C++
For Engineers

(full-width.html)

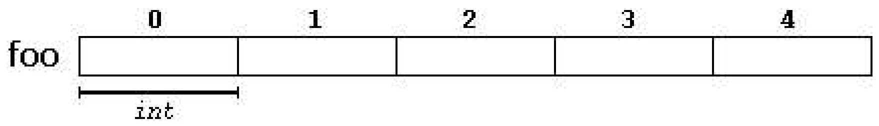
Arrays

ARRAYS IN C++

An array is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier.

Five values of type int can be declared as an array without having to declare five different variables (each with its own identifier).

For example, a five element integer array foo may be logically represented as;



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, with 0 being the first while 4 being the last; In C++, the index of the first array element is always zero. As expected, an n array must be declared prior its use. A typical declaration for an array in C++ is:

```
type name [elements];
```

where type is a valid type (such as `int`, `float` ...), name is a valid identifier and the elements field (which is always enclosed in square brackets `[]`), specifies the size of the array.

Thus, the foo array, with five elements of type `int`, can be declared as:

```
int foo [5];
```

NOTE

: The elements field within square brackets [], representing the number of elementsin the array, must be a constant expression, since arrays are blocks of static memory whose size must be known at compile time.

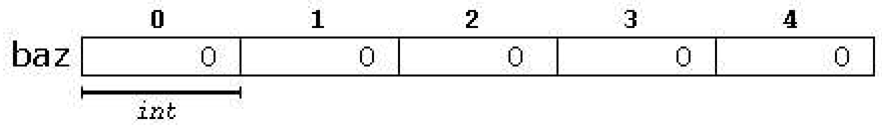
INITIALIZING ARRAYS

By default, are left uninitialized. This means that none of its elements are set to anyparticular value; their contents are undetermined at the point the array is declared.

The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

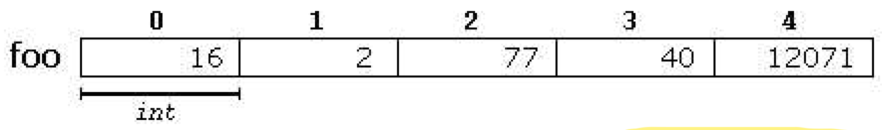
This creates an array of five int values, each initialized with a value of zero:



But, the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

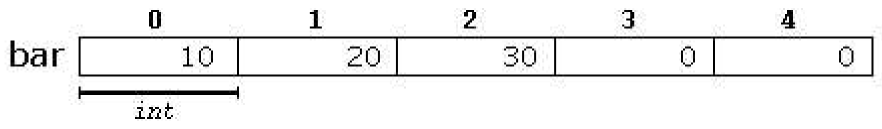
This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

Will create an array like this:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty[]. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array `foo` would be five `int` long, since we have provided five initialization values.

Finally, the evolution of C++ has led to the adoption of `universal initialization` also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

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

Here, the number of the array n is calculated by the compiler by using the formula n= #of initializers/sizeof(int).

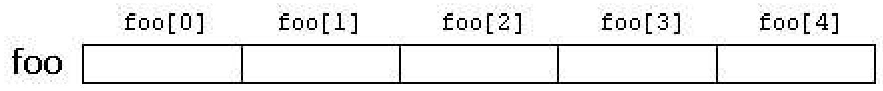
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

ARRAY ACCESSING

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

```
name[index]
```

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the third element of `foo` :

```
foo [2] = 75;
```

and, for example, the following copies the value of the fourth element of `foo` to a variable called x:

```
x = foo[3];
```

Therefore, the expression `foo[2]` or `foo[4]` is always evaluated to an int. Notice that the third element of `foo` is specified `foo[2]` , the second one is `foo[1]` , since the first one is `foo[0]` . It's last element is therefore `foo[4]` . If we write `foo[5]` , we would be accessing the sixth element of `foo` , and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed because index checking slows down program execution. At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets `[]` with arrays.

