CS102: Arrays

Arrays are used when you are dealing with several related variables of the same type.

Let's say that you want to write a program that lets the user enter 10 integers and then print them out in the reverse order of how were entered. To do this using what has been covered so far, you would have to have 10 different variables to represent the 10 different integers, 10 "scanf" statements to have the user enter values for them, and then 10 "printf" statements to print them out in reverse order. This is clearly too much code for such a simple task. Now imagine if you want to do the same thing with 100 integers or 1000 integers!

Fortunately, C allows arrays. An array can be thought of as a sequence of variables of the same type. To declare an array, you must specify the type, the array name (identifier), and the array **size** (number of values you will store). For example, to declare an array of 10 integers, the declaration may be as follows:

```
int numbers[10];
```

Note that the size of the array appears within brackets after the name of the array. So, this declares an array of 10 integers, and the name of the array is "numbers". The size must be a constant (although some C compilers allow variable-sized arrays).

To access the separate values in this array, you must use the name of the array followed by the **index** of the element you want to access within square brackets. The designers of C decided that the first element of the array has index 0. Therefore, the above array has 10 elements with indexes 0 through 9. To access the first element, you would use "numbers[0]", and to access the tenth (last) element, you would use "numbers[9]". The value inside of the brackets does not have to be a constant; it could be a variable or expression. For instance, if "x" is an integer variable, it is perfectly acceptable to use "numbers[x]" or even "numbers[x*2+1]".

Let's see how we would code the task mentioned above:

```
#include <stdio.h>

#define ARRAYSIZE 10

int main(void)
{
  int x, numbers[ARRAYSIZE];

  for (x = 0; x < ARRAYSIZE; x++)
  {
    printf("Enter number: ");
    scanf("%d", &numbers[x]);
  }
```

```
  for (x = ARRAYSIZE - 1; x >= 0; x--)
    printf("%d\n", numbers[x]);
}
```

There are a couple of things to notice here. First, notice that I chose to use a constant, namely ARRAYSIZE, to represent the number of integers used. (Alternatively, I could have made this a "const int".) The reason to do this (as opposed to typing the number 10 in all locations instead of ARRAYSIZE) is in case you think you might change your mind and want to change it later. For instance, let's say you want to change the program to let the user enter 20 integers and print them out in reverse order. Now, you only have to change 10 to 20 in one location. If you didn't use a constant, you would have to change 10 to 20 in three places (everywhere else that ARRAYSIZE is used in the code).

Next, notice that the first "for" loop loops from 0 to SIZE-1. This is because the array is indexed with these values. Of course, you could have looped from 1 to SIZE, and used the expression "x-1" inside the square brackets.

Now notice that the second "for" loop loops from SIZE-1 down to 0. This is because we are printing out the array in reverse order. Another choice would have been to loop from 1 to SIZE, in which case we would have used the expression "SIZE - x" inside the brackets. Also, since each iteration of this loop contains just one simple statement (the call to the "printf" function), there is no need to enclose it in curly braces, although it is fine to do so if you prefer for readability or consistency.

I have used a '/n' at the end of the "printf" statement in the second loop, so each value will be displayed on its own line. Often, when printing out the values of an array, you will want to print them out on one line separated by spaces or commas. Let's say you want to print out the 10 elements of "numbers" on one line separated by commas. The output loop could then be rewritten as follows:

```
printf("%d", numbers[ARRAYSIZE - 1]);
for (x = ARRAYSIZE - 2; x >= 0; x--)
     printf(", %d", numbers[x]);
printf("\n");
```

Here, we needed two extra statements. The reason is that we don't want a comma printed out before the first element or after the last, and you only want one newline printed out at the end. So, we print out the last element of the array first, then each other element in reverse order with a comma before each, and when the loop ends, we print out a newline once. Notice we are now starting "x" at SIZE-2 instead of SIZE-1 like before, since we have already printed out the last element of the array.

Remember that a variable is really a named memory location. Each type of variable has a certain size, or amount of memory, that it is allotted. On most modern systems, integers are allotted 4 bytes (32 bits). When an array is declared, enough memory is allotted for the entire array. One consecutive slot of memory is allotted. For instance, for an integer

array of size 10 on a system that uses 4 byte integers, 40 bytes will be allotted. Let us say the address of the first value in the array, for example "numbers[0]" in the above program, is located at memory address x. Then the second value will be located at x+4, the third at x+8, etc. More generally, the following expression holds true:

element address = array address + (sizeof(element) * index)

In fact, "sizeof" is actually an operator provided by C! You can use it to determine the size in bytes of a variable type or a variable. For instance, if we are dealing with a system that uses 4 byte integers, "sizeof(int)" will equal 4. Moreover, if "x" is declared to be an integer, than "sizeof(x)" will equal 4 as well.

What happens if you try to access an array with an index that is out of bounds? For example, let's say that "numbers" is an array of 10 integers, as in the above program. That means that the appropriate indices for the array range from 0 to 9. Let's say we try to access "numbers[10]", or even "numbers[100]". The program does not complain; it will access the value at the memory location given by the above formula! However, if we are reading the value, it will be unpredictable, and if we are assigning a value, we will be writing to memory that isn't owned by this array. It will cause an error sooner or later unless you are extremely lucky! Sometimes this type of bug is difficult to track down.

