Objectives:
(a) Describe how an array is stored in memory.
(b) Define a string, and describe how strings are stored.
(c) Describe the implications of reading or writing beyond the boundary of an array.
(d) Describe how to change the values of individual array elements.

I. Arrays

1. Why Use Arrays?

Consider the following problem: Suppose an instructor has just finished grading the six-week exams for the twenty midshipmen in her section of EC310. Write a C program that will compute the average and also determine how far each student’s individual grade is from the average.

One way to start this program would be to declare twenty variables to hold the twenty grades:

```c
float student1_grade, student2_grade, student3_grade ... etc., etc.
```

This is cumbersome. All these variables! Ugh. Ain’t nobody got time for that! Suppose there were 100 students in the section. We would need 100 variables to hold the 100 grades.

Suppose we consider a different (better!) approach. Instead of having 20 separate variables to hold our 20 six-week exam scores, we will use a "large box with multiple slots." We might name the entire large box `six_week_grade`. The top slot will hold the first student’s six-week exam grade, the second slot will hold the second student’s six-week exam grade, and so forth. Instead of using the term "large box with multiple slots", let’s call this an array. In C, an array is frequently termed a buffer.

So that we can more easily draw our arrays, we will imagine that our section has five students. The concepts will, of course, apply to arrays of any size. Let's say, for the purposes of this discussion, that our five students: Mid 1, Mid 2, Mid 3, Mid 4 and Mid 5, have grades of 98, 87.5, 94, 90 and 92, respectively.
2. **Arrays** An array is a collection of data, all of the same type. Recall that when we say *type* we are referring to `int`, `float` or `char`. More precisely, an array is a consecutive group of memory locations, all with the same name, all holding the same type of data. Our array of five student grades (all of type `float`) might be arranged in main memory as:

```plaintext
    08048372  98  ← First student's grade goes here
    08048376  87.5 ← Second student's grade goes here
    0804837a  94  ← Third student's grade goes here
    0804837e  90  ← Fourth student's grade goes here
    08048382  92  ← Fifth student's grade goes here
```

Notice first that our five array elements are stored in consecutive memory locations. Second, note that the addresses are separated by four bytes, since each value of type `float` is stored in four bytes. The precise location in main memory where the array will be stored is determined by the compiler.

You might be surprised to know that the third and most important point to note in the figure above are the ellipses (i.e., the three dots) shown at the top and bottom of the array! There are items in main memory "above" our array, and items in main memory "below" our array.

Suppose we want to give our array the name: `six_week_grade`. Recalling that our array will hold five grades, each of type `float`, we can declare our array as:

```c
float six_week_grade[ 5 ] ;
```

More generally, the syntax for declaring an array is

```plaintext
    type array_name[ number of items in the array ] ;
```

All items stored in the array must be of the same type  
Same rules as for variable names  
Also called the "size" of the array; must be an integer or an expression that evaluates to an integer.

The following would all be valid examples of array declarations:

```c
float temperatures[31];
int calories[90];
float migraine_intensity_level[1000];
```

Returning to our example, when we declare our array as

```c
float six_week_grade[ 5 ] ;
```

the compiler will reserve adjacent memory for five variables of type `float`. The entire array will be named `six_week_grade`. The picture would look like this (where the exact addresses are chosen by the compiler):
The values stored in these memory locations are, presently, "garbage values".

Note that when we declare an array, the size of the array can be a variable, so long as the variable has a value that is known. The following code is perfectly fine, and would produce the same array as that shown above:

```c
int number_in_class = 5;
float six_week_grade[number_in_class];
```

This will work since by the time we reach the declaration for the array

```c
float six_week_grade[number_in_class];
```

the value of the size of the array (the variable named number_in_class) is already explicitly known from the a prior declaration that has already been encountered:

```c
int number_in_class = 5;
```

**Practice Problem**

Write a declaration that could be used to hold the individual letter grades for 250 midshipmen.

**Solution:**

Obviously we do not want garbage values in our array. How do we get our student grades into this array?

**3. Array Elements** The individual elements in the array are variables. Each individual memory location in the array is indexed by a position number in the array.

