Sams Teach Yourself C in 24 Hours

Hour 12 - Storing Similar Data Items

Gather up the fragments that remain, that nothing be lost.

—John 6:12

In last hour's lesson you learned about pointers and the concept of indirection. In this lesson you'll learn about arrays, which are collections of similar data items and are closely related to pointers. The main topics covered in this lesson are

- Single-dimension arrays
- Indexing arrays
- Pointers and arrays
- Character arrays
- Multidimensional arrays
- Unsized arrays

What Is an Array?

You now know how to declare a variable with a specified data type, such as char, int, float, or double. In many cases, you have to declare a set of variables that have the same data type. Instead of declaring them individually, C allows you to declare a set of variables of the same data type collectively as an array.

An array is a collection of variables that are of the same data type. Each item in an array is called an element. All elements in an array are referenced by the name of the array and are stored in a set of consecutive memory slots.

Declaring Arrays

The following is the general form to declare an array:

```
data-type  Array-Name[Array-Size];
```

Here data-type is the type specifier that indicates what data type the declared array will be. Array-Name is the name of the declared array. Array-Size defines how many elements the array can contain. Note that the brackets ([and]) are required in declaring an array. The bracket pair ([and]) is also called the array subscript operator.

For example, an array of integers is declared in the following statement,

```
int array_int[8];
```

where int specifies the data type of the array whose name is array_int. The size of the array is 8, which means that the array can store eight elements (that is, integers in this case).

In C, you have to declare an array explicitly, as you do for other variables, before you can use it.

Indexing Arrays

After you declare an array, you can access each of the elements in the array separately.

For instance, the following declaration declares an array of characters:

```
char day[7];
```

You can access the elements in the array of day one after another.

The important thing to remember is that all arrays in C are indexed starting at 0. In other words, the index to the first element in an array is 0, not 1. Therefore, the first element in the array of day is day[0]. Because there are 7 elements in the day array, the last element is day[6], not day[7].

The seven elements of the array have the following expressions: day[0], day[1], day[2], day[3], day[4], day[5], and day[6].

Because these expressions reference the elements in the array, they are sometimes called array element references.

Initializing Arrays

With the help of the array element references, you can initialize each element in an array.

For instance, you can initialize the first element in the array of day, which was declared in the last section, like this:

```
day[0] = `S';
```

Here the numeric value of S is assigned to the first element of day, day[0].

Likewise, the statement day[1] = `M'; assigns `M' to the second element, day[1], in the array.

The second way to initialize an array is to initialize all elements in the array together. For instance, the following statement initializes an

integer array, arInteger:

```
int arInteger[5] = {100, 8, 3, 365, 16};
```

Here the integers inside the braces ({ and }) are assigned to the corresponding elements of the array arInteger. That is, 100 is given to the first element (arInteger[0]), 8 to the second element (arInteger[1]), 3 to the third (arInteger[2]), and so on.

Listing 12.1 gives another example of initializing arrays.

TYPE

Listing 12.1. Initializing an array.

```
1:  /* 12L01.c: Initializing an array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:     int list_int[10];
8:
9:     for (i=0; i<10; i++){
10:       list_int[i] = i + 1;
11:       printf( "list_int[%d] is initialized with %d.\n", i, list_int[i]);
12:    }
13:    return 0;
14: }
```

OUTPUT

The following output is displayed on the screen after the executable (12L01.exe) of the program in Listing 12.1 is created and run from a DOS prompt:

```
C:\app>12L01
list_int[0] is initialized with 1.
list_int[1] is initialized with 2.
list_int[2] is initialized with 3.
list_int[3] is initialized with 4.
list_int[4] is initialized with 5.
list_int[5] is initialized with 6.
list_int[6] is initialized with 7.
list_int[7] is initialized with 8.
list_int[8] is initialized with 9.
list_int[9] is initialized with 10.
C:\app>
```

ANALYSIS

As you can see in Listing 12.1, there is an integer array, called list_int, which is declared in line 7. The array list_int can contain 10 elements.

Lines 9_12 make up a for loop that iterates 10 times. The statement in line 10 initializes list_int[i], the ith element of the array list_int, with the value returned from the i + 1 expression.

