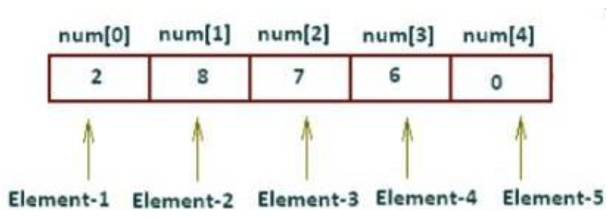
Chapter 7 - Arrays

Array Data Type

- How many variables do we need for the following problem?
    - Read in 1000 integers, output the number of values that are above the mean value.
- imagine keeping track of 5 test scores, or 100, or 1000 in memory
    - How would you name all the variables?
    - How would you process each of the variables?
- Arrays provide a way to
    - Declare multiple "variables" at once and
    - Refer to these variables using one common name
- So far we have used scalar/primitive data types
    - Each variable holds only one value .
- Composite data types:
    - A single variable can contain multiple values.
    - An array is a composite data type.

- An array contains multiple values of the **same** type.
- Values are stored consecutively in memory.
- An array definition in C++ :

```
int     num[5];
```



- **Name** of the array is : num
- 5 is the **size** decelerator: the number of elements (values) in the array.
- **int** is the type of each of the 5 elements

More Examples of Arrays :

```
float temperatures[100];
char  name[51];
long  units[50];
```

- The size must be an integer and a constant:
    - A literal or named constant

```
const int SIZE = 40;
double grades[SIZE];
```

Memory allocation

- When an array is defined, all of the memory it needs is allocated.

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

- An int requires 4 bytes
- numbers array requires 10 integers:
- 10 integers * 4 bytes = 40 bytes
- The memory is allocated sequentially

Array Elements

- Individual elements of the array have unique **subscripts** (also called an **indexes**)

- The subscripts are **0-based**

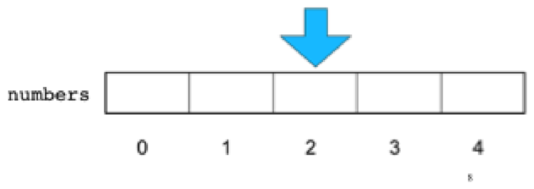
    - The first element has subscript 0
    - The second element has subscript 1
    - . . .
    - The last element has subscript (size -1)

Accessing Array Elements

- Syntax to access one element:

```
numbers[2] //the third element of numbers array
```

- Called “numbers at 2” or “numbers sub 2”



Array subscripts

- Square brackets in **definition** indicate size
- Square brackets in an **expression** indicate subscript.
- The subscript is always an **integer**, regardless of the type of the array elements.
- the subscript can be ANY integer expression
  - literal: 2
  - variable: i
  - expression: (i+2)/2

- Given the following array definition:

```
double numArray[10];
```

- The expression `numArray[i]` may be used exactly like any variable of type double.

Using array elements

- Examples of using Array Element

```
double values [10] ;

values[0] = 22.3;
values[1] = 11.1;

cout << "Enter a number: ";
cin >> values[2];

double sum = values[0] + values[1] + values[2];
double avg = sum/3.0;

cout << "Values at zero: " << values[0] << endl;


int i=2;
if (values[i] > 32.0)
     cout << "Above freezing" << endl;
```

Array initialization

- You can initialize arrays when they are defined :

```
const int NUM_SCORES = 3;
float scores[NUM_SCORES] = {86.5, 92.1, 77.5};
```

- Values are assigned in order :

```
scores[0] =  86.5
scores[1] =  92.1
scores[2] =  77.5
```

- Note : What happens if you uninitialize an array – What values are stored in them ?

Partial array initialization

- You can initialize only the first part of the array

```
const int NUM_SCORES = 10;
float scores[NUM_SCORES] = {86.5, 92.1, 77.5};
```

- The first 3 elements get the values specified.
- The remaining 7 elements gets initialized to 0.0
- The list of elements cannot have more elements than the size of the array.