In our array named `six_week_grade`, the individual variables in the array (i.e., the variables that will hold the 5 individual six-week scores) are indexed from 0 to 4. The index is placed in square brackets after the array name. So, the name of the first variable in the array is `six_week_grade[0]`, the name of the second variable in the array `six_week_grade[1]`, and so forth."
The individual array elements are variables and can be used in expressions just as you would ordinarily use any variable. So, for example, the line of code

\[
\text{six\_week\_grade}[\ 2\ ] = 94;
\]

would change our picture to this:

So, the third midshipman grade, which we refer to as `six\_week\_grade[2]`, is now stored in the array. (The other four array values are still garbage values.)

**Important point guaranteed to cause confusion**: Note that the first variable in the array has an index of zero, not one. This is counter-intuitive. In the above example, you would think that the first six-week exam grade should be indexed as `six\_week\_grade[1]` since, after all, it is the first score. *You would be wrong!* The first score in our array of scores is indexed as `six\_week\_grade[0]`. Most programming languages (e.g., C, C++, Java, JavaScript) start with the index at zero just to make it easier for the CPU to index into the array.

We could fill in our array in memory by adding the lines of code:

\[
\begin{align*}
\text{six\_week\_grade}[\ 0\ ] &= 98; \\
\text{six\_week\_grade}[\ 1\ ] &= 87.5; \\
\text{six\_week\_grade}[\ 3\ ] &= 90; \\
\text{six\_week\_grade}[\ 4\ ] &= 92;
\end{align*}
\]

after which our array is stored as shown:

It bears repeating: The individual array elements are variables and can be used in expressions just as you would ordinarily use any variable. If we wanted to add two points to the first midshipman's grade we could use

\[
\text{six\_week\_grade}[\ 0\ ] = \text{six\_week\_grade}[\ 0\ ] + 2;
\]
If we wanted to read the value of the third midshipman's grade in from the keyboard, we could use

```
scanf("%f", &six_week_grade[2] );
```

If we wanted to print the second midshipman grade to the monitor, we could use

```
printf("%f", six_week_grade[1] );
```

We can use array elements in Boolean expressions, such as `if ( score[3] > 90 ) ....`

Note that when referring to an array element, the index does not need to be an integer constant. We can use any expression in the brackets that evaluates to an integer. As an example, the `for` loop below might be used to read in from the keyboard the grades for 5 students.

```
for ( number = 0 ; number < 5 ; number = number + 1 )
{
    printf( "Enter score for student %d : " , number + 1 );
    scanf( "%f" , &six_week_grade[ number ] );
}
```

**Practice Problem**

Suppose we have 5 students in EC312. A portion of a C program that declares an array of `floats` named `six_week_grade` that will hold the midterm grades for the class is shown below. Your program should allow the user to enter the midterm grades at runtime, and should then print out the midterm grades. Your program output should appear as shown below:

```
midshipman@EC310:/work $ ./a.out
Enter score for student 1 : 98
Enter score for student 2 : 87.5
Enter score for student 3 : 94
Enter score for student 4 : 90
Enter score for student 5 : 92
Student 1: 98.000000
Student 2: 87.500000
Student 3: 94.000000
Student 4: 90.000000
Student 5: 92.000000
midshipman@EC310:/work $
```

Fill in the one missing line of code.

```
#include <stdio.h>
int main()
{
    float six_week_grade[5];
    int number;

    for ( number = 0 ; number < 5 ; number = number + 1 )
    {
        printf( "Enter score for student %d : " , number + 1 );
        scanf( "%f" , &six_week_grade[ number ] );
    }

    for ( number = 0 ; number < 5 ; number = number + 1 )
    {
        printf( "Student %d: %f
", number + 1, six_week_grade[number] );
    }
}
```
**Practice Problem**

Consider an array declared as

```c
float pay[4];
```

(a) How much memory is reserved for this array?

Solution:

(b) What are the four variables that are collected into this array?

Solution:

(c) What is the name of the array of four variables?

Solution:

(d) The first array element is stored at address 0x0000008e, what is the address of the second element?

Solution:

---

4. **Initialization of Arrays** It is possible to initialize the values of an array in the array's declaration. To initialize arrays in the declaration, we place the initial values in braces, and separate the values with commas.

