

# Lecture 1

## 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 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:

Consider the case where you want to record the test scores for 30 students in a class. To do so, you would have to allocate 30 variables!

```
int Student1;  
int Student2;  
int Student3;  
// ...  
int Student30;
```

Arrays give us a much easier way to do this:

```
int Scores[30]; // allocate 30 integers
```

In the above example, we declare an array named Scores. When used in an array definition, the subscript operator ([]) is used to tell the compiler how many variables to allocate. In this case, we're allocating 30 integers. Each of these variables in an array is called an element.

To access each of our 30 integer array elements, we use the subscript operator with an integer parameter called an index to tell the compiler which variable we want. The first element of our array is named Scores[0]. The second is Scores[1]. The tenth is Scores[9].

Note that in C++, arrays always count starting from zero! This means an array of size N has array elements 0 through N-1. This can be awkward for new programmers who are used to counting starting at 1.

Note that the subscript operator actually has two uses here: in the variable declaration, the subscript tells how many elements to allocate. When using the array, the subscript tells which array element to access.

```
int anArray[5]; // allocate 5 integers
anArray[0] = 7; // put the value 7 in element 0
```

## Static Allocation Of Arrays

All arrays discussed or used have been "statically allocated"

The array size was specified using a constant or literal in the code

When the array comes into scope, the entire size of the array can be allocated, because it was specified

You won't always know the array sizes when writing source code

Consider a program that modifies an image. As the developer, you won't know what image size the user will use

One solution: Declare the image array to be 5000 rows by 5000 columns

- Problem #1: This likely wastes a lot of memory – if the user uses an image that is 250x250, then there are 24,937,500 unused pixels. If each pixel requires 4 bytes, this is almost 100 MB (megabytes!) of wasted space

- Problem #2: What if the user needs to edit an image that is 6000x6000? Your program will fail, and likely result in a crash

If an array is "dynamically allocated", then space is not reserved for the array until the size is determined.

**Example:**

```
int anArray[3]; // allocate 3 integers  
anArray[0] = 2;  
anArray[1] = 3;  
anArray[2] = 4;
```

```
int nSum =0;  
for(int i=0;i<3;i++)  
nSum+= anArray[i];  
cout << "The sum is " << nSum << endl;
```

This program produces the result:

The sum is 9

*Array elements may be accessed by a non-constant integer variable:*

*Ex:*

```
int anArray[5];  
int nIndex = 3;  
anArray[nIndex] = 7;
```

However, when doing array declarations, the size of the array must be a constant.

```
int anArray[5]; // Ok -- 5 is a literal constant
```

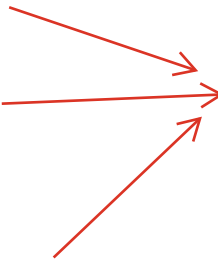
```
int anArray[ARRAY_SIZE]; // Ok -- IF ARRAY_SIZE is a symbolic constant
```

```
const int nArraySize = 5;
```

```
int anArray[nArraySize]; // Ok -- nArraySize is a variable constant
```

```
int nSize = 5;
```

```
int anArray[nSize]; // Not ok! -- nSize is not a constant!
```



To summarize, array elements can be indexed with constants or non-constants, but arrays must be declared using constants. This means that the array's size must be known at compile time!



Arrays can hold any data type, including floating point values and even structs:

```
Float x[10];
```

```
Double y[20];
```

Elements of an array are treated just like normal variables, and as such have all of the same properties.

adjacent

The array is assigned to a contiguous section of memory

In memory all elements of array will be stored as static allocation  
X[1] will have the address after x[0], and so on.

# Initializing Arrays

C++ provides a convenient way to initialize entire arrays via use of an initializer list.

```
int anArray[5] = { 3, 2, 7, 5, 8 };
```

```
cout << anArray[0] << endl;
```

```
cout << anArray[1] << endl;
```

```
cout << anArray[2] << endl;
```

```
cout << anArray[3] << endl;
```

```
cout << anArray[4] << endl;
```

Which prints:

3

2

7

5

8

decleration => `int arr[4]`

instialization => `int arr[4] = {1,2,3,4}`

in case of not insializing any element so garbage will be sto

What happens if you don't initialize all of the elements in an array? The remaining elements are initialized to 0:

```
int anArray[5] = { 3, 2, 7 };  
cout << anArray[0] << endl;  
cout << anArray[1] << endl;  
cout << anArray[2] << endl;  
cout << anArray[3] << endl;  
cout << anArray[4] << endl;
```

Which prints:

3  
2  
7  
0  
0

Consequently, to initialize all the elements of an array to 0, you can do this:

```
// Initialize all elements to 0  
int anArray[5] = { 0 };
```

u instialize first with zero and other not inzialized so will be zero

Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

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

```
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 << "    "<< j << "    "<< n[ j ] << endl;
    }
    return 0;}
```

## **Output**

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

## **Sizeof**

The sizeof operator can be used with arrays. It returns the total size allocated for the entire array:

```
int anArray[] = { 0, 1, 2, 3, 4 }; // declare array of 5 elements
```

```
cout << sizeof(anArray); // prints 20 (5 elements * 4 bytes each)
```

# Multidimensional Array

An array of arrays is called a multidimensional array.

```
int anArray[3][5]; // a 3-element array of 5-element arrays
```

In this case, since we have 2 subscripts, this is a two-dimensional array. In a two-dimensional array, it is convenient to think of the first subscript as being the row, and the 2nd subscript as being the column. Conceptually, the above two-dimensional array is laid out as follows:

[0][0] [0][1] [0][2] [0][3] [0][4]

[1][0] [1][1] [1][2] [1][3] [1][4]

[2][0] [2][1] [2][2] [2][3] [2][4]

In memory, The first row occupies the first portion, The second row occupies the second portion

To access the elements of a two-dimensional array, simply use two subscripts:

```
anArray[2][3] = 7;
```

To initialize a two-dimensional array, it is easiest to use nested braces, with each set of numbers representing a row:

```
int anArray[3][5] = { { 1, 2, 3, 4, 5, }, // row 0
{ 6, 7, 8, 9, 10, }, // row 1
{ 11, 12, 13, 14, 15 } // row 2
};
```

When the C++ compiler processes this list, it actually ignores the inner braces altogether. However, we highly recommend you use them anyway for readability purposes.

Two-dimensional arrays with initializer lists can omit (only) the first size specification:

```
int anArray[ ][5] = {{ 1, 2, 3, 4, 5, },{ 6, 7, 8, 9, 10, },{ 11, 12, 13, 14, 15 } };
```



However, the following is not allowed:

```
int anArray[ ][ ] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } }; //error
```

```
int anArray[2][ ] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } }; // error
```

Because the inner parenthesis are ignored, the compiler can not tell whether you intend to declare a 1×8, 2×4, 4×2, or 8×1 array in this case.

Just like normal arrays, multidimensional arrays can still be initialized to 0 as follows:

```
int anArray[3][5] = { 0 };
```

Accessing all of the elements of a two-dimensional array requires two loops: one for the row, and one for the column. Since two-dimensional arrays are typically accessed row by row, generally the row index is used as the outer loop.

```
for (int nRow = 0; nRow < nNumRows; nRow++)  
for (int nCol = 0; nCol < nNumCols; nCol++)  
    cout << anArray[nRow][nCol];
```

Multidimensional arrays may be larger than two dimensions.  
Here is a declaration of a three-dimensional array:

```
int anArray[5][4][3];
```

Three-dimensional arrays are hard to initialize in any kind of intuitive way using initializer lists, so it's typically better to initialize the array to 0 and explicitly assign values using nested loops.

## Example

This program calculates and prints a multiplication table for all values between 1 and 9 (inclusive). Note that when printing the table, the for loops start from 1 instead of 0. This is to omit printing the 0 column and 0 row, which would just be a bunch of 0s!

```
    // Declare a 10x10 array
const int nNumRows = 10;
const int nNumCols = 10;
int nProduct[nNumRows][nNumCols] = { 0 };
// Calculate a multiplication table
for (int nRow = 0; nRow < nNumRows; nRow++)
    for (int nCol = 0; nCol < nNumCols; nCol++)
        nProduct[nRow][nCol] = nRow * nCol;
// Print the table
for (int nRow = 1; nRow < nNumRows; nRow++)
{
    for (int nCol = 1; nCol < nNumCols; nCol++)
        cout << nProduct[nRow][nCol] << "\t";
    cout << endl; }
}
```