Implicit array sizing via initialization

- When you initialize, you don't need to specify the size

```
float scores[] = {86.5, 92.1, 77.5};
```

- The size of the array is the number of the elements listed

Arrays of char

- `char word[] ="football"; //automatically adds '\0'`

- The size of the array is the length of the string plus one (for the null character) so 9 here.
- Can also use a list of chars to initialize:

```
char word[]= {'f','o','o','t','b','a','l','l','\0'};
```

- **Must include** the ***null*** character in this case.

- If you **forget** the null character, operations on the char array may not behave correctly.

- Arrays of char in C++ are called " C-String "

Note : Arrays of char are sometimes handled differently from other arrays.

- For example , you **can** output array of char

```
char word[]={'f','o','o','t','b','a','l','l','\0'};
cout << word << endl; // outputs: football
```

- But you **can not** output array of int :

```
int numbers[] = {1, 2, 3};
cout << numbers << endl; // won't work like you want
```

Operations over arrays

- Generally , there is NO operation you can perform over entire arrays
- Some operations may appear to work (no errors) but you ***don't*** get the desired results.

```
int numbers1[] = {1, 2, 3};
int numbers2[] = {4, 5, 6};
int numbers3[3] ;

cin >> numbers1; //input, won't work
cout << numbers1 << endl; //output, won't work

numbers1 = numbers2; //assignment, won't work
if (numbers1==numbers2) //comparison, won't work
...
numbers3 = numbers1 + numbers2; //addition, won't work
```

- Except for I/O for char arrays, array operations must be done one element at a time.

Problem Statement : Input 8 programming assignment grades for 1 student in CS1428 and store them in scores array.

```
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
cout << "Enter the " << NUM_SCORES << " scores:  " << endl;
cin >> scores[0] >> scores[1];
cin >> scores[2] >> scores[3];
cin >> scores[4] >> scores[5];
cin >> scores[6] >> scores[7];
```

- Is there a better way?

Array input using a loop

- We can use a for loop to input into the array
  - the subscript can be a variable

```
const int NUM_SCORES = 8;
 int scores[NUM_SCORES];
 cout << "Enter the " << NUM_SCORES  << " scores:  " << endl;

 for (int i=0; i < NUM_SCORES; i++) {
      cin >> scores[i];
 }
```

Array output using a loop

- We can use a for loop to output the elements of the array

```
const int NUM_SCORES = 8;
     int scores[NUM_SCORES];
cout << "Enter the " << NUM_SCORES << " scores:  " << endl;

for (int i=0; i < NUM_SCORES; i++) {
     cin >> scores[i];
     }

cout << "You entered these values: \n";
for (int i=0; i < NUM_SCORES; i++) {
     cout << scores[i] << " \n";
}
```

Summing & Averaging values in an array

- We can use a for loop to sum the elements of
- the array (running total)

```
int total = 0; //initialize accumulator
for (int i=0; i < NUM_SCORES; i++) {   // Summing
      total = total + scores[i];
}

//  Calculating  Average

double average =  static_cast<double>(total) / NUM_SCORES;
```

Finding the maximum and the minimum values in an array

- We can use a for loop to find the max and min values :

- **Note** : keep track of the max and the min values so far

```
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
cout<< "Enter the " << NUM_SCORES   << " scores:  " << endl;
for (int i=0; i < NUM_SCORES; i++) {
      cin >> scores[i];
}

int maximum = scores[0]; //init max to first elem

for (int i=1; i < NUM_SCORES; i++) { //start i at 1
      if (scores[i] > maximum)
            maximum = scores[i]
}

int minimum = scores[0]; //init min to first elem

for (int i=1; i < NUM_SCORES; i++) { //start i at 1
      if (scores[i] < minimum)
            minimum = scores[i];

}
```

Finding the maximum value in an array, and its position.