Our array of student grades could have been initialized by the declaration:

```c
float six_week_grade[ 5 ] = { 98 , 87.5 , 94 , 90 , 92 };
```

There are some caveats to this: First, if we initialize only the first part of an array, the remaining elements are initialized to zero. For example, the declaration

```c
float six_week_grade[ 5 ] = { 98 , 87.5 };
```

has the exact same effect as

```c
float six_week_grade[ 5 ] = { 98 , 87.5 , 0 , 0 , 0 };
```

The second caveat: If you initialize an array when it is declared, you can omit the array size. The size will be set permanently to be the minimum size needed to store the initialization values. So, for example, the array declaration

```c
float six_week_grade[ 5 ] = { 98 , 87.5 , 94 , 90 , 92 };
```

is the same as the declaration

```c
float six_week_grade[   ] = { 98 , 87.5 , 94 , 90 , 92 };
```

When dealing with arrays, you must note that we use the square brackets `[]` in two different ways:

- In the array declaration, the number in the square brackets gives the size of the array (i.e., the number of items in the array). This value must be known when the array is declared.

- Anywhere outside the declaration, the number in the square brackets tells which element in the array (the specific variable) we are referring to.
After six pages of array syntax, you might be thinking: "Wonderful, so what?" As we will see arrays have a horrendous security vulnerability baked into them.

5. **The Dreaded Out-of-Range Error**  
   C will not prevent you from trying to access an array element that is out of the array's range. Stated another way, C will not prevent you from trying to read to or write to "nonexistent" array elements. What exactly does this mean? Consider the array declaration

   ```c
   int salary[3];
   ```

   which declares an array with three variables: `salary[0]`, `salary[1]` and `salary[2]`. But what happens if we have a statement such as

   ```c
   printf( "%d", salary[3] );
   ```

   when there is no variable `salary[3]`? Let's see! Consider the program below.

   ```c
   #include <stdio.h>
   int main()
   {
       int salary[3] = { 1000 , 1500 , 2000 };
       int j;
       for ( j = 0 ; j <= 3 ; j = j + 1 )
       {
           printf("Salary %d is %d \n", j+1 , salary[j] );
       }
   }
   ```

   The output from this program is:

   ```
   midshipman@EC310-VM:~/books src $ ./a.out
   Salary 1 is 1000
   Salary 2 is 1500
   Salary 3 is 2000
   Salary 4 is -1208131596
   midshipman@EC310-VM:~/books src $ 
   ```

   No compilation error results in the program above...but do you see the potential dangers? Where does the last number come from? The answer: It is the value that is located in memory immediately after `salary[2]`! This is a garbage value.

   ![Array diagram](image)

   When we index an array variable using an index outside the range of indices specified in the array's declaration, we commit an *out-of-range error.* Again, it is critical to note that C will not prevent you from looking into memory beyond the end of your array.
What would be the danger in the following program snippet:

```c
float salaries[3];
int j;

(other code not shown)
for ( j = 0 ; j <= 3 ; j = j + 1 )
{
    printf("Enter salary %d: ", j + 1);
    scanf("%f", &salaries[j]);
}

(more code)
```

The program above does not produce any compilation errors, but running this program is potentially very dangerous. Do you see why? Notice that the for loop, in its final iteration, attempts to enter a value into a variable named `salaries[3]`. There is no variable named `salaries[3]`! The program will simply write over whatever was stored in the memory location following `salaries[2]`.

```
                        salary
                           .
            b048370    1000   salary[ 0 ]
            b048374    1500   salary[ 1 ]
            b048376    2000   salary[ 2 ]
    b04837c    Crucial value Not part of the array
    b04837e    Critical value Not part of the array

The program will Overwrite this value!
```

We now switch gears (slightly) and talk about arrays of characters—that is, arrays where each element in the array is of type `char`.

## II. Strings

1. **Introduction**  Suppose the contents of 5 consecutive bytes in computer memory are as follows:

   0100 1110
   0110 0001
   0111 0110
   0111 1001
   0000 0000

   As ASCII characters, this is the same as:

   'N'
   'a'
   'v'
   'y'
   0
Note that 0 is the NULL character. In C, a *string* of characters is a NULL-terminated sequence of characters.

2. **String literals** A literal is similar in notion to a constant. For example, the integer 5 is a literal. The float value 3.1416 is a literal. We can have string literals as well. A string literal is written as a list of characters enclosed within double quotes. For example, "Navy" is a string literal.

Although the length of the string is 4 (the 4 characters 'N', 'a', 'v' and 'y'), it takes 5 bytes to store this string in memory. This is because the NULL character has to be included for C to treat the collection of characters as a string.

