Lesson 1: Arrays and Strings

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.

(e) Demonstrate the ability to analyze C programs that employ if-else statements and for loops.

(f) Apply Boolean logic to evaluate how selection structures can alter program flow.

1. Arrays

An *array* is a collection of data, stored as a consecutive group of memory locations, all with the same name and all holding the same type of data. The variables that are used in this class are:

Kind of data	type	Amount of space reserved
Integer (e.g., 1, −5, 39)	`int`	stored in 4 bytes
Real numbers (e.g., 2.35, −9.9)	`float`	stored in 4 bytes
A single character (e.g., A, $, b)	`char`	stored in 1 byte

Notice that integers are always stored as four bytes. So, for example, the integer value 2 is represented in binary as 10_2 and would be stored as the four-byte quantity:

```
00000000  00000000  00000000  00000010
```

From our definition of variables we see that an array is simply a collection of variables. Many of the same conventions we learned about variables extend to arrays.

Consider the following problem: Suppose an instructor has just finished grading the six-week exams for the twenty midshipmen in her section of EC312 and would like to write a C program that will compute the average student grade.

One way to start this program would be to have 20 variable declarations:

```
#include <stdio.h>
int main( )
{
      float student1_grade, student2_grade, student3_grade … etc., etc.

      statements;
}
```

Suppose there were 100 students in the section. We would need to declare 100 variables to hold the 100 grades. The need for more complex data structures becomes clear as we see the limitations in scaling up or having a more complex program. Arrays were developed to be implemented in these situations.

1.1 <u>Array Declaration Syntax</u> Every array must be declared before it can be used in any statement. When you declare an array in a C program, you tell the computer the array's name (i.e., its identifier), what kind of data you are storing in the variable and how many elements are in the array. The compiler then creates machine language instructions which reserve the appropriate amount of memory to hold the array.

An array declaration has the following syntax:

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

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

Examples of Array Declarations. If we wanted to declare an array `grades`,as shown above, which would hold three students grades as integers, we would use:

```
int grades[3];
```

The following would all be valid examples of array declarations:

```
float temperatures[31];
int calories[90];
```

Practice Problem 3.1

Write a declaration that could be used to hold the percentage grades of 250 midshipmen.

Solution:

Note that when we declare an array, the size of the array can be a variable, as long as the variable has a value that is known. Once an array is declared, its size cannot be changed even if the variable that was used to declare the array is changed. The following code is perfectly fine, and would produce the same array as shown above:

```
int number_in_class = 3;
int grades[number_in_class];
```

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

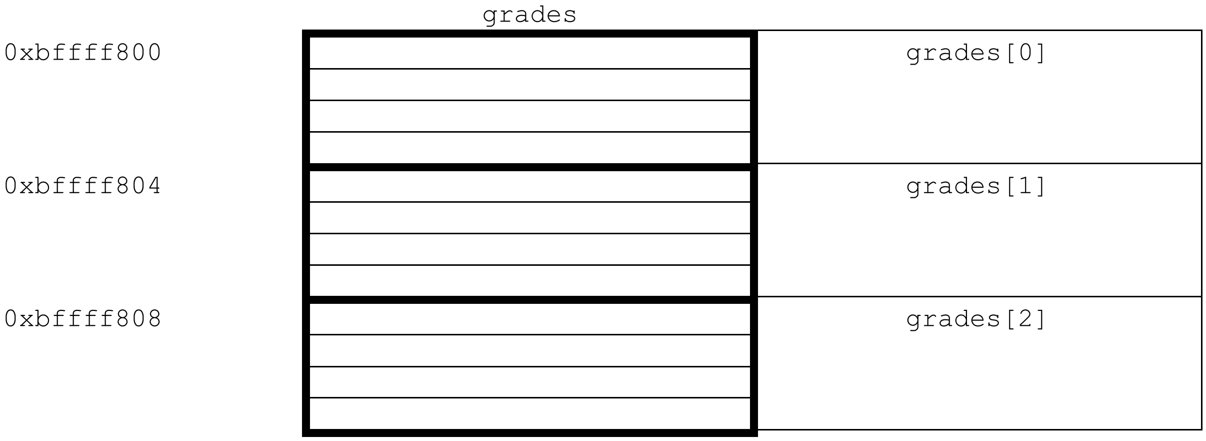
```
int grades[number_in_class];
```

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

```
 int number_in_class = 3;
```

1.2 <u>Array Types</u> When you declare an array, you must tell the computer the ***type***, i.e., the kind of data the array will be holding. Recall that when we say *type* we are referring to `int`, `float` or `char`, our variable types for this course. Note: Arrays of type char are a special type of array called strings and will be discussed in Section 2.

