Arrays and Strings

Section 3.1 and Section 3.2

Arrays

- An **array** is intended for storing multiple values with the same variable name
  - Like a box with multiple compartments
- Each value has to be the same data type
  - Can't mix and match

- Each compartment in the array is known as an **element** or **cell** (in textbook)
- Size of each element depends on the data type
- An **index** is used to refer to a particular element in an array

```
int a[2];
float b[3];
double c[4];
char d[5];

a[0] = 5;
b[1] = 4.0;
c[2] = 14.7;
d[4] = 'a';
```

Variable	Address	Value
a[0]	400 – 403	5
a[1]	404 – 407	
b[0]	408 – 411	
b[1]	412 – 415	4.0
b[2]	416 – 419	
c[0]	420 – 427	
c[1]	428 – 435	
c[2]	436 – 443	14.7
c[3]	444 – 451	
d[0]	452	
d[1]	453	
d[2]	454	
d[3]	455	
d[4]	456	'a'

- Blank cells aren't empty, have random values there
- Range of valid indices for an array are known as the array bounds
  - First element always at index 0
  - Last element always at 1 less than the number of elements
- C does not check that indices stay within proper bounds!

```
b[4] = 15.9;

printf("%f\n", b[4]);
```

- b[4] is out of bounds for b
- Each element in b uses 4 bytes (float data type)
- Each element in c uses 8 bytes (double data type)
- So "b[4]" corresponds to second 4 bytes in c[0]

```
b[33333] = 15.9;

printf("%f\n", b[33333]);
```

- Array accesses waaaay out of bounds will likely cause a crash
- Operating system recognizes access attempt is out of bounds allowed for that particular program
- Often gives the beloved "segmentation fault" message

- **out_of_bounds.c**
  - Accessing into another array
  - Trying to access way out of bounds

Multidimensional Arrays

- Can have multidimensional arrays in C
- However, actually laid out like a long single dimensional array in memory
  - Arranged in order of the rows

```
int a[3][2];
a[0][1] = 7;
a[1][0] = 13;
```

Variable	Address	Value
a[0][0]	400 – 403	
a[0][1]	404 – 407	7
a[1][0]	408 – 411	13
a[1][1]	412 – 415	
a[2][0]	416 – 419	
a[2][1]	420 – 423	

Strings

- In C, a string is a special type of array
  - An array of characters that ends with **'\0'**
    - Known as the **null character**
    - The array can be longer than where the **'\0'** is, but the string is understood to be from index 0 until the element containing '\0'

```
char d[8];

d[0] = 'H';
d[1] = 'e';
d[2] = 'l';
d[3] = 'l';
d[4] = 'o';
d[5] = '\0';
```

Variable	Address	Value
d[0]	400	'H'
d[1]	401	'e'
d[2]	402	'l'
d[3]	403	'l'
d[4]	404	'o'
d[5]	405	'\0'
d[6]	406	
d[7]	407	

- To print out a string, we could print it character by character..

```
printf("%c%c%c%c%c\n", d[0], d[1], d[2], d[3], d[4]);
```

- But actually have a format specifier for strings: **%s**
    - Prints out from the first character until the element that contains **'\0'**

```
printf("%s\n", d);
```

- Notice only needed to supply the variable name d
- An array's variable name actually contains the address of the beginning of the array (address of the first element)

Address Label	Variable	Address	Value
d	d[0]	400	'H'
	d[1]	401	'e'
	d[2]	402	'l'
	d[3]	403	'l'
	d[4]	404	'o'
	d[5]	405	'\0'
	d[6]	406	
	d[7]	407	

- To get a string input from the user, scanf() uses **%s** as well

```
int x;
float f;
char s[6];

scanf("%d", &x);
scanf("%f", &f);
scanf("%s", s);
```

- Notice the string name didn't have a **&** in front of it like the int and float variables did
- **&** is the **address of** or **reference operator**
- Provides the memory address of variable it precedes
- Since array names contain the memory address already, don't need an **&**

- **ref_op.c**
    - **%p** for printing out a memory address with printf()
    - **&** for getting the address of a variable

Making String Variables

- There's a few variations of making string variables in C
- Most basic is to make an empty array of characters:
    - **char s1[10];**
    - Contains 10 slots, so it could hold a string of up to 9 characters + the null character

- Could fill with scanf()
  - **scanf("%s", s1);**
- Or could fill each slot individually
  - **s1[0] = 'h';**
  - **s1[1] = 'i';**
  - **s1[2] = '\0';**
- Some other possible ways to fill too

- Another way is to make a string variable that already has a string in it:
  - **char * s2 = "hello";**
    - * here is not multiplication, talk more about later in course
  - Creates an array of size six (five letters in "hello" plus '\0')
  - Note the size is not explicitly specified, it counts how many elements the array needs

- Another variation is to make an array of characters and specify its contents, stating explicitly how many slots to have:
  - **char s3[10] = "bye";**
  - Though only four slots are being used, the other six still exist
    - Other unused slots are populated with '\0'