3. **String variables** In C, strings—i.e., sequences of characters—are stored as arrays of characters. Here is an example of declaring a string variable using a character array:

   ```c
   char school[5] = "Navy";
   ```

The array named school holds 5 characters: 'N', 'a', 'v', 'y', 0.

It is worth noting again that the length of the string "Navy" is 4, but it actually has 5 characters, since the NULL character which appears at the end is actually part of the string. So if you want a string to hold "Navy", it must have space allotted for at least 5 characters.

You may be wondering: Why the zero at the end of the string? *What's up with that?*

The zero at the end tells C when to stop accessing the array! For example, we have seen many times already that strings are printed to the screen with the `%s` conversion specifier, as in:

   ```c
   printf( "Go %s! Beat %s!\n", school, "Army" );
   ```

How does the printf know when to stop printing out the string named school? As it turns out, printf will print out the first element in the array named school, and then printf will continue to print out characters, one-by-one, until it reaches the zero i.e., the NULL.

3. **Changing the value of a string variable**

   We can initialize a string variable when it is declared (as in `char school[5] = "Navy";`). But, if we assign a string a value when it is declared, can we change the value later?
With other types, such as int, float and char, you were allowed to assign values to the variables in a manner like this:

```c
int favorite_number;
favorite_number = 7;
```

Unfortunately, assignments cannot be done like this with strings. The code below will not compile:

```c
char school[5];
school = "Navy";
```

So... how can we change the value of a string? There are two ways.

A. **Changing a string value character by character.** We can change the value of a string by changing the values of the individual characters. For example:

```c
char school[5] = "Navy";
printf( "%s\n", school );
school[0] = 'U';
school[1] = 'S';
school[2] = 'N';
school[3] = 'A';
printf( "%s\n", school );
```

**Practice Problem**

Continuing the example above, what would happen if we modified two lines of code as shown below:

```c
school[2] = 'A';
school[3] = 0;
printf( "%s\n", school );
```

**Solution**

B. **Changing a string value using strcpy** The second way to change a string’s value is with the “string copy” function. The syntax is

```c
strcpy( s1 , s2 );
```

This function copies the string s2 to the string s1. When the string s2 is copied over the string s1, the `strcpy` function automatically places a closing NULL at the end of the new (modified) string s1.

To use the function `strcpy`, you must have the following line at the top of your program:

```c
#include<string.h>
```

One final note about strings. When entering strings from the keyboard don’t use the ampersand: &. For example, to enter a midshipman’s last name from the keyboard, we would use:

```c
char mid_name[24];
scanf( "%s " , mid_name );
```
Practice Problem

We want to write a C program that declares a string (a character array) and initializes it to "Military Academy", prints this string to the screen, then, within the program, changes the string to the name of your favorite college, and then, once again, prints the string to the monitor.

Your program output should appear as shown below:

![Midshipman@EC310-VM:~/ ]  $ ./a.out
Military Academy
Naval Academy
Midshipman@EC310-VM:~/ ]  $

Fill in the three missing lines of code.

Solution:

```c
#include <stdio.h>
#include <string.h>
int main()
{

    char phrase[] = "Military Academy";

     // Fill in the missing lines here

}
```

OPTIONAL FUN VIDEOS

"It came from California... the students were safe... their computers weren't..."

http://www.youtube.com/watch?v=G2i_6j55bS0

What's that you say? You don't see the danger!. Let's watch another video.

https://youtu.be/Ar-l3FRUdGw
Answer the questions about the character string in memory shown below, where the first element in the string is 0x53.

(a) What is the minimum number of bytes that could have safely been allocated for this string?
(b) Write this declaration, naming the array 'myString'.
(c) What is the address of 'myString[0]'?
(d) What character is at myString[1]?

Solution: (a) (b) (c) (d)

Practice Problem

(a) Write the declaration for an array named LuckyNumbers which will hold 6 integers.

Solution:

(b) Complete this statement to display the 4th LuckyNumber:

```c
    printf("The fourth lucky number is %d\n", _____________);
```

Solution: `printf("The fourth lucky number is %d\n", _____________);`

(c) What happens if I attempt to display LuckyNumbers[9]?
   i. Will it return a value?
   ii. Will I receive an error message?
   iii. Will the program crash?

Solution: i: _____________ ii: _____________ iii: _____________