1.3 <u>Array Storage</u> We can imagine that the memory locations for our array are actually labeled with the array's name and each element of the array is known by its name and index. For example, looking at the example array declaration above we can imagine that in main memory, the array is held in a box with multiple slots labeled `grades[0], grades[1] and grades[2]`:



There are several things we should note from this image:

- The three array elements are stored in consecutive memory locations.
- The addresses are separated by four bytes, since each value of type integer is stored in four bytes.
- The array elements are stacked in order from the top down.

The precise location in main memory where the array will be stored is determined by the compiler.

1.4 Array Identifiers and Array Elements Much the same as the more general class of variables, the programmer is free to choose the name of the array. The individual elements in the array are variables. Each individual memory location in the array is indexed by a position number in the array and each element of the array can be referenced by its name and index. The individual array elements are variables and can be used in statements in the body of the C program just as you would ordinarily use any variable. It bears repeating: *The individual array elements are variables and can be used in expressions just as you would ordinarily use any variable.* This includes using array elements in assignment statements, output statements, and input statements.

The first variable in an array has an index of zero. There is an underlying “mismatch” here between the first element and index of 0, but 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.

For example to assign a value to the first element of the array we would use:

```
grades[0] = 98;
```

If we wanted to add two points to the first midshipman's grade we could use:

```
grades[0]  = grades[0] + 2;
```

If we wanted to print the second midshipman’s grade to the monitor we could use:

```
printf("%d" , grades[1]);
```

If we wanted to read the value of the third midshipman's grade in from the keyboard we could use:

```
scanf("%d", &grades[2]);
```

1.5 Initializing Arrays The values stored in memory are, presently, "*garbage values.*" 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 curly braces, and separate the values with commas. Our array of student grades could have been initialized by the declaration:

```
int grades[3] = {98, 100, 85};
```

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

```
int grades[3] = {98 , 100};
```

has the exact same effect as

```
int grades[3] = {98 , 100, 0};
```

The second caveat: If you initialize all values in 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

```
int grades[3] = {98, 100, 85};
```

is the same as the declaration

```
int grades[ ] = {98, 100, 85};
```

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’s index) we are referring to.

Practice Problem 3.2

Consider an array declared as `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:

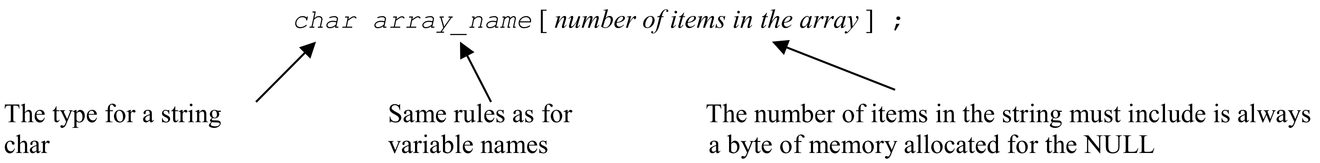
(d) The first array element is stored at address `0x0000008e`, what is the address of the second element?

Solution:

2. Strings

Just as every square is a rectangle, but not every rectangle is a square, a string is a special type of array. A *string* is an array of characters where each element in the array is of type `char` and the array is terminated by a null character. What does "terminate" mean? It means, the computer can identify the end of a string by finding the NULL character (0x00) after the sequence of characters that make up the string.

2.1 <u>String Declaration Syntax</u> The syntax to declare a string is the same as the syntax to declare an array, except there are two differences to note: the data type is `char`, and the number of items in the array must include space for a NULL value (`0x00`), which always terminates a string.



`char array_name` [*number of items in the array*] `;`

The type for a string
char

Same rules as for
variable names

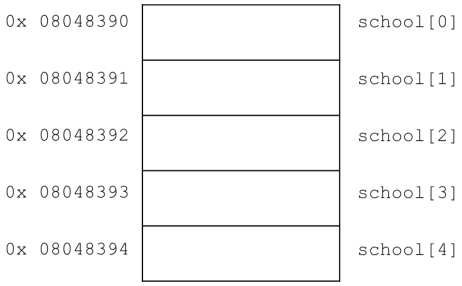
The number of items in the string must include is always
a byte of memory allocated for the NULL

2.2 <u>String Storage</u> The only noticeable distinction in string storage from array storage is that while the string elements are stored in consecutive memory locations, the addresses are not separated by 4 bytes. As type `char`, each element is one byte, so the consecutive elements of the array are stored at consecutive memory addresses with no skipping (i.e. the address of the consecutive elements differ by 1). Again, the precise location in main memory where the array will be stored is determined by the compiler.

