C++ arrays guideline

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 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 double, use this statement: double balance[10];| Initializing Arrays: You can initialize C++ array elements either one by one or using a single statement as follows: double balance[5] = {1000. 0, 2. 0, 3. 4, 17. 0, 50. 0};| The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

Following is an example to assign a single element of the array: If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write: double 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 element number 5th in the array a value of 50. 0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index.

Following is the pictorial representaion 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: double salary = balance[9];| The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. eclaration, assignment and accessing arrays: #include using namespace std; #include using std:: setw; int main (){ 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 } cout << " Element" << setw(13) << " Value" << endl; // output each array element's value for (int j = 0; j ; 10; j++) { cout ;; setw(7);; j ;; setw(13) ;; n[j] ;; endl; } return 0;}| This program makes use setw() function to format the output. When the above code is compiled and executed, it produces following result: Element Value 0 100 1 101 2 102 3 103 4 104 5 105 6 106 7 107 8 108 9 109| sequence: Copy decl-specifier identifier [constant-expression] decl-specifier identifier [] decl-specifier identifer [][constant-expression] . . . decl-specifier identifier [constant-expression] [constant-expression] . . . 1. The declaration specifier: * An optional storage class specifier. * Optional const and/or volatile specifiers. The type name of the elements of the array. 2. The declarator: * The identifier. * A constant expression of integral type enclosed in brackets, []. If multiple dimensions are declared using additional brackets, the constant expression may be omitted on the first set of brackets. * Optional additional brackets enclosing constant expressions. 3. An optional initializer. See Initializers. The number of elements in the array is given by the constant expression. The first element in the array is the 0th element, and the last element is the (n-1) element, where n is the number of elements the array can contain. The constant-expression must be of an integral type and must be greater than 0.

A zero-sized array is legal only when the array is the last field in a struct or union and when the Microsoft extensions (/Ze) are enabled. The following example shows how to define an array at run time: Copy // arrays. cpp // compile with: /EHsc #include ; iostream; int main() { using namespace std; int size = 3, i = 0; int* myarr = new int[size]; for (i = 0 ; i ; size ; i++) myarr[i] = 10; for (i = 0 ; i ; size ; i++) printf_s(" myarr[%d] = %d
", i, myarr[i]); delete [] myarr; } Arrays are derived types and can therefore be constructed from any other derived or fundamental type except functions, references, and void. Arrays constructed from other arrays are multidimensional arrays.

These multidimensional arrays are specified by placing multiple bracketed constant expressions in sequence. For example, consider this declaration: Copy int i2[5][7]; It specifies an array of type int, conceptually arranged in a two-dimensional matrix of five rows and seven columns, as shown in the following figure: Conceptual Layout of Multidimensional Array In declarations of multidimensioned arrays that have an initializer list (as described in

Initializers), the constant expression that specifies the bounds for the first dimension can be omitted. For example: Copy // arrays2. cpp // compile with: /c const int cMarkets = 4; // Declare a float that represents the transportation costs. ouble TransportCosts[][cMarkets] = { { 32. 19, 47. 29, 31. 99, 19. 11 }, { 11. 29, 22. 49, 33. 47, 17. 29 }, { 41. 97, 22. 09, 9. 76, 22. 55 } }; The preceding declaration defines an array that is three rows by four columns. The rows represent factories and the columns represent markets to which the factories ship. The values are the transportation costs from the factories to the markets. The first dimension of the array is left out, but the compiler fills it in by examining the initializer. Topics in this section: * Using Arrays * Arrays in Expressions * Interpretation of Subscript Operator * Indirection on Array Types * Ordering of C++ Arrays Example

The technique of omitting the bounds specification for the first dimension of a multidimensional array can also be used in function declarations as follows: Copy // multidimensional_arrays. cpp // compile with: /EHsc // arguments: 3 #include ; limits; // Includes DBL_MAX #include ; iostream; const int cMkts = 4, cFacts = 2; // Declare a float that represents the transportation costs double TransportCosts[][cMkts] = { { 32. 19, 47. 29, 31. 99, 19. 11 }, { 11. 29, 22. 49, 33. 47, 17. 29 }, { 41. 97, 22. 09, 9. 76, 22. 55 } }; // Calculate size of unspecified dimension const int cFactories = sizeof TransportCosts / sizeof(double[cMkts]); double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts); sing namespace std; int main(int argc, char *argv[]) { double MinCost; if (argv[1] == 0) { cout ;; " You must specify the number of markets. " ;; endl; exit(0); } MinCost = FindMinToMkt(*argv[1] - '0', TransportCosts, cFacts); cout ;; " The minimum cost to Market " ;; argv[1] ;; " is: " ;; MinCost ;; " "; } double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) { double MinCost = DBL_MAX; for(int i = 0; i ; cFacts; ++i) MinCost = (MinCost ; TransportCosts[i][Mkt]) ? MinCost : TransportCosts[i][Mkt]; return MinCost; }] This article is part of our on-going C programming series.