Line 11 then prints out the name of the element, list_int[i], and the value assigned to the element.

The Size of an Array

As mentioned earlier in this lesson, an array consists of consecutive memory locations. Given an array, like this:

```
data-type  Array-Name[Array-Size];
```

you can then calculate the total bytes of the array by the following expression:

```
sizeof(data-type) * Array-Size
```

Here data-type is the data type of the array; Array-Size specifies the total number of elements the array can take. The result returned by the expression is the total number of bytes the array takes.

Another way to calculate the total bytes of an array is simpler; it uses the following expression:

```
sizeof(Array-Name)
```

Here Array-Name is the name of the array.

The program in Listing 12.2 shows how to calculate the memory space taken by an array.

TYPE

Listing 12.2. Calculating the size of an array.

```
1:  /* 12L02.c: Total bytes of an array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int total_byte;
7:     int list_int[10];
8:
```

```
9:     total_byte = sizeof (int) * 10;
10:    printf( "The size of int is %d-byte long.\n", sizeof (int));
11:    printf( "The array of 10 ints has total %d bytes.\n", total_byte);
12:    printf( "The address of the first element: 0x%x\n", &list_int[0]);
13:    printf( "The address of the last element:  0x%x\n", &list_int[9]);
14:    return 0;
15: }
```

OUTPUT

After running the executable 12L02.exe, I have the following output printed on my screen:

ANALYSIS

```
C:\app>12L02
The size of int is 2-byte long.
The array of 10 ints has total 20 bytes
The address of the first element: 0x1806
The address of the last element:  0x%1818
C:\app>
```

Note that you might get different address values when you run the program in Listing 12.2 on your machine. However, the difference between the address of the first element and the address of the last element should be the same as the one obtained from the output on my machine.

In Listing 12.2, there is an integer array, list_int, which is declared in line 7. The total memory space taken by the array is the result of multiplying the size of int and the total number of elements in the array. As declared in this example, there are a total of 10 elements in the array list_int.

The statement in line 10 prints out the size of int on my machine. You can see from the output that each integer element in the array takes 2 bytes. Therefore, the total memory space (in bytes) taken by the array is 10 * 2. In other words, the statement in line 9 assigns the value of 20, returned by the sizeof (int) * 10 expression, to the integer variable total_byte. Line 11 then displays the value contained by the total_byte variable on the screen.

To prove that the array does take the consecutive memory space of 20 bytes, the address of the first element in the array is printed out by the statement in line 12. Note that the ampersand (&), which was introduced as the address-of operator in Hour 11, "An Introduction to Pointers," is used in line 12 to obtain the address of the first element, list_int[0], in the array. Here the address of the first element is the start address of the array. From the output, you can see that the address of the list_int[0] element is 0x1806 on my machine.

Then, the &list_int[9] expression in line 13 returns the address of the last element in the array, which is 0x1818 on my machine. Thus, the distance between the last element and the first element is 0x1818_0x1806, or 18 bytes long.

As mentioned earlier in the book, hexadecimal is a 16-based numbering system. We know that 0x1818 minus 0x1806 produces 0x0012 (that is, 0x12). Then 0x12 in hexadecimal is equal to 1*16 + 2 that yields 18 in decimal.

Because each element takes 2 bytes, the total number of bytes taken by the array list_int is indeed 20 bytes. You can calculate it another way: The distance between the last element and the first element is 18 bytes. The total number of bytes taken by the array should be counted from the very first byte in the first element to the last byte in the last element. Therefore, the total number bytes taken by the array is equal to 18 plus 2, that is 20 bytes.

Figure 12.1 shows you the memory space taken by the array list_int.

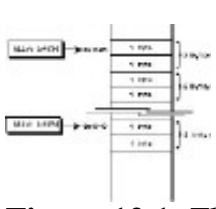

Figure 12.1. *The memory space taken by the array list_int.*

Arrays and Pointers

As I mentioned earlier in this hour, pointers and arrays have a close relationship in C. In fact, you can make a pointer that refers to the first element of an array by simply assigning the array name to the pointer variable. If an array is referenced by a pointer, the elements in the array can be accessed with the help of the pointer.