Here is an example that reads in text one character at a time until an EOF is encountered and keeps track of how many times each digit (0 through 9) occurs.

```
#include <stdio.h>

int main(void)
{
  char c;
  int ndigit[10];
  int x;

  for (x = 0; x <= 9; x++)
    ndigit[x] = 0;

  while ((c = getchar()) != EOF)
  {
    if ((c >= '0') && (c <= '9'))
      ++ndigit[c - '0'];
  }

  for (x = 0; x <= 9; x++)
    printf("count of %d is %d\n", x, ndigit[x]);

  return 0;
}
```

There are a few things to notice here. First, notice that we have to initialize all the values in the array. Just like with simple variables, if the array values are not initialized, they are unpredictable. Why do we have to initialize the array here but not in the first program? Because in the first program, we have the user enter all the values, and these values are assigned directly to the array. In this program, we are counting digits, and incrementing the values in the array but never assigning them directly to a specific value. We need all the values to start at 0 to have this program work correctly.

Also note that a single value in an array can be assigned a value just like any other variable. Just specify the array with the index to the left of an equal sign (the assignment operator). The following are valid expressions (assuming numbers is an array of integers):

```
numbers[5] = 820;
numbers[x] = numbers[x-1] + numbers[x-2];
```

It is NOT acceptable to assign one array to another array. For example, let's say that numbers and elements are both arrays of 10 integers. It is NOT allowable to say:

```
elements = numbers;
```

Instead, you have to copy one element (value) at a time.

Looking back at the program, notice that when we read a digit, we can check if the character 'c' represents a digit by checking if it is greater than or equal to the character '0' and less than or equal to the character '9'. This is because the ASCII codes for the digits 0 through 9 are 48 through 57 respectively. It is for this reason also that we can subtract the character '0' from the character 'c' to find the appropriate index in the array. Characters used in expressions are treated as their ASCII values, so '0' - '0' = 0, '1' - '0' = 1, etc.

It is also possible to initialize an array in C as it is declared. To do this, the starting values of the array must be specified within curly braces and separated by commas. For example:

```
int numbers[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

is a valid declaration which initialized the array.

So far, all the arrays we have looked at have been one-dimensional arrays, which could be thought of as a single sequence of values. We can also declare **two-dimensional arrays**, which are often used to represent grids, game boards, tables, etc. By convention, the first dimension (or index) is used to represent the rows, numbered top to bottom starting at 0, and the second dimension (or index) is used to represent the columns, numbered left to right starting at 0. You don't have to use this convention, but almost all programmers do, so if you don't, you will likely get confused when looking at other programmers' code and they will get confused when they look at yours.

For example, we could have used a two-dimensional array of integers to store the multiplication table we created with nested loops when we covered nested loops. The array might have been declared and filled in as follows:

```
int mult_table[10][10];

for (x = 0; x <= 9; x++)
  for (y = 0; y <= 9; y++)
    mult_table[x][y] = x*y;
```

Of course, the variables "x" and "y" would also have to be declared.

The code to display the matrix (without row and column headers, and with values right-justified) could look as follows:

```
for (x = 0; x <= 9; x++)
{
  for (y = 0; y <= 9; y++)
    printf("%3d", mult_table[x][y]);
  printf("\n");
}
```

Note that we are specifying a width for each number so that the numbers that should be in the same columns will line up correctly. The inner loop prints out one row of the matrix, and an end-of-line character is printed after each row.

One possible use for two-dimensional arrays is to represent game boards for games such as Checkers. The declaration for such an array may be as follows:

```
char board[8][8];
```

Possible values for each slot in the board might be 'r' for a normal red checker, 'R' for a red king, 'b' for a normal black checker, 'B' for a black kind, and some other default value to represent empty spaces. Of course, if you prefer, you could also use an integer array, and use specific integers to represent the different types of possible pieces.

You can also initialize a two dimensional array along with the declaration. Let's say you want to declare the following two-dimensional array of integers:

```
00    01    02
10    11    12
20    21    22
30    31    32
40    41    42
```

This array has 5 rows and 3 columns. The 10's digit of each number is the row, and the 1's digit of each number is the column. This table could be declared and initialized as follows:

```
int table [5][3] =
{{00, 01, 02},
 {10, 11, 12},
 {20, 21, 22},
 {30, 31, 32},
 {40, 41, 42}};
```

Of course, C generally ignores white space, and you don't have to write each row on its own line like I did here. Also, to C, 00, 01, and 02 are equivalent to 0, 1, and 2. These are the values that are stored, and when you print the numbers out, you won't see the 0 in the 10's column (unless you use the 0 flag and a width of 2 in your "printf" statement).

If you want to use "scanf" to have the user enter a value for a slot in a two-dimensional array, the call may look as follows:

```
scanf("%d", &table[x][y]);
```

Of course, the variables "x" and "y" here are assumed to be integers in the appropriate ranges, but you could also use constants or other expressions. Don't forget the '&' symbol!

How is a two-dimensional array stored in memory? The entire two-dimensional array has its values stored consecutively. The values in the row 0 (top row) are stored first (from left to right), then the values in row 1 (the second row), etc. So, the element in a particular row and column is stored at the address given by the following formula:

element address = array address + (sizeof(element) * (row index * number of columns + column index))

Why? The row index multiplied by the number of columns is the number of values in the two-dimensional array in all rows above (i.e., previous to) the current row, and they are stored first. The column index is the number of values in the current row of the two-dimensional array to the left of (i.e., previous to) the current element, and they are stored first. So the expression "row index * number of columns + column index" is the number of elements in the two-dimensional array stored before the current element. Each element takes up sizeof(element) bytes. So the formula takes the starting address of the array and adds the number of bytes that is the offset to the current element.

It is possible to have arrays with more than two dimensions. You can have three-dimensional arrays, four-dimensional arrays, etc.