For example, if we have the following string:

```
        char school[5];
```

we can imagine that in main memory, each individual character would be stored at consecutive memory locations labeled `school[0], school[1], school[2], school[3] and school[4]:`



0x 08048390		school[0]
0x 08048391		school[1]
0x 08048392		school[2]
0x 08048393		school[3]
0x 08048394		school[4]

2.3 <u>String Elements</u> The individual elements in the string are variables of type char. Each element of the string can be referenced by its name and index and assigned a value.

For example to assign a value to the first element of the string we could use (recalling from Chapter 2 that we set individual `char` values with single quotes):

```
        school[0] = 'N';
```

The hexadecimal value of the character `N`, which is `0x4E` from the ASCII table, would then be stored at this location. Assigning each character to a place in a string is tedious and inefficient, thus we look to declare a string and initialize its value in one line of C code.

2.4 <u>Initializing Strings</u> The values stored in these memory locations are "*garbage values*" when the string is declared, but not initialized. Just as with variables and arrays, it is possible to initialize the values of a string in the strings' declaration. To initialize strings in the declaration, we place the initial values in double quotes.

An example of declaring and initializing a string variable using a character array:

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

The string school is stored in 5 bytes in memory. The first four memory locations store the 4 characters `'N'`, `'a'`, `'v'` and 'y' and the fifth element of the array is the NULL. This is because the NULL character is necessary to terminate a string and is a part of the string.

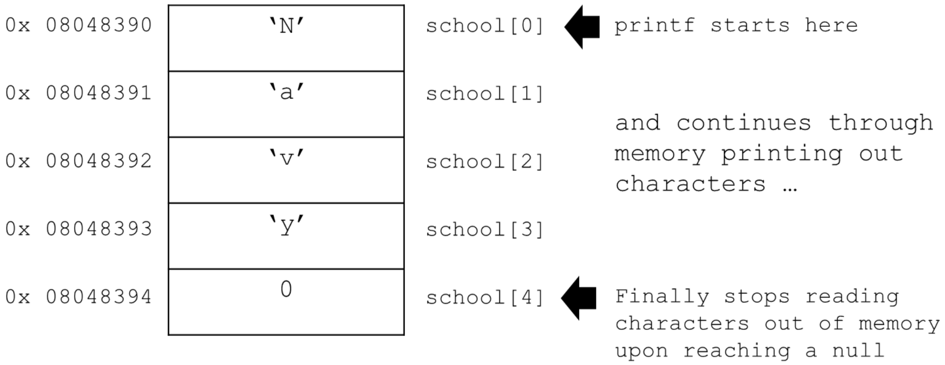
The array named school holds 5 characters: `'N', 'a', 'v', 'y', 0x00`.

	school	
0x 08048390	'N'	school[0]
0x 08048391	'a'	school[1]
0x 08048392	'v'	school[2]
0x 08048393	'y'	school[3]
0x 08048394	0	school[4]

The NULL at the end is a very important element of the array! The NULL signifies the end of the array. For example, we have seen that strings are printed to the screen with the `%s` conversion specifier, as in:

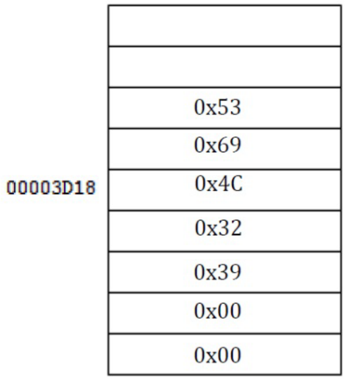
```
        printf( "Go %s!\n", school);
```

The `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 NULL), at which point `printf` knows to stop printing characters.



Practice Problem 3.3

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 stored at `myString[1]`?

Solution:

(a) (b) (c) (d)

2.5 Changing the Values of a String There are two ways the value of the individual variables in a string. Changing a string value character by character using assignment statements for each element of the string, as shown below, or by using the `strcpy` function which is explained later in this Chapter in Section 2.6.

