C++ Arrays

Source: Tutorials Point C++ Arrays

C++ provides a data structure, **the array**, which stores a fixed-size, sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The compiler will reserve a section of the memory to hold all of the data that is defined for the array. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type float, use this statement −

```
float balance[10];
```

Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows –

```
float balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. If any values are not defined, they will be left empty. For example, the 4th and 5th elements of the array will be empty if only 3 values were provided.

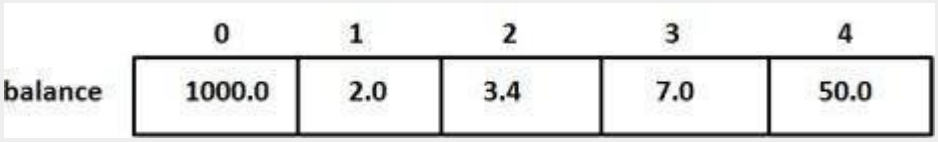
If you omit the size of the array, an array just big enough to hold the values defined is created. Therefore, if you write –

```
float balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array a value of 50.0. There is a 4 inside of the brackets because the base index used to define the individual elements starts at 0. This means the highest index is always going to be one less than the maximum size of the array. Shown below is the pictorial representation of the same array we discussed above.



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
float salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to the salary variable. Below is an example that declares, assigns values, and accesses elements to an array.

```
int setup () {
   Serial.begin(9600);
   int n[ 10 ]; // n is an array of 10 integers
   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // set element at location i to i + 100
   }
   Serial.print(“Element        ”);
   Serial.println("Value");
   // output each array element's value
   for ( int i = 0; i < 10; i++ ) {
      Serial.print(“      “);
      Serial.print(i);
      Serial.print(“           “);
      Serial.println(n[ i ])
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

C++ Passing Arrays to Functions

C++ does not allow you to pass an entire array as an argument to a function. However, you can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you will have to declare a pointer to the array as a formal parameter in the function declaration. We will quickly review function declarations so that you understand what is being done here.

C++ Function Declarations – *A Review*

When you need to create a custom function that will be used in your program, it must be declared and/or defined. The format for a declaration is shown below.

[Data type of output parameter] [Function Name] ([Data Types of input parameters]) {}

The first part of the function declaration is the data type of the output parameter. This is the type of value that will be returned from the function. This can be an int, float, pointer, or etc. If there is no value to be returned, you will use the void data type. You only need to indicate the data type, as there is no need to name the output.

The second part of the function declaration is the name of the function. This is what is used to call the function. A general convention for naming functions is to have the first word in lowercase and capitalize the first letter of the second word.

The third part of the function declaration are the data types of the input parameters. Any variables or values that the function will need to use must be defined here. This is related to the concept of the scope of a function because copies of those variables or values are created when the input parameters are defined. They are not readily available to the custom functions unless the variables or values have a global scope. It is generally not recommended to use global variables. If there are no inputs, you will have nothing inside of the parentheses.

The final part of a function declaration are the brackets. This is where the function is defined and the code to be performed are placed here. When programming within the Arduino IDE, you will not need to worry about where a function is declared or defined as long as it is located within the main ".ino" file. If a function is called near the beginning of the program and it is defined much later in the code, the compiler will not have an issue with finding the function and will not give a scope error. For this reason, you only need to declare and define a function a single time rather than provide a declaration as part of the header (.h) file, as is the case within traditional C++ IDEs, like Eclipse. Here is an example of a function declaration.

```
void myFunction(int input1, float input2, char input3) {}
```

This function is named myFunction and will return no output. It has three input values that are of three different types. The inputs are named input1, input2, and input3 and are only available to be used within the function. There are used more as place holders for the variable or value that you intend to use in the function.

If you are trying to call this example function, you will need to use the following code with the assumption that x is defined as an int, y is defined as a float, and z is defined as a char.

```
myFunction(x, y, z);
```

Review - C++ Arguments versus Parameters

Now that we have covered function declarations, it is important to note the distinction between arguments and parameters. A **parameter** is defined in the function declaration and indicates the type of data that is used. The parameter name is usually a placeholder

to represent the kind of value or variable that the function works with. In the function declaration of myFunction the parameters are input1, input2, and input3.

An **argument** is the actual value or variable being used when a function is called. Arguments are passed to a function and are modified by the function code. From the example above, x, y, and z are the three arguments being sent to myFunction.

Defining A Pointer Parameter

After all of that review, we can now introduce how to define a pointer parameter for a function declaration. There are three ways to do this and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

This form is the general form that has been used the longest and is distinct because of the use of the * to indicate it is a pointer. This is also the most outdated method and should not be used.

```
void myFunction(int *param) {
   .
   .
   .
}
```

Way-2

This is the modern way to define a pointer parameter because it is clear that an array is being used and the intended size of that array is known.

```
void myFunction(int param[10]) {
   .
   .
   .
}
```

Way-3

The final method is to define the pointer parameter as an unsized array that can work with any size array argument that is passed to it.

```
void myFunction(int param[]) {
   .
   .
   .
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows –

```
float getAverage(int arr[], int size) {
  int i, sum = 0;
  float avg;
   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }
   avg = float(sum) / size;
   return avg;
}
```

Now, let us call the above function as follows –

```
#include <iostream>
using namespace std;
// function declaration:
float getAverage(int arr[], int size);
int main () {
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   float avg;
   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;
   // output the returned value
   Serial.print("Average value is: ");
   Serial.print(avg);
```

```
   return 0;
}
```

When the above code is compiled together and executed, it produces the following result

```
Average value is: 214.4
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

If you want to return an array as an output parameter, it is not allowed and you must return a pointer to an array instead. You can do this by defining the data type and using an * to indicate it is a pointer as shown below.

```
int * myFunction() {}
```

If the output array is created within the function, it is suggested to define it as a static array so that the pointer being returned is not the address of a local variable that could disappear once the program is out of the scope of the function. What this is referring to is the possibility that the program will delete all local variables after it returns from the function and the pointer is now going to a location that no longer has the correct values. By using static, it will preserve the data at the address defined by the pointer.

The example shown below will return a pointer to an array of 10 random numbers.

```
int * getRandom() {
  static int r[10];
  srand((unsigned)time(NULL);
   for (int i = 0; i < 10; ++i) {
      r[i] = rand();
      cout << r[i] << endl;
   }
  return r;
}
```

You can also define a pointer as a variable and initialize its value to be the address of an array. The pointer will need to be the same data type as the array. The example shown below works with the getRandom() function.

```
int *p;
p = getRandom();
```

By modifying the pointer, you can access the different elements of the array instead of using the index value. The code below will modify the 2nd element of the array of random numbers to be 20 and the 7th element to be 100.

```
*(p + 1) = 20;
*(p + 6) = 100;
```

Using PROGMEM

Normally, all arrays are created and stored within SRAM. The size of the array and its contents can always change as the program is running. For the cases where the data in an array is constant and it takes up a lot of space, we can save it all in flash program memory. This will free up memory in SRAM for the program to use and helps to keep the code organized.

In order to do this with the Arduino IDE, we will be using the pgmspace library that is available for the AVR architecture only. It provides definitions for the different data types that can be saved into flash program memory and all of the functions needed to interact (read, find, compare, etc) with the data saved there. To use the library, we need to add the following line of code to the very top of our program.

```
#include <avr/pgmspace.h>
```

Once it has been included, we can start defining the data to be saved to flash program memory using the variable modifier "PROGMEM". It can be added before the data type of the variable or after the variable name and it indicates that this variable (arrays included) is to be saved in flash program memory. Below are some examples of how to use it.

```
const dataType variableName[] PROGMEM = {};  // use this form
const PROGMEM  dataType  variableName[] = {}; // or this form
const dataType PROGMEM variableName[] = {};   // not this one
```

After the data has been defined and saved, you will need to use the `pgm_read_byte()` function to access the data. There are several functions for each of the data types and you need to provide the address that the variable is located at. If you are accessing data

from an array, you can use the `pgm_read_byte_near()` function that will add a value to the starting address of the array to get your desired index value. For example, if you needed to get the 5th element from the array saved in flash program memory, the code would look like this.

```
X = pgm_read_byte_near(arrayName + 4);
```

For more information about the functions and the pgmspace library, click here. Here is a nice example of how to use PROGMEM and the pgmspace library. It will print out the values and strings saved in the charSet and signMessage arrays.

```
#include <avr/pgmspace.h>


// save some unsigned ints
const PROGMEM  uint16_t charSet[]  = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM  = {"I AM PREDATOR,  UNSEEN COMBATANT.
CREATED BY THE UNITED STATES DEPART"};

unsigned int displayInt;
int k;    // counter variable
char myChar;

void setup() {
  Serial.begin(9600);
  while (!Serial);

  // put your setup code here, to run once:
  // read back a 2-byte int
  for (k = 0; k < 5; k++)
  {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();

  // read back a char
  int len = strlen_P(signMessage);
  for (k = 0; k < len; k++)
  {
    myChar =  pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }

  Serial.println();
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Running Averages

To finish up our discussions of arrays, we will be covering how to implement a running average in our program. It is very useful for inputs that have slight variations such as the IR sensor or an ADC input and will provide an approximate value with more stability. Depending on the number of samples used to calculate the running average, the accuracy of the final value will change. If the data can be obtained quickly, it is best to have a large sample size.

The example program below is reading from an analog pin and computing the running average for a sample size of 200. Each sample is added to the previous average calculated and this repeats until the total sample size is reached.

```
int readADC(int sensorPin){
int n = 200;          // number of ADC samples
int x_i;              // ADC input for sample i
float A_1;            // current  i running average
float A_0;            // previous i-1 running average

// rapid calculation method http://en.wikipedia.org/wiki/Standard_deviation
A_0 = 0;
for (int i=1; i <= n; i++){
    x_i = analogRead(sensorPin);
    A_1 = A_0 + (x_i - A_0)/i ;
    A_0 = A_1;
}

// Serial.print(", mean = ");
// Serial.println(A_1);

return (int(A_1));
}
```

Review Questions

1. The following code will create an array of how many elements? int testarray[14];
2. Write the code that will assign the value of the 3rd element of the array called balance to a variable called change
3. Can an array be defined without an array size?
4. The action of sending variable values to be used in a function is called what?
5. The void data type is used for what?

6. Write one of the ways to define a pointer as an input parameter for the function testFunction()
7. Write the function definition for a function called returnPointer that has no input parameters and returns a pointer to an integer
8. Where are arrays normally saved to?
9. What function from the pgmspace library should be used to read a value from an array?
10. What type of loop is required to implement a running average?

Answers

1. 14 elements
2. change = balance[2];
3. No, all arrays must have a defined size.
4. Passing arguments (may need to remove "values" from the question to be less confusing)
5. Void is used for defining that a function does not have an output returned
6. Void testFunction(int *pointer) {} , void testFunction(int pointer[]) {}, void testFunction(int pointer[10]) {}
7. int *pointer returnPointer() {}
8. SRAM
9. pgm_read_byte_near()
10. for loop