- Keep track of the maximum value, AND what its position is :

```
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
// input code goes here
int indexOfMax = 0; //initialize  indexOfMax to first
int maximum = scores[0]; // initialize  max to first elem
for (int i=1; i < NUM_SCORES; i++) { //start i at 1
      if (scores[i] > maximum) {
            maximum = scores[i];
            indexOfMax = i;    }
}
cout << "The highest score was " << maximum
<< " and it was assignment " << indexOfMax+1
<< endl;
```

Counting values in an array that pass a test

- Use a for loop and a counter, incr counter for elements that pass the test (i.e. elem > 75)

```
const int NUM_SCORES = 8;
int scores[NUM_SCORES];
// input code goes here
int count = 0; // initialize  count to zero
for (int i=0; i < NUM_SCORES; i++) {
      if (scores[i] > 75) {
      count++;
      }
}
cout << "There were " << count << " scores above 75." << endl;
```

Copying elements of an array into another array

```
const int SIZE = 4;
int values1[SIZE] = {100, 200, 300, 400};
int values2[SIZE];
values2 = values1; //WRONG, won't work right
for (int i = 0; i < SIZE; i++) {
      values2[i] = values1[i];
}
```

Array compare (for equality)

- Can not use == when comparing two arrays

```
const int SIZE = 4;
int values1[SIZE] = {100, 200, 300, 400};
int values2[SIZE] = {100, 200, 300, 400};

if (values2 == values1) //WRONG, won’t work right
      cout << "equal!" << endl;

bool arraysEqual = true; //flag, assume true
int i = 0;
while (arraysEqual && i < SIZE) {
      if (values1[i] != values2[i])
            arraysEqual = false;
      i++;
}

if (arraysEqual) cout << "equal!"  ;   else  cout << "not equal!" ;
```

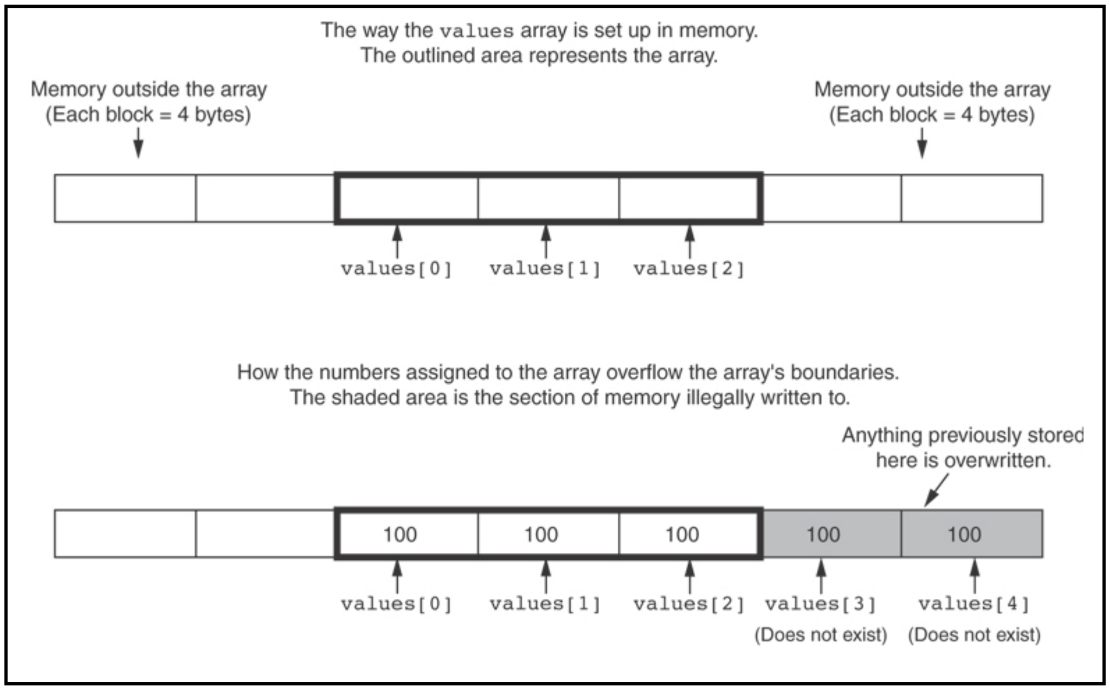
What is the output?

```
int numArray[5] = {6,7,8,9,0};
int count = 2;
numArray[count]++;
numArray[count++];
cout << count << endl;
for (int i=0; i<5; i++) {
      cout << numArray[i] << " ";
}
cout << endl;
```