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:

```
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 3.4

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

```
school[2] = 'A';
school[3] = 0 ;    // Note: this value is the value 0 (NULL)
printf( "%s\n", school );
```

Solution:

2.6 String Library

There are predefined libraries which we may import into our C programs which bring in additional functionality that does not need to be defined in the body of the program. To include the library that allows us access to functions for strings, the following statement is added to the header portion (i.e. top or beginning) of a C program:

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

2.6.1 Changing a string value using `strcpy` The second way to change a string's value is with the "string copy" function. The syntax is:

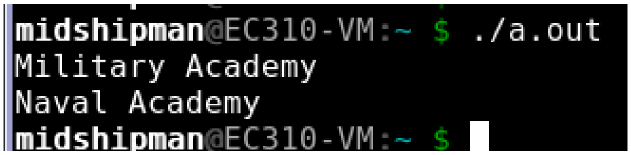
```
strcpy(String1, String2);
```

where `String1` and `String2` are either string variables or text contained in double quotes. This function copies the string `String2` into the memory for the string `String1`. NOTE: When the string `String2` is copied over the string `String1`, the `strcpy` function automatically places a closing NULL at the end of the new (modified) string `String1`.

A `strcpy` example. Consider the C program:

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

        char phrase[]  = "Military Academy" ;
        printf (“%s\n”, phrase);
        strcpy (phrase, “Naval Academy”);
        printf (“%s”, phrase);

}
```

After this C program was compiled and run from the command line the output to the screen would be:



```
midshipman@EC310-VM:~ $ ./a.out
Military Academy
Naval Academy
midshipman@EC310-VM:~ $
```

One final note about strings. When entering strings from the keyboard, don’t use the ampersand (`&`) in a `scanf` statement. This is because (as we will discover later), the *name* of an array refers to the *address* of the array. Since `scanf` needs the address of where the input will be stored, we only need provide the array’s name. For example, to enter a midshipman's last name from the keyboard, we would use:

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

3. More C functionality

In Sections 1 and 2 of this Chapter, we saw that arrays and string provided us more options and efficiency in our variable declarations in C programs. In this section we explore C program functionality and move beyond our three basic statements from Chapter 2.

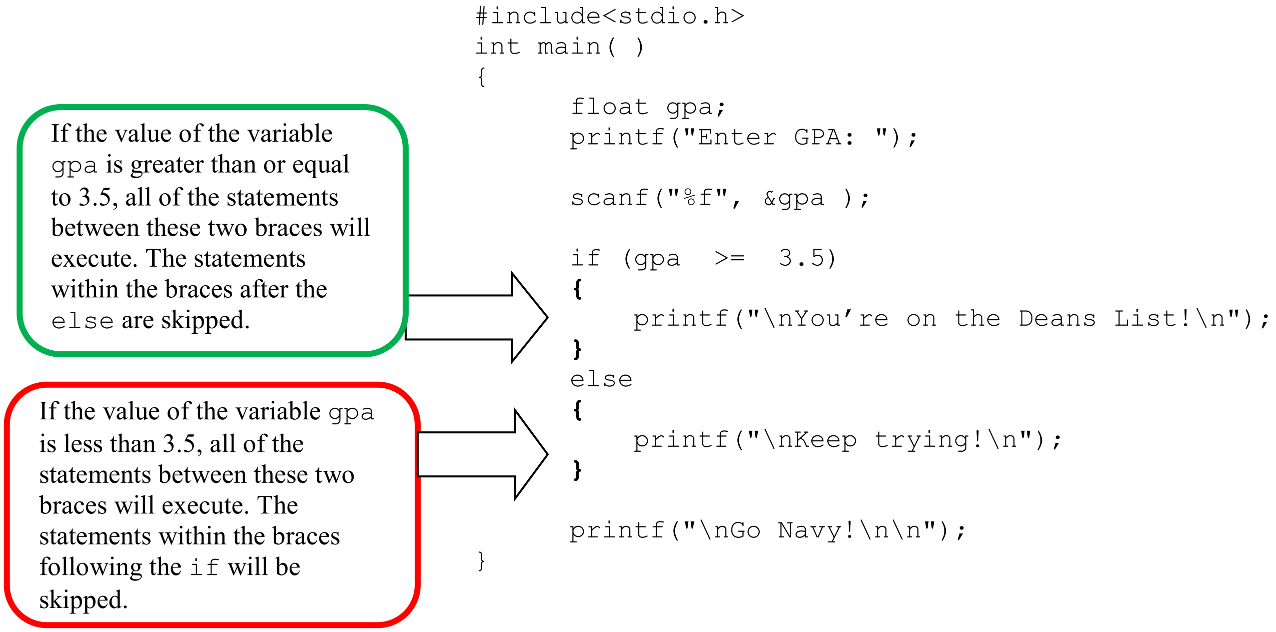
3.1 Flow Control The order in which program statements are executed is called *flow control*. All of the programs that we have seen so far consist of statements executed in order, one after the other. As we will see, we often need to vary the order in which statements are executed.