For instance, the following statements declare a pointer and an array, and assign the address of the first element to the pointer variable:

```
char  *ptr_c;
char  list_c[10];
ptr_c = list_c;
```

Because the address of the first element in the array list_c is the beginning address of the array, the pointer variable ptr_c is actually now referencing the array via the beginning address.

Listing 12.3 demonstrates how to reference an array with a pointer.

TYPE

Listing 12.3. Referencing an array with a pointer.

```
1:  /* 12L03.c: Referencing an array with a pointer */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int *ptr_int;
```

```
7:     int list_int[10];
8:     int i;
9:
10:    for (i=0; i<10; i++)
11:       list_int[i] = i + 1;
12:    ptr_int = list_int;
13:    printf( "The start address of the array: 0x%p\n", ptr_int);
14:    printf( "The value of the first element: %d\n", *ptr_int);
15:    ptr_int = &list_int[0];
16:    printf( "The address of the first element: 0x%p\n", ptr_int);
17:    printf( "The value of the first element: %d\n", *ptr_int);
18:    return 0;
19: }
```

OUTPUT

After the executable 12L03.exe is run from a DOS prompt, the following output is printed on my screen:

ANALYSIS

```
C:\app>12L03
The start address of the array: 0x1802
The value of the first element: 1
The address of the first element: 0x1802
The value of the first element: 1
C:\app>
```

In Listing 12.3, an integer pointer variable, ptr_int, is declared in line 6. Then, an integer array, list_int, which is declared in line 7, is initialized by the list_int[i] = i + 1 expression in a for loop. (See lines 10 and 11.)

The statement in line 12 assigns the address of the first element in the array to the pointer variable ptr_int. To do so, the name of the array list_int is simply placed on the right side of the assignment operator (=) in line 12.

Line 13 displays the address assigned to the pointer variable ptr_int. The output shows that 0x1802 is the start address of the array. (You might get a different address on your machine.) The *ptr_int expression in line 14 returns the value referenced by the pointer. This value is the value contained by the first element of the array, which is the initial value, 1, given in the for loop. You can see that the output from the statement in line 14 shows the value correctly.

The statement in line 15 is equivalent to the one in line 12, which assigns the address of the first element to the pointer variable. Lines 16 and 17 then print out the address and the value kept by the first element, 0x1802 and 1, respectively.

In Hour 16, "Applying Pointers," you'll learn to access an element of an array by incrementing or decrementing a pointer.

Displaying Arrays of Characters

This subsection focuses on arrays of characters. On most machines, the char data type takes one byte. Therefore, each element in a character array is one byte long. The total number of elements in a character array is the total number of bytes the array takes in the memory.

More importantly in C, a character string is defined as a character array whose last element is the null character (\0). Hour 13, "Manipulating Strings," introduces more details about strings.

In Listing 12.4, you see various ways to display an array of characters on the screen.

TYPE

Listing 12.4. Printing out an array of characters.

```
1:  /* 12L04.c: Printing out an array of characters */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char array_ch[7] = {`H', `e', `l', `l', `o', `!', `\0'};
7:     int i;
8:
9:     for (i=0; i<7; i++)
10:       printf("array_ch[%d] contains: %c\n", i, array_ch[i]);
11:    /*---  method I ---*/
12:    printf( "Put all elements together(Method I):\n");
13:    for (i=0; i<7; i++)
14:       printf("%c", array_ch[i]);
15:    /*---  method II ---*/
16:    printf( "\nPut all elements together(Method II):\n");
17:    printf( "%s\n", array_ch);
18:
19:    return 0;
20: }
```

OUTPUT

The following output is a copy from my screen (I obtained these results by running the executable 12L04.exe):

ANALYSIS

```
C:\app>12L04
array_ch[0] contains: H
array_ch[1] contains: e
array_ch[2] contains: l
array_ch[3] contains: l
```

```
array_ch[4] contains: o
array_ch[5] contains: !
array_ch[6] contains:
Put all elements together(Method I):
Hello!
Put all elements together(Method II):
Hello!
C:\app>
```

As you can see from Listing 12.4, a character array, array_ch, is declared and initialized in line 6. Each element in the character array is printed out by the printf() function in a for loop shown in lines 9 and 10. There are a total of seven elements in the array; they contain the following character constants: `H', `e', `l', `l', `o', `!', and `\0'.

There are two ways to display all characters in the array, and to treat them as a character string.

Lines 12_14 show the first way, which fetches each individual element, array_ch[i], consecutively in a loop, and prints out one character next to another by using the character format specifier %c in the printf() function in line 14.

The second way is simpler. You tell the printf() function where to find the first element to start with. Also, you need to use the string format specifier %s in the printf() function as shown in line 17. Note that the array_ch expression in line 17 contains the address of the first element in the array—that is, the start address of the array.

You may be wondering how the printf() function knows where the end of the character array is. Do you remember that the last element in the character array array_ch is a \0 character? It's this null character that marks the end of the character array. As I mentioned earlier, a character array ended with a null character is called a character string in C.

The Null Character (\0)
The null character (\0) is treated as one character in C; it is a special character that marks the end of a string. Therefore, when functions like printf() act on a character string, they process one character after another until they encounter the null character. (You'll learn more about strings in Hour 13.)

The null character (\0), which is always evaluated as FALSE, can also be used for a logical test in a control-flow statement. Listing 12.5 gives an example of using the null character in a for loop.

TYPE

Listing 12.5. Stopping printing at the null character.

```
1:  /* 12L05.c: Stopping at the null character */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char array_ch[15] = {`C', ` `,
7:                          `i', `s', ` `,
8:                          `p', `o', `w', `e', `r',
9:                          `f', `u', `l', `!', `\0'};
10:    int i;
11:    /*  array_ch[i] in logical test */
12:    for (i=0; array_ch[i] != `\0'; i++)
13:       printf("%c", array_ch[i]);
14:    return 0;
15: }
```

OUTPUT

By running the executable 12L05.exe, I obtain the following output:

ANALYSIS

```
C:\app>12L05
C is powerful!
C:\app>
```

In Listing 12.5, a character array, array_ch, is declared and initialized with
the characters (including the space characters) from the string C is powerful!, in lines 6_9.

Note that the last element in the array contains the null character (\0), which is needed later in a for loop.

The for loop in lines 12 and 13 tries to print out each element in the array array_ch to show the string C is powerful! on the screen. So in the first expression field of the for statement (loop), the integer variable i, which is used as the index to the array, is initialized with 0.

Then, the expression in the second field, array_ch[i] != `\0', is evaluated. If the expression returns logical TRUE, the for loop iterates; otherwise, the loop stops. Starting at the first element in the array, the array_ch[i] expression keeps returning TRUE until the null character is encountered. Therefore, the for loop can put all characters of the array on the screen, and stop printing right after the array_ch[i] returns logical FALSE. In fact, you can simplify the array_ch[i] != `\0' expression in the second field of the for statement to array_ch[i] because `\0' is evaluated as FALSE anyway.

Multidimensional Arrays

So far, all the arrays you've seen have been one-dimensional arrays, in which the dimension sizes are placed within a pair of brackets ([and]).

In addition to one-dimensional arrays, the C language also supports multidimensional arrays. You can declare arrays with as many dimensions as your compiler allows.

The general form of declaring a N-dimensional array is

```
data-type  Array-Name[Array-Size1][Array-Size2]. . .
[Array-SizeN];
```

where N can be any positive integer.

Because the two-dimensional array, which is widely used, is the simplest form of the multidimensional array, let's focus on two-dimensional arrays in this section. Anything you learn from this section can be applied to arrays of more than two dimensions, however.

For example, the following statement declares a two-dimensional integer array:

```
int  array_int[2][3];
```

Here there are two pairs of brackets that represent two dimensions with a size of 2 and 3 integer elements, respectively.

You can initialize the two-dimensional array array_int in the following way:

TYPE

```
array_int[0][0] = 1;
array_int[0][1] = 2;
array_int[0][2] = 3;
array_int[1][0] = 4;
array_int[1][1] = 5;
array_int[1][2] = 6;
```

which is equivalent to the statement

```
int array_int[2][3] = {1, 2, 3, 4, 5, 6};
```

Also, you can initialize the array_int array in the following way:

```
int array_int[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Note that array_int[0][0] is the first element in the two-dimensional array array_int; array_int[0][1] is the second element in the array; array_int[0][2] is the third element; array_int[1][0] is the fourth element; array_int[1][1] is the fifth element; and array_int[1][2] is the sixth element in the array.

The program in Listing 12.6 shows a two-dimensional integer array that is initialized and printed out on the screen.

Listing 12.6. Printing out a two-dimensional array.

OUTPUT

```
1:  /* 12L06.c: Printing out a 2-D array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int two_dim[3][5] = {1, 2, 3, 4, 5,
7:                          10, 20, 30, 40, 50,
8:                          100, 200, 300, 400, 500};
9:     int i, j;
10:
11:    for (i=0; i<3; i++){
12:       printf("\n");
13:       for (j=0; j<5; j++)
14:          printf("%6d", two_dim[i][j]);
15:    }
16:    return 0;
17: }
```

ANALYSIS

The following output is obtained by running the executable 12L06.exe:

```
C:\app>12L06
     1     2     3     4     5
     10    20    30    40    50
     100   200   300   400   500
C:\app>
```

As you can see in Listing 12.6, there is a two-dimensional integer array, two_dim, declared and initialized in lines 6_8.

In lines 11_15, two for loops are nested together. The outer for loop increments the integer variable i and prints out the newline character \n in each iteration. Here the integer variable i is used as the index to the first dimension of the array, two_dim.

The inner for loop in lines 13 and 14 prints out each element, represented by the two_dim[i][j] expression, by incrementing the index to the second dimension of the array. Therefore, I obtain output like the following

```
     1     2     3     4     5
     10    20    30    40    50
     100   200   300   400   500
```

after the two nested for loops are run successfully.

Unsized Arrays

As you've seen, the size of a dimension is normally given during the declaration of an array. It means that you have to count each element in an array. It could be tedious to do so, though, especially if there are many elements in an array.

The good news is that the C compiler can actually calculate a dimension size of an array automatically if an array is declared as an unsized array. For example, when the compiler sees the following unsized array:

```
int  list_int[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90};
```

it will create an array big enough to store all the elements.

TYPE

Likewise, you can declare a multidimensional unsized array. However, you have to specify all but the leftmost (that is, the first) dimension size. For instance, the compiler can reserve enough memory space to hold all elements in the following two-dimensional unsized array:

```
char list_ch[][2] = {
         `a', `A',
         `b', `B',
         `c', `C',
         `d', `D',
         `e', `E',
         `f', `F',
         `g', `G'};
```

The program in Listing 12.7 initializes a one-dimensional character unsized array and a two-dimensional unsized integer array, and then measures the memory spaces taken for storing the two arrays.

Listing 12.7. Initializing unsized arrays.

```
1:  /* 12L07.c: Initializing unsized arrays */
2:  #include <stdio.h>
3:
```

OUTPUT

```
4:  main()
5:  {
6:     char array_ch[] = {`C', ` `,
7:                        `i', `s', ` `,
8:                        `p', `o', `w', `e', `r',
9:                        `f', `u', `l', `!', `\0'};
10:    int list_int[][3] = {
11:           1, 1, 1,
12:           2, 2, 8,
13:           3, 9, 27,
14:           4, 16, 64,
15:           5, 25, 125,
16:           6, 36, 216,
17:           7, 49, 343};
18:
19:    printf("The size of array_ch[] is %d bytes.\n", sizeof (array_ch));
20:    printf("The size of list_int[][3] is %d bytes.\n", sizeof (list_int));
21:    return 0;
22: }
```

ANALYSIS

The following output is obtained by running the executable 12L07.exe:

```
C:\app>12L07
The size of array_ch[] is 15 bytes.
The size of list_int[][3] is 42 bytes.
C:\app>
```

A character unsized array, array_ch, is declared and initialized in lines 6_9. In
lines 10_17, a two-dimensional unsized integer array, list_int, is declared and initialized too.

The statement in line 19 measures and prints out the total memory space (in bytes) taken by the array array_ch. The result shows that the unsized character array is assigned 15 bytes of memory to hold all its elements after compiling. When you calculate the total number of the elements in the character array manually, you find that there are indeed 15 elements. Because each character takes one byte of memory, the character array array_ch takes a total of 15 bytes accordingly.

Likewise, the statement in line 20 gives the total number of bytes reserved in the memory for the unsized two-dimensional integer array list_int. Because there are a total of 21 integer elements in the array, and an integer takes 2 bytes, the compiler should allocate 42 bytes for the integer array list_int. The result printed out by the printf() function in line 20 proves that there are 42 bytes reserved in the memory for the two-dimensional integer array. (If the size of int or char is different on your machine, you may get different values for the sizes of the arrays in the program of Listing 12.7.)

Summary

In this lesson you've learned the following:

- An array is a collection of variables that are of the same data type.
- In C, the index to an array starts at 0.
- You can initialize each individual element of an array after the declaration of the array, or you can place all initial values into a data block surrounded by { and } during the declaration of an array.

- The memory storage taken by an array is determined by the product of the size of the data type and the dimensions of the array.
- A pointer is said to refer to an array when the address of the first element in the array is assigned to the pointer. The address of the first element in an array is also called the start address of the array.
- To assign the start address of an array to a pointer, you can either put the combination of the address-of operator (&) and the first element name of the array, or simply use the array name, on the right side of an assignment operator (=).
- A character array is considered a character string in C if the last element in the array contains a null character (\0).
- The null character (\0) marks the end of a string. C functions, such as printf(), will stop processing the string when the null character is encountered.
- C supports multidimensional arrays, too. A pair of brackets (the array subscript operator—[and]) indicates a dimension.
- The compiler can automatically calculate the memory space needed by an unsized array.

In the next lesson you'll learn more about strings in C.

Q&A

Q Why do you need to use arrays?

A In many cases, you need to declare a set of variables that are of the same data type. Instead of declaring each variable separately, you can declare all variables collectively in the format of an array. Each variable, as an element of the array, can be accessed either through the array element reference or through a pointer that references the array.

Q What is the minimum index in an array?

A In C, the minimum index of a one-dimensional array is 0, which marks the first element of the array. For instance, given an integer array,

```
int array_int[8];
```

the first element of the array is array_int[0].

Likewise, for a multidimensional array, the minimum index of each dimension starts at 0.

Q How do you reference an array by using a pointer?

A You can use a pointer to reference an array by assigning the start address of an array to the pointer. For example, given a pointer variable ptr_ch and a character array array_ch, you can use one of the following statements to reference the array by the pointer:

```
ptr_ch = array_ch;
```

```
ptr_ch = &array_ch[0];
```

Q What can the null character do?

A The null character (\0) in C can be used to mark the end of a string. For instance, the printf() function puts the next character on the screen when the null character is encountered. Also, the null character always returns FALSE in a logical test.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. What does the following statement do?

   ```
   int array_int[4] = {12, 23, 9, 56};
   ```

2. Given an array, int data[3], what's wrong with the following initialization?

   ```
   data[1] = 1;
   data[2] = 2;
   data[3] = 3;
   ```

3. How many dimensions do the following arrays have?
   - char array1[3][19];
   - int array2[];
   - float array3[][8][16];
   - char array4[][80];
4. What's wrong with the following declaration?

   ```
   char list_ch[][] = {
            `A', `a',
            `B', `b',
            `C', `c',
            `D', `d',
            `E', `e'};
   ```

Exercises

1. Given this character array:

   ```
   char array_ch[5] = {`A', `B', `C', `D', `E'};
   ```

write a program to display each element of the array on the screen.

2. Rewrite the program in exercise 1, but this time use a for loop to initialize the character array with `a', `b', `c', `d', and `e', and then print out the value of each element in the array.
3. Given this two-dimensional unsized array:

```
char list_ch[][2] = {
         `1', `a',
         `2', `b',
         `3', `c',
         `4', `d',
         `5', `e',
         `6', `f'};
```

   write a program to measure the total bytes taken by the array, and then print out all elements of the array.
4. Rewrite the program in Listing 12.5. This time put a string of characters, I like C!, on the screen.
5. Given the following array:

```
double list_data[6] = {
         1.12345,
         2.12345,
         3.12345,
         4.12345,
         5.12345};
```

   use the two equivalent ways taught in this lesson to measure the total memory space taken by the array, and then display the results on the screen.