Lecture 2
Arrays, Pointers and References in C++

C++ Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Declare an array

To declare an array, define the variable type, specify the name of the array followed by **square brackets** and specify the number of elements it should store.

Syntax

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

Example

```
string cars[4];
```

We have now declared a variable that holds an array of four strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

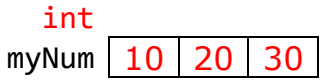
```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

string cars	"Volvo"	"BMW",	"Ford"	"Mazda"

To create an array of three integers, you could write:

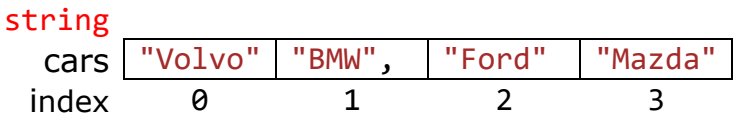
Example

```
int myNum[3] = {10, 20, 30};
```



Access the Elements of an Array

You access an array element by referring to the **index** number.



This statement accesses the value of the **first element** in **cars**:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];
// Outputs Volvo
```

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];
// Now outputs Opel instead of Volvo
```

Loop Through an Array

You can loop through the array elements with the `for` loop.

The following example outputs all elements in the **cars** array:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
for(int i = 0; i < 4; i++) {
  cout << cars[i] << "\n";
}
```

The following example outputs the index of each element together with its value:

Example

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
for(int i = 0; i < 4; i++) {
  cout << i << ": " << cars[i] << "\n";
}
```

Omit Array Size

You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it:

```
string cars[] = {"Volvo", "BMW", "Ford"};//size of array is always 3
```

This is completely fine. However, the problem arise if you want extra space for future elements. Then you have to overwrite the existing values:

```
~~string cars[] = {"Volvo", "BMW", "Ford"};~~
string cars[] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
```

If you specify the size however, the array will reserve the extra space:

```
string cars[5] = {"Volvo", "BMW", "Ford"}; // size of array is 5,
even though it's only three elements inside it
```

Now you can add a fourth and fifth element without overwriting the others:

```
cars[3] = "Mazda";
cars[4] = "Tesla";
```

Omit Elements on Declaration

It is also possible to declare an array without specifying the elements on declaration, and add them later:

```
string cars[5];
cars[0] = "Volvo";
cars[1] = "BMW";
...
```

Multidimensional arrays

In C++, we can create an array of an array, known as a multidimensional array.

Syntax

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

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 think 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]

Example

```
int  test[2][3] = { {2, 4, 5}, {9, 0, 19}};
```

	Col 1	Col 2	Col 3
Row 1	2	4	5
Row 2	9	0	19

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;
}
```

Three-dimensional array

```
int test[2][3][4] = {
              { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
              { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
               };
```

3	4	2	3	13	4	56	3
0	-3	9	11	5	9	3	5
23	12	23	2	5	1	4	9

The first dimension has the value 2. So, the two elements comprising the first dimension are:

```
Element 1 = { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} }
Element 2 = { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
```

The **second** dimension has the value 3. Notice that each of the elements of the first dimension has three elements each:

```
{3, 4, 2, 3}, {0, -3, 9, 11} and {23, 12, 23, 2} for Element 1.
{13, 4, 56, 3}, {5, 9, 3, 5} and {5, 1, 4, 9} for Element 2.
```

Finally, there are four int numbers inside each of the elements of the **second** dimension:

Example

```
// C++ Program to Store value entered by user in
// three dimensional array and display it.

#include <iostream>
using namespace std;

int main() {
    // This array can store up to 12 elements (2x3x2)
    int test[2][3][2] = {
                                {
                                    {1, 2},
                                    {3, 4},
                                    {5, 6}
                                },
                                {
                                    {7, 8},
                                    {9, 10},
                                    {11, 12}
                                }
                            };

    // Displaying the values with proper index.
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                cout << "test[" << i << "][" << j << "][" << k << "] =
                        " << test[i][j][k] << endl;
            }
        }
    }
```

```
    return 0;
}
```

The basic concept of printing elements of a 3d array is similar to that of a 2d array.