3.1.1 The `if-else` statement Consider the following example:

Write a program that accepts the user’s GPA as an input and prints “You’re on the Dean's List!” if the GPA is greater than or equal to 3.5, and prints, “Keep trying!” if the GPA is less than 3.5.

Right now, we can’t solve this simple problem because we do not have the ability to introduce two cases in a C program. To solve this problem, C provides an instruction that allows the user to select which statements to execute based on the value of one or more variables. This useful C instruction is the `if-else` statement.

The program that solves the problem above is shown below:



```
#include<stdio.h>
int main( )
{
      float gpa;
      printf("Enter GPA: ");

      scanf("%f", &gpa );

      if (gpa  >=  3.5)
      {
          printf("\nYou’re on the Deans List!\n");
      }
      else
      {
          printf("\nKeep trying!\n");
      }

      printf("\nGo Navy!\n\n");
}
```

If the value of the variable `gpa` is greater than or equal to 3.5, all of the statements between these two braces will execute. The statements within the braces after the `else` are skipped.

If the value of the variable `gpa` is less than 3.5, all of the statements between these two braces will execute. The statements within the braces following the `if` will be skipped.

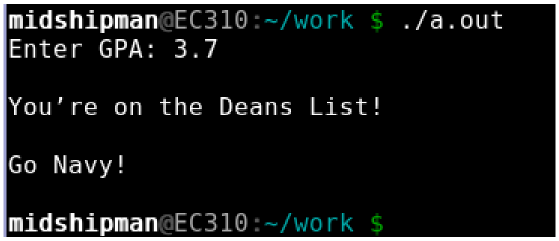
In the code above—immediately after the word `if`—we have a Boolean expression in parenthesis:

```
gpa >= 3.5.
```

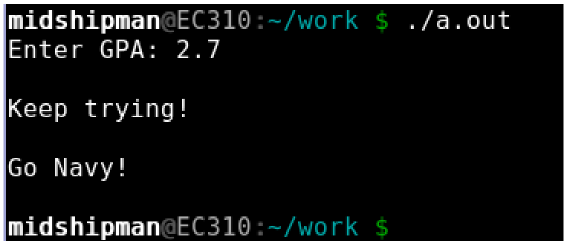
A Boolean expression is an expression that always evaluates to either **true** or **false**. If this particular Boolean expression is true (i.e., if the value of the variable `gpa` is indeed greater than or equal to 3.5), the statements contained within the first set of braces (following the word `if`) will be executed, and the statements within the second set of braces (following the word `else`) will be skipped. On the other hand, if the Boolean expression is false, the statements within the braces following the word `else` will execute and the statements within the braces following the word `if` will be skipped.

Note: there is no semi-colon at the end of the `if` line or the `else` line.

Shown below are two separate executions of the program shown above. Note that in both cases, the `printf` statement `printf("\nGo Navy!\n\n”);` is executed.



```
midshipman@EC310:~/work $ ./a.out
Enter GPA: 3.7

You’re on the Deans List!

Go Navy!

midshipman@EC310:~/work $
```



```
midshipman@EC310:~/work $ ./a.out
Enter GPA: 2.7

Keep trying!

Go Navy!

midshipman@EC310:~/work $
```

The simplest Boolean expression compares numbers and/or variables using a *comparison operator*. You should be familiar with the usual operators: >, >=, < and <=, == and !=. The table below summarizes these comparison operators.

Comparison Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Is equal to
!=	Is not equal to

In C, we can check for equality by using two equals signs in a row, with no space between them. So, for example, a Boolean expression that can be used to check if a `float` variable named `hours` is equal to forty would be:

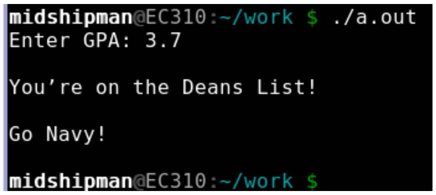
```
hours == 40
```

In C, we can check for inequality by using an exclamation sign followed by an equals sign. So, for example, a Boolean expression that can be used to check if a `char` variable named `grade` is not equal to F would be:

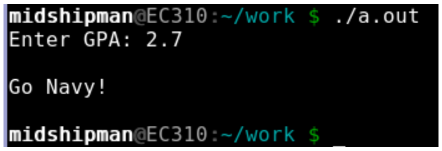
```
grade != 'F'
```

There are two modifications we can make to the `if-else` statement. The first modification is that we don’t *need to* use the `else` part of the statement. In this case, the program performs the statements in braces following the word `if` when the Boolean expression is true, and skips these statements if the Boolean expression is false. Consider our earlier program without the `else` portion, and the corresponding screen captures:

```
#include<stdio.h>
int main( )
{
float gpa;
      printf("Enter GPA:  ");

scanf("%f", &gpa) ;

if (gpa >= 3.5)
{
            printf("\nYou’re on the Deans List!\n");
}

printf("\nGo Navy!\n\n");
}
```



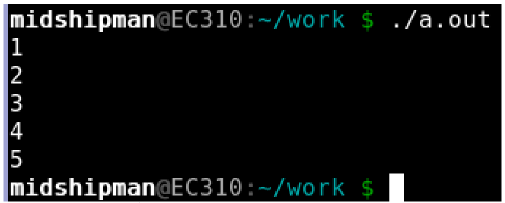
```
midshipman@EC310:~/work $ ./a.out
Enter GPA: 3.7

You’re on the Deans List!

Go Navy!

midshipman@EC310:~/work $
```



```
midshipman@EC310:~/work $ ./a.out
Enter GPA: 2.7

Go Navy!

midshipman@EC310:~/work $ _
```

The second modification is that if there is only a single statement within the curly braces of the `if` or the `else`, then the braces are optional. The programs shown above will work just as well without the braces surrounding the `printf` statements.

3.1.2. The `for` statement Many programs include some actions we would like to perform iteratively. A part of a program that repeats a number of statements is called a *loop*. Let's jump right into examining a program that uses a `for` loop, along with its corresponding output.



```
midshipman@EC310:~/work $ ./a.out
1
2
3
4
5
midshipman@EC310:~/work $
```

```
#include<stdio.h>
int main()
{
        int count;


        for( count = 1 ; count <= 5 ; count = count + 1)
        {
             printf("%d\n", count);
        }
}
```

Any statements within curly braces following the word `for` comprise the *body* of the loop; these statements will be executed each time the loop iterates. In this example, there is only one statement within the body of the `for` loop:

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

Each time the loop iterates, the program will print out the value of the variable `count`, followed by a new line. The question remains: what controls the number of times the loop will iterate?

In this example, the variable `count` will be used to determine the number of times the loop executes. When we enter the `for` loop, the loop control variable (i.e., `count`) is initialized:

```
for( count = 1 ; count <= 5 ; count = count + 1)
```

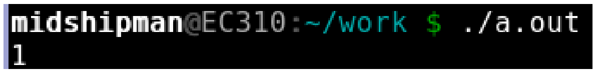
This tells how the loop control variable is initialized.
This initialization occurs only once.

Next, the program checks to see if the Boolean expression is true:

```
for(count = 1; count <= 5; count = count + 1)
```

The loop control variable is compared to 5. This Boolean expression is used to determine if the loop should execute.

Since the variable `count` (at this point in time) is equal to 1, the Boolean expression is true and we execute the statement in the body of the loop. The output we see on the screen is:



```
midshipman@EC310:~/work $ ./a.out
1
```

When we finish executing the body of the loop, we update the loop control variable:

```
for(count = 1 ; count <= 5; count = count + 1)
```

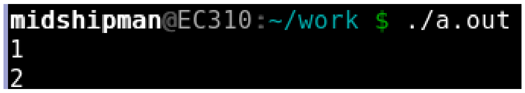
The loop control variable is updated.

The loop control variable `count` is now equal to 2. We once again return to the Boolean expression:

```
for(count = 1; count <= 5; count = count + 1)
```

and see that it is true (2 is indeed less than or equal to 5) and we again execute the body of the loop.

The screen output is now:

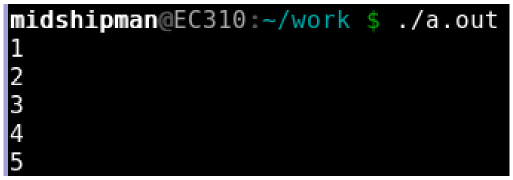


```
midshipman@EC310:~/work $ ./a.out
1
2
```

When we finish executing the body of the loop, we update the loop control variable:

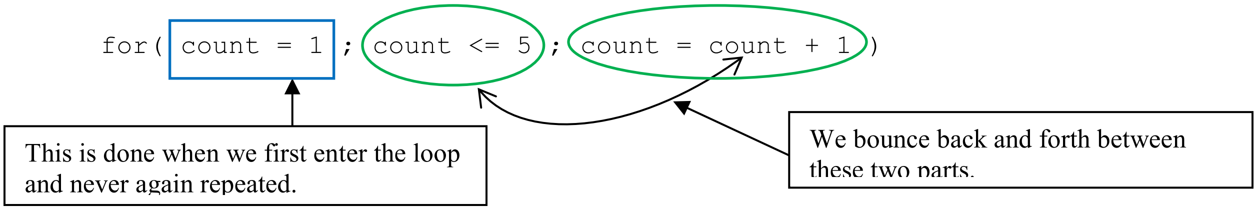
```
for( count = 1 ; count <= 5 ; count = count + 1 )
```

and `count` becomes 3. We then return to the Boolean expression, note that it is true, execute the loop, and update the loop control variable to 4. The loop executes again, and `count` is then updated to 5. The loop executes again (since 5 is less than or equal to 5) and `count` is then updated to 6. When `count` is updated to 6 the Boolean expression becomes false and we exit the loop. That is, when count is updated to 6 and the Boolean expression is false we do not execute the body of the loop anymore. The final screen output is:



```
midshipman@EC310:~/work $ ./a.out
1
2
3
4
5
```

Note that in the `for` loop, the initialization is done only once, and we then "bounce back and forth" between the Boolean expression and the update of the loop control variable.

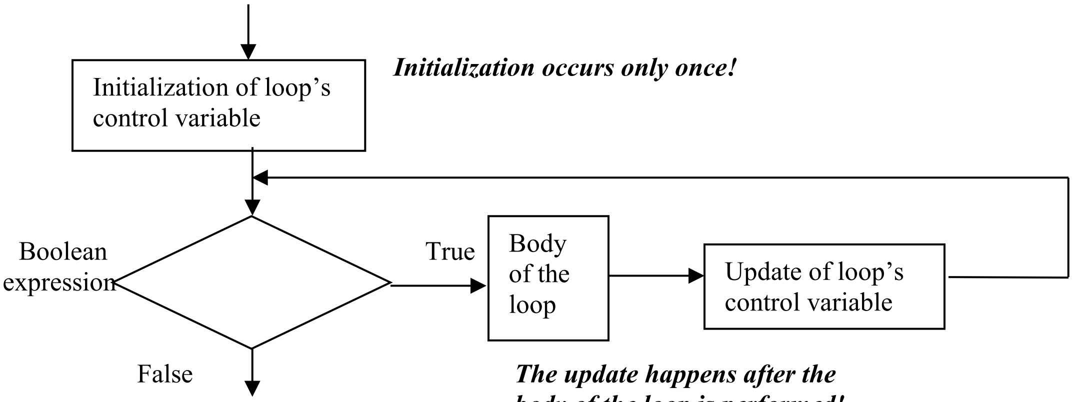


```
for( count = 1 ; count <= 5 ; count = count + 1 )
```

This is done when we first enter the loop and never again repeated.

We bounce back and forth between these two parts.

A *flowchart* for the `for` loop is shown below:


Initialization of loop's control variable
Boolean expression
True
Body of the loop
Update of loop's control variable
False

Initialization occurs only once!

The update happens after the body of the loop is performed!

Note: there is no semi-colon at the end of the `for` line.

Practice Problem 3.5

For each of the `for` loops shown below, state how many times the loop will iterate.

```
a) for( i = 1; i <= 100; i = i + 1)
b) for( i = 3; i > 1; i = i – 1)
c) for( i = 7; i <= 21; i = i + 7)
```

Solution:

(a) (b) (c)

Practice Problem 3.6

Examine the following C program and describe the expected output.

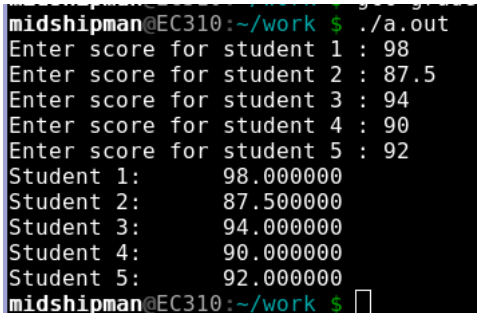
```
#include<stdio.h>
int main( )
{
     int count;
     for(count = 1; count <= 2; count = count + 1 )
     {
          if(count > 1)
               printf("Cyber\n");
          else
               printf("Fun\n");
     }
}
```

Solution:

The following practice problem combines what we have learned about arrays and for loops in one problem. It is a demonstration of the utility of both as tools to write more capable C programs.