Here is the output:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

# Arrays and functions

// function uses array as argument

// the array is passed by its address, so if it is changed in function, it will be changed in main

```
void fun(int k[]);  
int main()  
{int m[10];    int i;  
cout<<" array in main "<<endl;  
  for (i=0;i<10;i++)  
  { m[i]=2*i; cout<<m[i]<<" ";}  cout<<endl;  
                                fun(m );  
cout<<" array in main after calling function "<<endl;  
  for(i=0;i<10; i++)  
    cout<<m[i]<<" ";  return 0; }  
  
void fun(int k[] )  
{ cout<<" array in function "<<endl;  
  for(int i=0;i<10;i++)  
{ cout<<k[i]<<" ";  
  k[i]= 3*i; }  cout<<endl; }
```

### Solution

array in main

0 2 4 6 8 10 12 14 16 18

array in function

0 2 4 6 8 10 12 14 16 18

array in main after calling function

0 3 6 9 12 15 18 21 24 27

*// passing elements of array by value or by reference*

```
void fun_1( int x, int &y)  
{cout<< " x " <<x<<" y " <<y<<endl;  
x=20; y=-7;}
```

```
void fun_2(int k[ ] )  
{  
    cout<<endl<<" array in function 2 " <<endl;  
    for(int i=0;i<5;i++) cout<<k[i]<<" ";  
    cout<<endl<<" array in function 2 " <<endl;  
    for(int i=0;i<5;i++){k[i]=2*i;cout<<k[i]<<" "};  
}
```

```
main() { int m[5]= {5, 6, 7,8,9};  
cout<<" array " <<endl;  
    for(int i=0;i<5;i++)cout<<m[i]<<" ";  
cout<<endl; fun_1(m[0],m[4]);  
cout<<" array after calling function fun_1 " <<endl;  
for(int i=0;i<5;i++)cout<<m[i]<<" "; cout<<endl;  
fun_2(m);cout<<endl<<" array after calling function fun_2 " <<endl;  
for(int i=0;i<5;i++)cout<<m[i]<<" "};
```



## Solution

array

5 6 7 8 9

x 5 y 9

array after calling function fun\_1

5 6 7 8 -7

array in function 2

5 6 7 8 -7

array in function 2

0 2 4 6 8

array after calling function fun\_2

0 2 4 6 8

// return many values from function

// use function to get sum of array, maximum value and its location

```
int fun(int k[ ],int &m, int &z )
```

```
{ int a=0; m=0; z=0;
```

```
  cout<<endl<<" array in function "<<endl;
```

```
  for(int i=0;i<5;i++) cout<<k[i]<<"  ";
```

```
  for(int i=0;i<5;i++){ a+=k[i];
```

```
    if(k[i]>m){m=k[i]; z=i;} }
```

```
  return a; }
```

```
main( ){ int m,z, h; int x[5]= {10, 3, 7, 50, 20};
```

```
  cout<<" array in main "<<endl;
```

```
  for(int i=0;i<5;i++) cout<<x[i]<<"  ";
```

```
  cout<<endl<<endl;
```

```
  h=fun(x,m,z); cout<<endl<<" returned values from function "<<endl<<endl;
```

```
  cout<<" sum = "<<h<<"  max value = "<<m<<"  at index "<<z<<endl;}
```

## **Solution**

array in main

10 3 7 50 20

array in function

10 3 7 50 20

returned values from function

sum = 90   max value = 50   at index 3

```
// program to read scores of n students in m subjects and get average score
// Use function to get average score of any student
// function has array as argument
```

```
float average(float s[100][10],int k, int l);
main( ) { int i,n,m;
float score[100][10], av[100];
cin>>n>>m;
    for( i=0; i<n; i++)
        {for(int j=0;j<m;j++)
            { cin>>score[i][j]; cout<<score[i][j]<<" ";}
        cout<<endl;}
    for( i=0;i<n;i++)
        {av[i]=average(score, m, i);
        cout<<" student number " <<(i+1)<<" average " <<av[i]<<endl;} }
```

```
float average(float s[100][10],int k, int l)
{float v=0;
for(int j=0;j<k;j++)
    v=v+s[l][j];    v=v/k;    return v;}
```

```

// Use Array as global
float s[10][6], av[10];
void average(int k, int l);
main( ) { int i,n,m;
cin>>n>>m;
for( i=0; i<n; i++)
{for(int j=0;j<m;j++)
{ cin>>s[i][j]; cout<<s[i][j]<<" ";}
cout<<endl;}
for( i=0;i<n;i++)
{average(m, i);
cout<<" student number " <<(i+1)<<" average
"<<av[i]<<endl;}}

```

```

void average(int k, int l)
{av[l]=0;
for(int j=0;j<k;j++)
av[l]=av[l]+s[l][j];
av[l]=av[l]/k; }

```