However, since we are manipulating 3 dimensions, we use a nested for loop with 3 total loops instead of just 2:

the outer loop from i == 0 to i == 1 accesses the first dimension of the array

the middle loop from j == 0 to j == 2 accesses the second dimension of the array

the innermost loop from k == 0 to k == 1 accesses the third dimension of the array

As we can see, the complexity of the array increases exponentially with the increase in dimensions.

Dealing with strings in C++

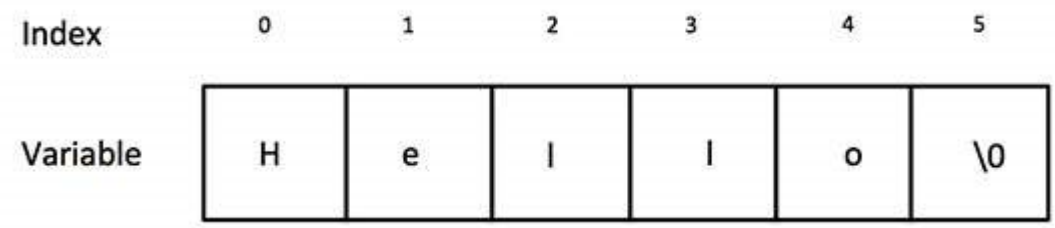
This string is a one-dimensional array of characters which is terminated by a **null** character **'\0'**. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

Example

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o'};
```

If you follow the rule of array initialization, then you can write the above statement as follows –

```
char greeting[] = "Hello";
```



The C++ compiler automatically places the '\0' at the end of the string when it initializes the array.

String Concatenation

The `+` operator can be used between strings to add them together to make a new string. This is called **concatenation**:

Example

```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName + " " + lastName;
cout << fullName;
```

C++ Pointers

In C++, pointers are variables that store the **memory addresses** of other variables.

Declaring pointers

Synax

```
type* name;
```

where **type** is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to.

A pointer variable points to a data type (like `int` or `string`) of the same type, and is created with the `*` operator. The address of the variable you're working with is assigned to the pointer.

Example

```
int* number;

char* character;

double* decimals;
```

Tip: There are three ways to declare pointer variables, but the first way is preferred:

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

Example

```
int* pointVar, var;
```

Here, we have declared a pointer pointVar and a normal variable var.

Assigning Addresses to Pointers

Example

```
int* pointVar, var;

var = 5;

// assign address of var to pointVar pointer

pointVar = &var;
```

Here, 5 is assigned to the variable var. And, the address of var is assigned to the pointVar pointer with the code pointVar = &var.

& is the address-of operator, and can be read simply as "address of"

***** is the dereference operator, and can be read as "value pointed to by"

Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the * operator. For example:

```
int* pointVar, var;

var = 5;

// assign address of var to pointVar

pointVar = &var;



// access value pointed by pointVar

cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to pointVar. We have used the *pointVar to get the value stored in that address.

When * is used with pointers, it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, *pointVar = var.

Modify the Pointer Value

You can also change the pointer's value. But note that this will also change the value of the original variable:

Example

```
string food = "Pizza";
string* ptr = &food;

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Access the memory address of food and output its value (Pizza)
cout << *ptr << "\n";

// Change the value of the pointer
*ptr = "Hamburger";

// Output the new value of the pointer (Hamburger)
```

```
cout << *ptr << "\n";

// Output the new value of the food variable (Hamburger)
cout << food << "\n";
```

C++ References

A reference variable is a "reference" to an existing variable, and it is created with the `&` operator:

Declaring References

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food
```

Now, we can use either the variable name `food` or the reference name `meal` to refer to the `food` variable:

Example

```
string food = "Pizza";
string &meal = food;

cout << food << "\n";  // Outputs Pizza
cout << meal << "\n";  // Outputs Pizza
```

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are −

- Once a reference is initialized to an object, it **cannot be changed** to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be **initialized** when it is created. Pointers can be initialized at any time.

The `&` operator was used to create a reference variable. But it can also be used to get the **memory address** of a variable; which is the location of where the variable is stored on the computer.