is that people will attempt to print an array by simply putting the array value into some output stream and expecting it show all values in the array. Unfortunately, this only printed the memory location of the beginning of the array. As it turns out, C is exactly the same:

```c
//Declare an array named 'array' of length 5 which will hold integers
int array[5];

//...
//This section populates the array as above
//...

printf("array = 0x%X\n", array);

//The above will print something like "array = 0x578A9D20"
```

Where it gets interesting is when we dereference the array:

```c
//Declare an array named 'array' of length 5 which will hold integers
int array[5];

//...
//This section populates the array as above
//...

printf("*array = %d\n", *array);

//The above will now print "*array = 0"
```

Conveniently, when we dereference the array we get the value that is held in array[0]! The next thing that may be of interest is what happens when we "increment" array:

```
//Declare an array named 'array' of length 5 which will hold integers
int array[5];

//...
//This section populates the array as above
//...

printf("*(array+1) = %d\n", *(array+1));
//The above will now print "*(array+1) = 1"

printf("array+1 = %d\n", array+1);
//The above will print something like "array = 0x578A9D24"
```

The first print statement *feels* like it makes sense – we're now getting the value of array[1] – but the second print statement actually says something deeper. As it turns out, when we increment the pointer `array` by 1, we end up increasing its value by 4.[1] After a bit of reflection, it should become apparent that there's a second way to access the slots of an array:

```
int array[5];

//...
//This section populates the array as above
//...

//This function is semantically equivalent to the above printing function -- neat!
for(i=0; i<5; i++){
  printf("array[%d] = %d\n", i, *(array + i));
}
```

Make printing function for arrays[2]

Since we now know that our variable `array` *is* a pointer, it should become apparent how we could split the array printing into its own function:

```

void arrayPrinter(int* array, int length){
  int i;
  for(i=0; i<length; i++){
    printf("array[%d] = %d\n", i, *(array + i));
  }
}


int main(int argc, char** argv){

  int array[5];

  //...
  //This section populates the array as above
  //...


  //Since array is simply a pointer to the beginning of our array
  //  we invoke our function like this
  arrayPrinter(array);
}
```

[1]For more info, see `https://stackoverflow.com/questions/5610298/why-does-int-pointer-increment-by-4-rather-than-1`

[2]I played around with moving this into the previous section, but I'm unsure where it works better

Dynamic Array Allocation

So far, we've only seen how to declare an array when we know the size up front. As it turns out, if we don't know the size of the array at compile time, we need to do something different:

```
int main(int argc, char** argv){
  int length;

  printf("Enter the length of the array:\n");
  //We need to provide the address of length -- we're changing its value as a side effect
  scanf("%d", &length);

  //malloc has a signature which looks like:
  //  void* malloc(size)

  //The size that the array needs to be is sizeof(int)*length
  int* array = malloc(length *sizeof(int));

  //At this point we have an array just as before
}
```

At this point, it's useful to note that even though this memory has been allocated to us, the computer did not "set" the memory in any way[3] – it will hold whatever value was there before it was allocated!

In-place Swap

```
void in_place_swap(int* x, int* y){
  *x ^= *y;
  *y ^= *y;
  *x ^= *y;
}

int main(int argc, char** argv){

  int x = 1;
  int y = 2;

  //We need to provide the memory locations of x and y to our function
  in_place_swap(&x, &y);
  //After this, x=2, y=1... wow!
}
```

To see why this works, why not do a little example:

```
void in_place_swap(int* x, int* y){

  //We'll do this in binary so it becomes apparent
  //Let *x = 01, *y = 10

  *x ^= *y;
  //Now *x = 01 ^ 10 = 11

  *y ^= *y;
  // *y = 11 ^ 10 = 01... the original value of *x

  *x ^= *y;
  // *x = 11 ^ 01 = 10... the original value of *y
}
```

[3]I mentioned `calloc()` at this point in the lab, and played around with uninitialized variables