C++: No bounds checking

- When you use a value as an array subscript, C++ does not check it to make sure it is a valid subscript.
- In other words, you can use subscripts that are beyond the bounds of the array.

```
const int SIZE = 3;
int values[SIZE];
for (int i=0; i < 5; i++) {
      values[i] = 100;
}
```


The way the values array is set up in memory.
The outlined area represents the array.
Memory outside the array (Each block = 4 bytes)
Memory outside the array (Each block = 4 bytes)
values[0]
values[1]
values[2]
How the numbers assigned to the array overflow the array's boundaries.
The shaded area is the section of memory illegally written to.
Anything previously stored here is overwritten.
100
100
100
100
100
values[0]
values[1]
values[2]
values[3] (Does not exist)
values[4] (Does not exist)

Be careful not to use invalid subscripts - Doing so can, and without warning:

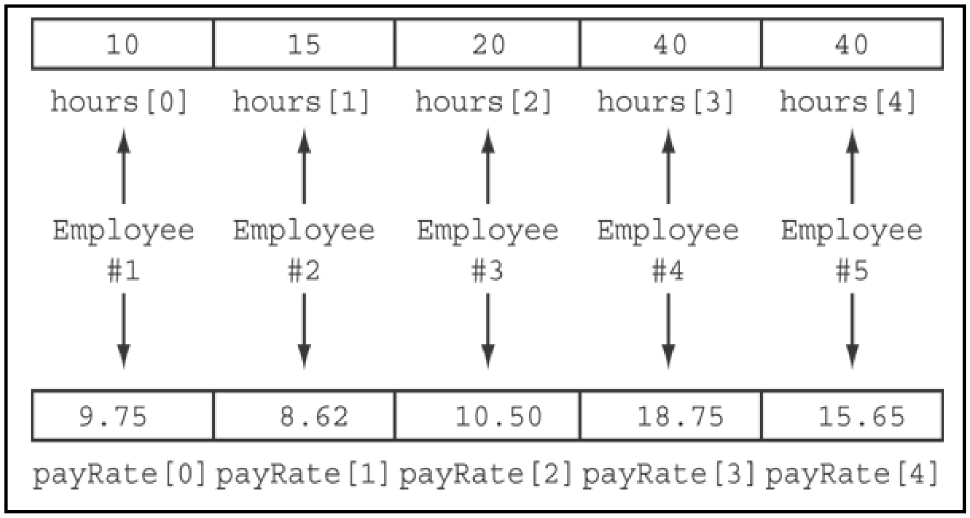
- corrupt other memory locations
- crash program
- lock up computer
- cause elusive bugs

It's easy to get the loop index off by one, especially if you start at 1 instead of 0 - use <= instead of <

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
            numbers[count] = 0;
```

Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- A subscript is used to relate arrays: elements at same subscript are related, belong to the same entity
- Arrays may be of different types



Example : Employee hours worked and pay rate