There are times while writing C code, you may want to store multiple items of same type as contiguous bytes in memory so that searching and sorting of items becomes easy. For example: 1. Storing a string that contains series of characters. Like storing a name in memory. 2. Storing multiple strings. Like storing multiple names. C programming language provides the concept of arrays to help you with these scenarios. 1. What is an Array? An array is a collection of same type of elements which are sheltered under a common name. An array can be visualised as a row in a table, whose each successive block can be thought of as memory bytes containing one element.

Look at the figure below : An Array of four elements: +===+ | elem1 | elem2 | elem3 | elem4 | +===+ The number of 8 bit bytes that each element occupies depends on the type of array. If type of array is ‘ char’ then it means the array stores character elements. Since each character occupies one byte so elements of a character array occupy one byte each. 2. How to Define an Array? An array is defined as following : ; type-of-array; ; name-of-array; [; number of elements in array;]; * type-of-array: It is the type of elements that an array stores.

If array stores character elements then type of array is ‘ char’. If array stores integer elements then type of array is ‘ int’. Besides these native types, if type of elements in array is structure objects then type of array becomes the structure. * name-of-array: This is the name that is given to array. It can be any string but it is usually suggested that some can of standard should be followed while naming arrays. At least the name should be in context with what is being stored in the array. * [number of elements]: This value in subscripts [] indicates the number of elements the array stores. For example, an array of five characters can be defined as : char arr[5]; 3.

How to Initialize an Array? An array can be initialized in many ways as shown in the code-snippets below. Initializing each element separately. For example : int arr[10]; int i = 0; for(i= 0; i; sizeof(arr); i++) { arr[i] = i; // Initializing each element seperately } Initializing array at the time of declaration. For example : int arr[] = {'1','2','3','4','5'}; In the above example an array of five integers is declared. Note that since we are initializing at the time of declaration so there is no need to mention any value in the subscripts []. The size will automatically be calculated from the number of values. In this case, the size will be 5.

Initializing array with a string (Method 1): Strings in C language are nothing but a series of characters followed by a null byte. So to store a string, we need an array of characters followed by a null byte. This makes the initialization of strings a bit different. Let us take a look : Since strings are nothing but a series of characters so the array containing a string will be containing characters char arr[] = {'c','o','d','e',''}; In the above declaration/initialization, we have initialized array with a series of character followed by a ‘? (null) byte. The null byte is required as a terminating byte when string is read as a whole. Initializing array with a string (Method 2): har arr[] = " code"; Here we neither require to explicitly wrap single quotes around each character nor write a null character. The double quotes do the trick for us. 4. Accessing Values in an Array Now we know how to declare and initialize an array. Lets understand, how to access array elements. An array element is accessed as : int arr[10]; int i = 0; for(i= 0; i; sizeof(arr); i++) { arr[i] = i; // Initializing each element separately } int j = arr[5]; // Accessing the 5th element of integer array arr and assigning its value to integer 'j'. As we can see above, the 5th element of array is accessed as ‘ arr[5]‘. Note that for an array declared as int arr[5].

The five values are represented as: arr[0] arr[1] arr[2] arr[3] arr[4] and not arr[1] arr[2] arr[3] arr[4] arr[5] The first element of array always has a subscript of ’0? 5. Array of Structures The following program gives a brief idea of how to declare, initialize and use array of structures. #include; stdio. h; struct st{ int a; char c; }; int main() { struct st st_arr[3]; // Declare an array of 3 structure objects struct st st_obj0; // first structure object st_obj0. a = 0; st_obj0. c = 'a'; struct st st_obj1; //Second structure object st_obj1. a = 1; st_obj1. c = 'b'; struct st st_obj2; // Third structure object st_obj2. a = 2; st_obj2. c = 'c'; t_arr[0] = st_obj0; // Initializing first element of array with first structure object st_arr[1] = st_obj1; // Initializing second element of array with second structure object st_arr[2] = st_obj2; // Initializing third element of array with third structure object printf("

First Element of array has values of a = [%d] and c = [%c]

", st_arr[0]. a, st_arr[0]. c); printf("

Second Element of array has values of a = [%d] and c = [%c]

", st_arr[1]. a, st_arr[1]. c); printf("

Third Element of array has values of a = [%d] and c = [%c]

", st_arr[2]. a, st_arr[2]. c); return 0; } The output of the above program comes out to be : $. /strucarr