Practice Problem 3.7

Suppose we have 5 students in EC312. A portion of a C program that declares an array of `float`s named `six_week_grade` that will hold the midterm grades for the class is shown below. The program should allow the user to enter the midterm grades when the program runs, and should then print out the midterm grades. When the program runs, the 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)
{

            (missing line of code)

      }
}
```

4. 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:

```
int salaries[3];
```

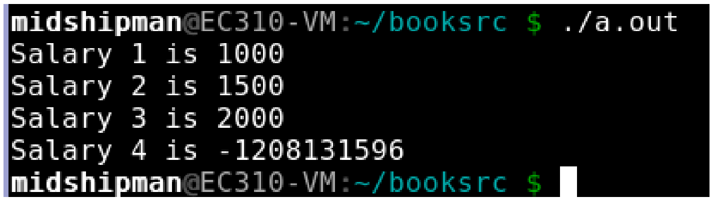
which declares an array with three variables: `salaries[0], salaries[1] and salaries[2]`. But what happens if we have a statement such as:

```
printf(“%d”, salaries[3]);
```

There is no variable `salaries[3]`. Let's see! Consider the program that follows:

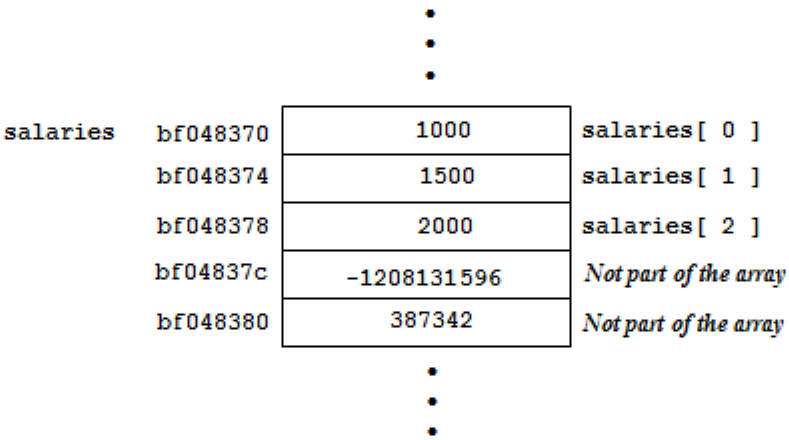
```
#include <stdio.h>
int main()
{
      int salaries[3] = {1000 , 1500 , 2000};
      int j;

      for (j = 0; j <= 3; j = j + 1)
      {
            printf("Salary %d is %d \n",  j+1 , salaries[j]);
      }
}
```

The output from this program is:



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

The program compiles and runs...but do you see the potential dangers? Where does the last number come from? The answer: it is the value of the four bytes in memory immediately after `salaries[2]` read as an integer value! This is a garbage value.



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?

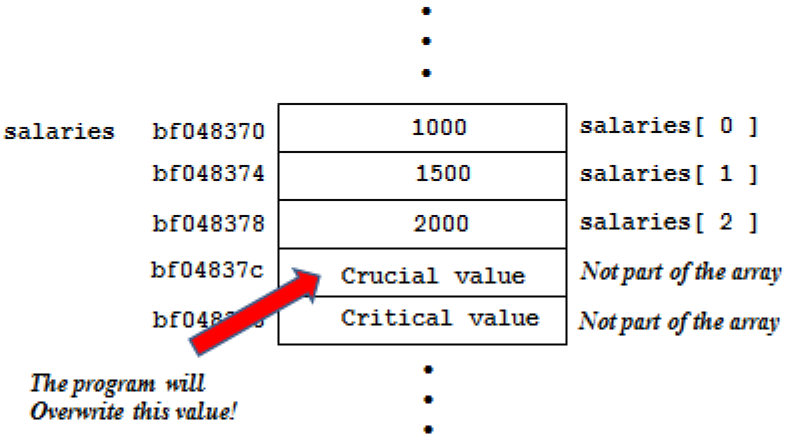
```
float salaries[3];
int j;
```

(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 compiles and runs, but running this program is potentially ***very dangerous.*** Do you see why? Notice that the `for` loop, in its final iteration, it attempts to enter a value into a variable named `salaries[3]`. ***But there is no variable named `salaries[3]`!*** The program will simply write over whatever was stored in the four bytes following the memory location of `salaries[2]`.



Practice Problem 3.8

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

Solution:

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

```
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: