**Arrays and Strings**

# Arrays

So far we have used variables to store values in memory for later reuse. We are now exploring a means to store *multiple* values together as one unit, the *array.*

Unlike other languages arrays in c are not Objects!!!!!

An array is a fixed number of *elements* of the same type stored sequentially in memory. Therefore, an integer array holds some number of integers, a character array holds some number of characters, and so on. The size of the array is referred to as its *dimension.* To declare an array in C++, we write the following:

*type arrayName*[*dimension*];

To declare an integer array named arr of four elements, we write int arr[4];

The elements of an array can be accessed by using an *index* into the array. Arrays in C++ are zero-indexed, so the first element has an index of 0. So, to access the third element in arr, we write arr[2]; The value returned can then be used just like any other integer.

Like normal variables, the elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program. There are several ways to initialize the array. One way is to declare the array and then initialize some or all of the elements:

int arr[4];

arr[0] = 6;

arr[1] =0;

arr[2] =9;

arr[3] =6;

Another way is to initialize some or all of the values at the time of declaration:

int arr[4] = { 6, 0, 9, 6};

Sometimes it is more convenient to leave out the size of the array and let the compiler determine the array's size for us, based on how many elements we give it:

int arr[] = { 6, 0, 9, 6, 2, 0, 1, 1};

Here, the compiler will create an integer array of dimension 8.

The array can also be initialized with values that are not known beforehand:

#include <iostream>

using namespace std; 3

int main() {

int arr[4];

cout << “Please enter 4 integers:“ << endl;

for(int i = 0; i < 4; i++)

cin >> arr[i];

|  |  |
| --- | --- |
|  | cout << “Values in array are now:“; |
|  | for(int i = 0; i < 4; i++)  cout << “ “ << arr[i]; |
|  | cout << endl; |
| } | return 0; |

Note that when accessing an array the index given must be a positive integer from 0 to n-1, where n is the dimension of the array. The index itself may be directly provided, derived from a variable, or computed from an expression:

arr[5];

arr[i];

arr[i+3];

Arrays can also be passed as arguments to functions. When declaring the function, simply specify the array as a parameter, without a dimension. The array can then be used as normal within the function. For example:

#include <iostream>

using namespace std;

long sum(const int array[], const int length) {

long sum = 0;

for(int i = 0; i < length; i++)

{

sum += array[i];

}

return sum;

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7};

cout << "Sum: " << sum(arr, 7) << endl;

return 0;

}

The function sum takes a constant integer array and a constant integer length as its arguments and adds up length elements in the array. It then returns the sum, and the program prints out Sum: 28.

It is important to note that arrays are *passed by reference* and so any changes made to the array within the function will be observed in the calling scope.

**Exercise : Write a function that returns the product of an array of 10 floats**

C++ also supports the creation of multidimensional arrays, through the addition of more than one set of brackets. Thus, a two-dimensional array may be created by the following:

*type arrayName*[*dimension1*][*dimension2*];

The array will have *dimension1* x *dimension2* elements of the same type and can be thought of as an array of arrays. The first index indicates which of *dimension1* subarrays to access, and then the second index accesses one of *dimension2* elements within that subarray. Initialization and access thus work similarly to the one-dimensional case:

#include <iostream>

int main() {

int twoDimArray[2][4];

twoDimArray[0][0] = 6;

twoDimArray[0][1] = 0;

twoDimArray[0][2] = 9;

twoDimArray[0][3] = 6;

twoDimArray[1][0] = 2;

twoDimArray[1][1] = 0;

twoDimArray[1][2] = 1;

twoDimArray[1][3] = 1;

for (int row = 0; row < 2; row++)

{

for (int col = 0; col < 4; col++)

{

std::cout << twoDimArray[row][col];

}

std::cout << std::endl;

}

std::cout << std::endl;

return 0;

}

**Exercise: write code that outputs the sum of the elements**

**of each row.**

The array can also be initialized at declaration in the following ways:

int twoDimArray[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };

int twoDimArray[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } };

Note that dimensions must *always* be provided when initializing multidimensional arrays, as it is otherwise impossible for the compiler to determine what the intended element partitioning is. For the same reason, when multidimensional arrays are specified as arguments to functions, all dimensions but the first *must* be provided (the first dimension is optional), as in the following:

int aFunction(int arr[][4]) { … }

Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory. Declaring int arr[2][4]; is the same thing as declaring int arr[8];.

**Exercise: Write code for a function that searches a 2 dimemsional array of integers for a particular value and returns true if the value is found.**

# 2 Strings

String literals such as “Hello, world!” are actually represented by C++ as a sequence of characters in memory. In other words, a string is simply a character array and can be manipulated as such.

Consider the following program:

#include <iostream>

using namespace std;

int main() {

char helloworld[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ',

'w', 'o', 'r', 'l', 'd', '!', '\0' };

cout <<helloworld<<endl;

return 0;

}

**Exercise : Remove the ‘\0’ from the character array declaration and run the program. What Happens? Add**

**char helloworld2[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ',**

**'w', 'o', 'r', 'l', 'd', '!', '\0' };**

**On line 7. Run the code again. What is outputted?**

This program prints Hello, world! Note that the character array helloworld ends with a special character known as the *null character*. This character is used to indicate the end of the string.

Character arrays can also be initialized using string literals. In this case, no null character is needed, as the compiler will automatically insert one:

char helloworld[] = “Hello, world!”;

The individual characters in a string can be manipulated either directly by the programmer or by using special functions provided by the C/C++ libraries. These can be included in a program through the use of the #include directive. Of particular note are the following:

* + cctype (ctype.h): character handling
  + cstdio (stdio.h): input/output operations
  + cstdlib (stdlib.h): general utilities
  + cstring (string.h): string manipulation Here is an example to illustrate the cctype library:

#include <iostream>

#include <cctype>

using namespace std;

int main()

{

char messyString[] = "t6H0I9s6.iS.999a9.STRING";

char current = messyString[0];

for(int i = 0; current != '\0'; current = messyString[++i])

{

if(isalpha(current))

{

if (isupper(current))

{

cout << tolower(current);

}

else {

cout << current;

}

}

else if(ispunct(current)) cout << ' ';

}

cout << endl;

return 0;

}

**Exercise: The above program does not output the correct string. Fix it!**

This example uses the isalpha, isupper, ispunct, and tolower functions from the cctype library. The is- functions check whether a given character is an alphabetic character, an uppercase letter, or a punctuation character, respectively. These functions return a Boolean value of either true or false. The tolower function converts a given character to lowercase.

The for loop beginning at line 9 takes each successive character from messyString until it reaches the null character. On each iteration, if the current character is alphabetic and uppercase, it is converted to lowercase and then displayed. If it is already lowercase it is simply displayed. If the character is a punctuation mark, a space is displayed. All other characters are ignored. The resulting output is this is a string.

Here is an example to illustrate the cstring library:

1. #include <iostream>
2. #include <cstring>
3. using namespace std; 4
4. int main() {
5. char fragment1[] = "I'm a s";
6. char fragment2[] = "tring!";
7. char fragment3[20];
8. char finalString[20] = ""; 10
9. strcpy(fragment3, fragment1);
10. strcat(finalString, fragment3);
11. strcat(finalString, fragment2);

14

1. cout << finalString;
2. return 0;
3. }

This example creates and initializes two strings, fragment1 and fragment2. fragment3 is declared but not initialized. finalString is partially initialized (with just the null character).

fragment1 is copied into fragment3 using strcpy, in effect initializing fragment3 to I'm a s. strcat is then used to concatenate fragment3 onto finalString (the function overwrites the existing null character), thereby giving finalString the same contents as fragment3. Then strcat is used again to concatenate fragment2 onto finalString. finalString is displayed, giving I'm a string!.