First Element of array has values of a = [0] and c = [a] Second Element of array has values of a = [1] and c = [b] Third Element of array has values of a = [2] and c = [c] 6. Array of Char Pointers The following program gives a brief Idea of how to declare an array of char pointers : #include; stdio. h; int main() { // Declaring/Initializing three characters pointers char *ptr1 = " Himanshu"; char *ptr2 = " Arora"; char *ptr3 = " TheGeekStuff"; //Declaring an array of 3 char pointers char* arr[3]; // Initializing the array with values arr[0] = ptr1; arr[1] = ptr2; arr[2] = ptr3; //Printing the values stored in array printf("

[%s]

", arr[0]); printf("

[%s]

", arr[1]); rintf("

[%s]

", arr[2]); return 0; } The output of the above program is : $. /charptrarr [Himanshu] [Arora] [TheGeekStuff] 7. Pointer to Arrays Pointers in C Programming language is very powerful. Combining pointers with arrays can be very helpful in certain situations. As to any kind of data type, we can have pointers to arrays also. A pointer to array is declared as : ; data type; (*; name of ptr;)[; an integer;] For example : int(*ptr)[5]; The above example declares a pointer ptr to an array of 5 integers. Lets look at a small program for demonstrating this : #include; stdio. h; int main(void) { char arr[3]; char(*ptr)[3]; rr[0] = 'a'; arr[1] = 'b'; arr[2] = 'c'; ptr = ; arr; return 0; } In the above program, we declared and initialized an array ‘ arr’ and then declared a pointer ‘ ptr’ to an array of 3 characters. Then we initialized ptr with the address of array ‘ arr’. 8. Static vs Dynamic Arrays Static arrays are the ones that reside on stack. Like : char arr[10]; Dynamic arrays is a popular name given to a series of bytes allocated on heap. this is achieved through malloc() function. Like : char *ptr = (char*)malloc(10); The above line allocates a memory of 10 bytes on heap and we have taken the starting address of this series of bytes in a character pointer ptr.

Static arrays are used when we know the amount of bytes in array at compile time while the dynamic array is used where we come to know about the size on run time. 9. Decomposing Array into Pointers Internally, arrays aren’t treated specially, they are decomposed into pointers and operated there-on. For example an array like : char arr[10]; When accessed like : arr[4] = 'e'; is decomposed as : *(arr + 4) = 'e' So we see above that the same old pointers techniques are used while accessing array elements. 10. Character Arrays and Strings Mostly new programmers get confused between character arrays and strings. Well, there is a very thin line between the two. This thin line only comprises of a null character ‘? If this is present after a series of characters in an array, then that array becomes a string. This is an array: char arr[] = {'a', 'b', 'c'}; This is a string: char arr[] = {'a', 'b', 'c', ''}; Note : A string can be printed through %s format specifier in printf() while an printing an array through %s specifier in printf() is a wrong practice. 11. Bi-dimensional and Multi-dimensional Arrays The type of array we discussed until now is single dimensional arrays. As we see earlier, we can store a set of characters or a string in a single dimensional array. What if we want to store multiple strings in an array. Well, that wont be possible using single dimensional arrays. We need to use bi-dimensional arrays in this case.

Something like : char arr[5][10]; The above declaration can be thought of as 5 rows and 10 columns. Where each row may contain a different name and columns may limit the number of characters in the name. So we can store 5 different names with max length of 10 characters each. Similarly, what if we want to store different names and their corresponding addresses also. Well this requirement cannot be catered even by bi-dimensional arrays. In this case we need tri-dimensional (or multi-dimensional in general) arrays. So we need something like : char arr[5][10][50]; So we can have 5 names with max capacity of 10 characters for names and 50 characters for corresponding addresses.

Since this is an advanced topic, So we won’t go into practical details here. 12. A Simple C Program using Arrays Consider this simple program that copies a string into an array and then changes one of its characters : #include; stdio. h; #include; string. h; int main(void) { char arr[4];// for accommodating 3 characters and one null '' byte. char *ptr = " abc"; //a string containing 'a', 'b', 'c', '' memset(arr, '', sizeof(arr)); //reset all the bytes so that none of the byte contains any junk value strncpy(arr, ptr, sizeof("

abc")); // Copy the string " abc" into the array arr printf("

%s

", arr); //print the array as string rr[0] = 'p'; // change the first character in the array printf("

%s

", arr);//again print the array as string return 0; } I think the program is self explanatory as I have added plenty of comments. The output of the above program is : $. /array_pointer abc pbc So we see that we successfully copied the string into array and then changed the first character in the array. 13. No Array Bound Check in a C Program What is array bound check? Well this is the check for boundaries of array declared. For example : char arr[5]; The above array ‘ arr’ consumes 5 bytes on stack and through code we can access these bytes using : arr[0], arr[1], arr[2], arr[3], arr[4]

Now, C provides open power to the programmer to write any index value in [] of an array. This is where we say that no array bound check is there in C. SO, misusing this power, we can access arr[-1] and also arr[6] or any other illegal location. Since these bytes are on stack, so by doing this we end up messing with other variables on stack. Consider the following example : #include; stdio. h; unsigned int count = 1; int main(void) { int b = 10; int a[3]; a[0] = 1; a[1] = 2; a[2] = 3; printf("

b = %d

", b); a[3] = 12; printf("

b = %d

", b); return 0; } In the above example, we have declared an array of 3 integers but try to access the location arr[3] (which is illegal but doable in C) and change the value kept there.

But, we end up messing with the value of variable ‘ b’. Cant believe it? , check the following output . We see that value of b changes from 10 to 12. $. /stk b = 10 b = 12 C++ arrays, arrays and loops In this tutorial, we are going to talk about arrays. An array lets you declare and work with a collection of values of the same type. Let’s say you want to declare four integers. With the knowledge from the last few tutorials you would do something like this: int a , b , c , d; What if you wanted to declare a thousand variables? That will take you a long time to type. This is where arrays come in handy. An easier way is to declare an array of four integers, like this: int a[4];

The four separate integers inside this array are accessed by an index. Each element can be accessed, by using square brackets, with the element number inside. All arrays start at element zero and will go to n-1. (In this case from 0 to 3.) Note: The index number, which represents the number of elements the array is going to hold, must be a constant value. Because arrays are build out of non-dynamic memory blocks. In a later tutorial we will explain arrays with a variable length, which uses dynamic memory. So if we want to fill each element you get something like this: int a[4]; a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4; If you want to use an element, for example for printing, you can do this: out ;; a[1]; Arrays and loops One of the nice things about arrays is that you can use a loop to manipulate each element. When an array is declared, the values of each element are not set to zero automatically. In some cases you want to “ re-initialize” the array (which means, setting every element to zero). This can be done like in the example above, but it is easier to use a loop. Here is an example: #include; iostream; using namespace std; int main() { int a[4]; int i; for (i = 0; i ; 4; i++) a[i] = 0; for (i = 0; i ; 4; i++) cout ;; a[i] ;; ' '; return 0; } Note: In the first “ for loop” all elements are set to zero.

The second “ for loop” will print each element. Multi-dimensional arrays The arrays we have been using so far are called one-dimensional arrays. Here is an example of a one-dimensional array: int a[2]; 0| 1| 1| 2| Note: A one-dimensional array has one column of elements. Two-dimensional arrays have rows and columns. See the example below: int a[2][2]; | 0| 1| 0| 1| 2| 1| 4| 5| Note: a[0][0] contains the value 1. a[0][1] contains the value 2. a[1][0] contains the value 4. a[1][1] contains the value 5. So let’s look at an example that initialize a two-dimensional array and prints each element: #include; iostream; using namespace std; int main() { nt a[4][4]; int i , j; for (i = 0; i ; 4; i++) { for (j = 0; j ; 4; j++) { a[i][j] = 0; cout ;; a[i][j] ;; ' '; } } return 0; } Note: As you can see, we use two “ for loops” in the example above. One to access the rows the other to access the columns. You must be careful when choosing the index number, because there is no range checking done. So if you index (choose an element) past the end of the array, there is no warning or error. Instead the program will give you “ garbage” data or it will crash. Arrays as parameters In C++ it is not possible to pass a complete block of memory by value as a parameter to (for example) a function.

It is allowed to pass the arrays address to (for example) a function. Take a look at the following example: #include; iostream; using namespace std;

void printfunc(int my_arg[], int i) { for (int n= 0; n ; i; n++) cout ;; my_arg[n] ;; '
'; } int main() { int my_array[] = {1, 2, 3, 4, 5}; printfunc(my_array, 5); return 0; } The function printfunc accepts any array (whatever the number of elements) whose elements are of the type int. The second function parameter (int i) tells function the number of elements of the array, that was passed in the first parameter of the function. With this variable we can check (in the “ for” loop) for the outer bound of the array. That’s all for this tutorial. include ; iostream; using namespace std; int main() { int myarr[2][3]; for(int r = 0; r ; 2; r++){ for(int c = 0; c ; 3; c++){ myarr[r][c] = r*c+1; } } for(r = 0; r ; 2; r++){ for(int c = 0; c ; 3; c++){ cout ;; myarr[r][c] ;; " "; } cout ;; endl; } return 0; } #include ; iostream; using namespace std; int minArray(int arr[][5], int rowCap, int colCap) { int m = arr[0][0]; for (int r = 0; r ; rowCap; r++) for (int c = 0; c ; colCap; c++) if (arr[r][c] ; m) m = arr[r][c]; return m; } int main() { int x[3][5] = { {13, 4, 35, 22, 3}, {32, 3, 7, 3, 2}, {3, 4, 4, 4, 2}}; cout ;; minArray(x, 3, 5) ;; endl; return 0; }