```
const int NUM_EMPS = 5; // Number of Employees
int hours[NUM_EMPS]; // Holds hours worked
double payRate[NUM_EMPS]; // Holds pay rates

cout << "Enter the hours worked and pay rates:\n";

for(int i = 0; i < NUM_EMPS; i++) {
      cout << "Hours worked by employee " << i+1 << ": ";
      cin >> hours[i];
      cout << "Hourly pay rate for employee " << i+1 << ": ";
      cin >> payRate[i];
}

cout << "Here is the gross pay for each employee:\n";
cout << fixed << setprecision(2);
for(int i = 0; i < NUM_EMPS; i++) {
       double grossPay = hours[i] * payRate[i];
       cout << "Employee " << i+1 << ": $";
       cout << grossPay << endl;
}
```

Output :

```
Enter the hours worked and pay rates:
Hours worked by employee 1: 10
Hourly pay rate for employee 1: 9.75
Hours worked by employee 2: 15
Hourly pay rate for employee 2: 8.62
Hours worked by employee 3: 20
Hourly pay rate for employee 3: 10.50
Hours worked by employee 4: 40
Hourly pay rate for employee 4: 18.75
Hours worked by employee 5: 40
Hourly pay rate for employee 5: 15.65
Here is the gross pay for each employee:
Employee 1: $97.50
Employee 2: $129.30
Employee 3: $210.00
Employee 4: $750.00
Employee 5: $626.00
```

Passing Single-Dimension Array to Function

- In C++ arrays can only be **reference parameters**.
- It is not possible to pass an array by value. Therefore, the ampersand (&) is omitted.
- What is actually passed to the function, when an array is the formal parameter, is the base address of the array (the memory address of the first element in the array). This is true whether the array has one or more dimensions.
- When **declaring a 1-D array parameter**, the compiler only needs to know that the parameter is an array; it doesn't need to know its size. The complier will ignore it, if it is included. However, inside the function we still need to make sure that only legitimate array elements are referenced. Thus a **separate parameter** specifying the length of the array must also be passed to the function.

  int ProcessValues (int [], int);   // works with ANY 1-d array

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 modify , print and return the average of the numbers passed through the array
.

```
#include <iostream>
using namespace std;

// function prototype

void modifyArray(int arr[], int size);
double getAverage(int arr[], int size);
void printArray(int arr[], int size);

int main ()
{
    // an int array with 5 elements.
    int balance[5] = {1 , 2, 3, 4, 5 };
    double avg;

    // Print the Original Values of the Array
    printArray( balance , 5);

    // Modify the Array
    modifyArray( balance , 5 );

    // pass pointer to the array as an argument.
    avg = getAverage( balance, 5 ) ;

    // output the returned value
    cout << "Average value is: " << avg << endl;

   return 0;     }
```

```
// function definition

void modifyArray(int arr[], int size)
{

   for (int i = 0; i < size;    i ++ )
      {          arr[i]   +=  10 ;       }

// print the Array After Modification
// Call function printArray

   printArray ( arr , 5 );
}


double getAverage(int arr[], int size)
{
  int    i, sum = 0;
  double avg;


  for (i = 0; i < size;  i ++ )
    {       sum += arr[i];    }

  avg = double(sum) / size;


  return avg;
}

void printArray(int arr[], int size)
{

   cout<<"Array Values \n";
   for (int i = 0; i < size;    i++ )
     {      cout << arr[i] << "   " ;      }

   cout<<endl;
}
```

Sample Run

```
Array Values
1  2  3  4  5
Array Values
11  12  13  14  15
Average value is: 13
```

Since arrays are **always passed by reference**, **all changes made to the array elements inside the function will be made to the original array**. The only way to protect the elements of the array from being inadvertently changed, is to declare an array to be a ***const*** parameter.

Example:

```
#include <iostream>
using namespace std;

int SumValues (const int [], int);    //function prototype

int main( )
{
      const int length =10;
      int Array[10]={0,1,2,3,4,5,6,7,8,9};
      int total_sum;
      total_sum = SumValues (Array, length);      //function call
            cout <<"Total sum is " <<total_sum;
      return 0;

}

// function definition

int SumValues (const int values[], int  num_of_values)
{
          int sum = 0;
          for( int i=0; i < num_of_values; i++)
                  sum+=values[i];
          return sum;
      }
```

- Since you do not intend to change the values of an array in the above function, make it a const parameter to protect yourself. The original array (in the main) should not be const, or you wouldn't be able to make any changes to it at all.

- You do not however need to make the length of an array a const parameter, since it is passed by value and any changes to it will not affect the actual parameter. The compiler will not allow you to pass it by reference if it was declared as a const int in the main ( ).

Sample Run

```
Total sum is 45
```

Arrays and Overloaded Functions :

```
#include<iostream>
#include<iomanip>
using namespace std;

int     findmax(int [] );
float   findmax(float [] , int);
double  findmax(double []);

const int SIZE = 10;


int main()
{
  int  x = 10;
  int    intArray[]={ 1,8,4,2,3,0,9,3,5,7 };
  float floatArray[SIZE]={145.15,312.3,3163.2,119.13,710.1,315.4,511.2};
  double doubleArray[10]={15.12354323,2.41237763,63.29123876,
          19.67123863,78.34123541,35.44123009,51.21230392,53.40123967,
          39.80612304,59.11111};

  cout<<"largest value in the intArray is   "<<(findmax(intArray))<<"\n";
  cout<<"largest value in the floatArray is   "<<(findmax(floatArray , x ))<<"\n";
  cout<<"largest value in the doubleArray is "<<(findmax(doubleArray))<<"\n";

  return 0;
}

int findmax(int intArray[] )
{
   int max=0;
   for(int i=0;i< SIZE ;i++)
   {
     if(intArray[i]>max)
        {
          max=intArray[i];
        }
   }
   return max;
}
```

```
float findmax(float floatArray[] , int x)
{
   float max=0;
   for(int i=0; i< x ; i++)
   {
     if(floatArray[i]>max)
        {
          max=floatArray[i];
        }
   }
   return max;
}


double findmax(double doubleArray[])
{
   double max=0;
   for(int i=0;i<10;i++)
   {
     if(doubleArray[i]>max)
        {
          max=doubleArray[i];
        }
   }
   return max;
}
```

Sample Run

```
largest value in the intArray is 9
largest value in the floatArray is 3163.2
largest value in the doubleArray is 78.3412
```

What is the output of the following program :

```
#include <iostream>

using namespace std;



void display(char []);
void display(char [], char []);

int main()
{
   char first[] = "C programming";
   char second[] = "C++ programming";

   display(first);
   display(first, second);

   return 0;
}

void display(char s[])
{
   cout << s << endl;
}

void display(char s[], char t[])
{
   cout << s << "\t\t" << t << endl;
}
```

Sorting Arrays

```
#include <iostream>
using namespace std;

int main() {

            int Arr[100], n, i;

            cout << "\nEnter number of elements you want to insert    ";
            cin >> n;
   cout <<"\n\n";
            for (i = 0; i < n; i++) {
                        cout << "Enter element  " << i + 1 << ":  ";
                        cin >> Arr[i];
            }

            cout << "\n\nThe original array  " << endl;
            for (i = 0; i < n; i++)
                        cout << Arr[i] << "   ";

            int tmp;
            for (i = 0; i < n; i++)
      for ( int j = 0 ; j < n ; j++)
            if ( Arr[i] < Arr[j]) {
                        tmp = Arr[i];
                        Arr[i] = Arr[j];
                        Arr[j] = tmp;
            }

            cout << "\n\nThe Sorted    array  " << endl;
                        for (i = 0; i < n; i++)
                                    cout << Arr[i] << "   ";


            return 0;
}
```

```
Enter number of elements you want to insert
5


Enter element  1:  12
Enter element  2:  32
Enter element  3:  1
Enter element  4:  45
Enter element  5:  6


The original array
12   32   1   45   6

The Sorted  array
1   6   12   32   45
```

Reading Data from a File into an Array

In many circumstances you will need to read data from a file and store it in an array. The process is straightforward, and in most cases is best done with a loop. Each iteration of the loop reads an item from the file and stores it in an array element.

Example :

This program uses a partially-filled array to store monthly sales figures for a set of offices. It then finds and displays the total sales amount, the average sales amount, and a listing of the offices with sales below the average. The data to fill the array is read in from a file and the number of data values are counted.

```
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

int main() {

        const int SIZE = 20;
        ifstream fin;

        int numOffices, count;
        double sales[SIZE]; // Array to hold the sales data
        double totalSales = 0.0; // Accumulator initialized to zero
        double averageSales; // Average sales for all offices

        // Open the data file

        fin.open("inFile.txt");
        if (!fin) {
                cout << "Program is terminated - cannot locate input file"
                     << " or wrong file name ";
                return 1;
        }

        count = 0;

        // Read values from the file and store them in the array,
        // counting them and summing them as they are read in

        while (count < SIZE && fin >> sales[count])

        {
                totalSales += sales[count];
                count++;
        }
        numOffices = count;
        fin.close();
        // Calculate average sales

        averageSales = totalSales / numOffices;

        // Display total and average

        cout << fixed <<  setprecision(2);
        cout << "The total sales are   $" << setw(9) << totalSales << endl;
        cout << "The average sales are $" << setw(9) << averageSales << endl;

        // Display figures for offices performing below the average

        cout << "\nThe following offices have below-average " << "sales figures.\n";

        for (int office = 0; office < numOffices; office++) {
                if (sales[office] < averageSales) {
                        cout << "Office " << setw(2) << (office + 1) << " $"
                                << sales[office] << endl;

                }
        }
        return 0;
}
```

Sample Input **Sample Output**

inFile.txt

```
1000
2000
3000
4000
5700
6000
```

```
The total sales are   $ 21700.00
The average sales are $  3616.67

The following offices have below-average sales figures.
Office  1 $1000.00
Office  2 $2000.00
Office  3 $3000.00
```

Multi-dimensional Arrays

C++ allows multidimensional arrays. The general form of a multidimensional array declaration −

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array −

```
int threedim[5][10][4];
```

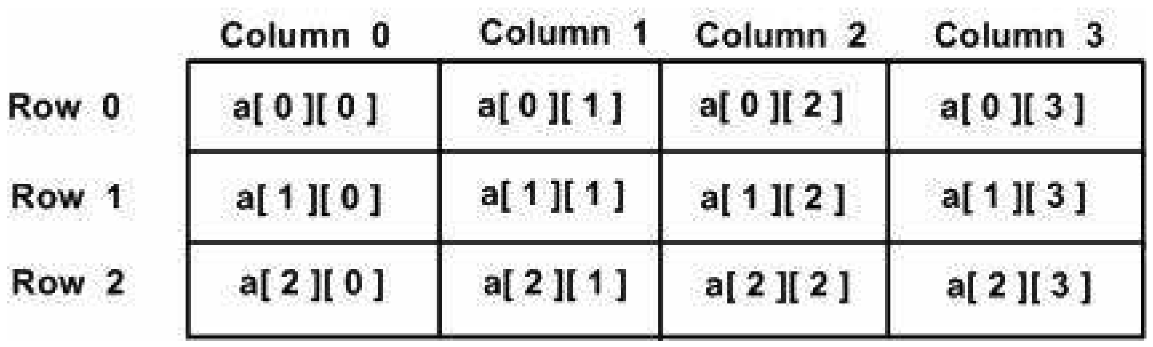
Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be thinked as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {

{0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
{4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
{8, 9, 10, 11}   /*  initializers for row indexed by 2 */

 };
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array.

Example

```
#include <iostream>
using namespace std;

int main () {
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

   // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ ) {

         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }

   return 0;
}
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Exercises :

What does the following code do and what is the output ?

```
#include <iostream>

using namespace std;

int billy[] = { 16, 2, 77, 40, 12071 };
int n, result = 0;

int main() {

	for (n = 0; n < 5; n++) {
		result += billy[n];
	}
	cout << result;

	return 0;
}
```

What does the following code do and what is the output ?

```
float a[100000];
int n = 0;
while (cin >> a[n] && n < 4) {
        n++;
}

for (int i = n - 1; i >= 0; i--) {
        cout << a[i] << endl;
}
```

Look at the following array definition.

```
        int numbers[5] = { 1, 2, 3 };
```

What value is stored in numbers[2] , numbers[4] , numbers[10] ?

What is the output of the following code?

```
    const int SIZE = 3 ;

    int myArray[SIZE] = { 10, 20, 30 };

    for (int i = 0; i < SIZE; ++i)
          ++myArray[i];

    for (int i = 0; i < SIZE; i++)
          cout << myArray[i] << '\n';
    cout << "\n\n";

    for (int i = 0; i < SIZE; ++i)
          myArray[i]++;

    for (int i = 0; i < SIZE; i++)
          cout << myArray[i] << '\n';
    cout << "\n\n";

    for (int i = 0; i < SIZE; ++i)
          myArray[i++];

    for (int i = 0; i < SIZE; i++)
          cout << myArray[i] << '\n';
```

What does the following code do and what is the final value of n ?

```
const int SIZE = 9 ;

int array[SIZE] = { 13, 11, 15, 9, 7, 5, 8, 3, 1 };

int n = 1;
for (int i = 1; i < SIZE; i++)
     if (array[i] < array[i - 1])
          n++;
     else
          n = 1;

cout << n;
```

What is the output of the following code?

```
int B[] = { 10,  25 , 38 , 4 };

for( int k = 0; k < 4; k++) {
    B[k] = B[k] + k;
  }
for (int i = 0; i < 4; i++)
                cout << B[i] << '\n';
```

What is the output of the following program ?

```
#include <iostream>
using namespace std;

void pM(int arr[5]);

int main() {
	int arr1[5] = { 25, 10, 54, 15, 40 };
	int arr2[5] = { 12, 23, 44, 67, 54 };
	pM(arr1);
	pM(arr2);
}

void pM(int arr[5]) {
	int x = arr[0] ,  y = arr[0];
	for (int i = 0; i < 5; i++) {
		if (x > arr[i])
			x = arr[i];
		else
			y = arr[i];

	}
	cout << "\n\n\nThe First  Element is:  " << x << "\n";
	cout << "\nThe Second Element is:  " << y << "\n";

}
```

What Does the following code do ?

```
#include<iostream>
using namespace std;

int main() {
	int a[50], n, i, j, temp;
	cout << "Enter the size of array: ";
	cin >> n;
	cout << "Enter the array elements: ";

	for (i = 0; i < n; ++i)
		cin >> a[i];

	for (i = 1; i < n; ++i) {
		for (j = 0; j < (n - i); ++j)
			if (a[j] > a[j + 1]) {
				temp = a[j];
				a[j] = a[j + 1];
				a[j + 1] = temp;
			}
	}

	cout << "Array after :";
	for (i = 0; i < n; ++i)
		cout << " " << a[i];

	return 0;
}
```

Assume the following array declaration :

```
char mArray[10] = { 'A','b',' ','c','1','2',' ','w','q','\0' };
```

Write a loop that displays the content of the array without any spaces

Write a loop that counts then displays the number of spaces in the array

Write a loop that counts then displays the number of digits in the array

Write a loop that counts then displays the number of small letters in the array

Following are some sample questions dealing with arrays:

- Assume A is an array of N integers. Find the sum of the first and last entries and assign it to the third element.
- Write C++ code segment that finds the sum of two arrays.
- Assume A , B are is arrays of N integers. Write a C++ code that displays the intersection of array A and Array B.