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
foo[0] = a;
foo[i] = 75;
b = foo [i+2];
foo[foo[i]] = foo[i] + 5;
```

FOR EXAMPLE:

```

// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int i, result=0;

int main ()
{
  for ( i=0 ; i<5 ; i++ )
  {
    result += foo[i];
  }
  cout << result;
  return 0;
}
```

```
12206
```

MULTIDIMENSIONAL ARRAYS

Multidimensional arrays can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a two-dimensional table made of elements, all of them hold same type of elements.

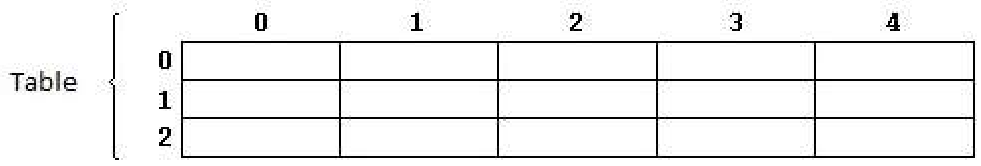
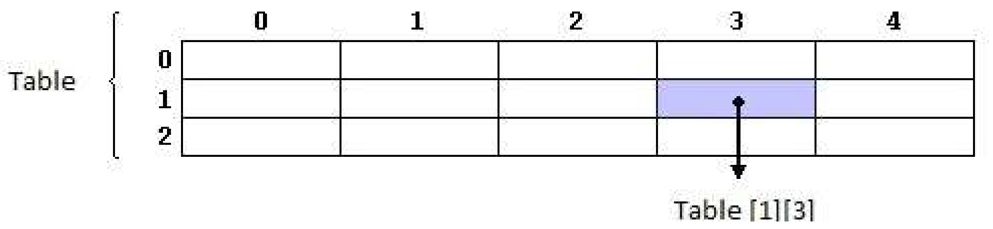


Table represents a bi-dimensional array of 3 per 5 elements of type int. The C++ syntax for this is

```
int Table [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
Table[1][3]
```



(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

```
char century [100][365][24][60][60];
```

Declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory! And such a declaration is highly improbable and it underscores inefficient use of memory space.

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

```
int Table [3][5];   // is equivalent to
int Table [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bi-dimensional array while the other uses a simple array:

MULTIDIMENSIONAL ARRAY

```
const int WIDTH = 5;
const int HEIGHT = 3;

int Table [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      Table[n][m]=(n+1)*(m+1);
    }
}
```

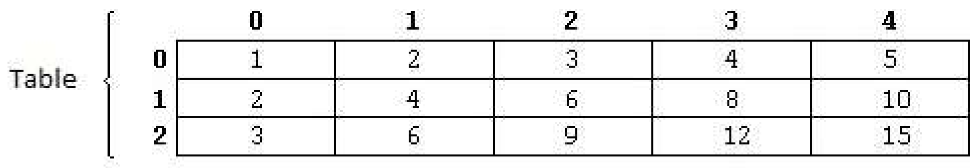
PSEUDO-MULTIDIMENSIONAL ARRAY

```
const int WIDTH = 5;
const int HEIGHT = 3;

int Table [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      Table[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:



Table

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Note that the code uses named constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

USING LOOP TO INPUT AN TWO-DIMENSIONAL ARRAY FROM USER

```
int mat[3][5], row, col ;
for (row = 0; row < 3; row++)
for (col = 0; col < 5; col++)
cin >> mat[row][col];
```

ARRAYS AS PARAMETERS

Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference. This means that the function can directly access and modified the contents of the passed array. When declaring a two-dimensional array as a formal parameter, we can omit the size of the first dimension, but not the second; that is, we must specify the number of columns. For example:

```
void print(int A[][3],int N, int M)
```

In order to pass to this function an array declared as:

```
int arr[4][3];
```

we need to write a call like this:

```
print(arr);
```

HERE IS A COMPLETE EXAMPLE:

```
#include <iostream>
using namespace std;
void print(int A[][3],int N, int M)
{
  for (R = 0; R < N; R++)
    for (C = 0; C < M; C++)
       cout << A[R][C];
}
int main ()
{
  int arr[4][3] ={{12, 29, 11},
                  {25, 25, 13},
                  {24, 64, 67},
                  {11, 18, 14}};
  print(arr,4,3);
  return 0;
}
```

Engineers use two dimensional arrays in order to represent matrices. The code for a function that finds the sum of the two matrices A and B are shown below.

FUNCTION TO FIND THE SUM OF TWO MATRICES

```
void Addition(int A[][20], int B[][20],int N, int M)
{
  for (int R=0;R<N;R++)
    for(int C=0;C<M;C++)
      C[R][C]=A[R][C]+B[R][C];
}
```

FUNCTION TO FIND OUT TRANSPOSE OF A MATRIX A

```

 void Transpose(int A[][20], int B[][20],int N, int M)
{
  for(int R=0;R<N;R++)
    for(int C=0;C<M;C++)
       B[R][C]=A[C][R];
}
```

ARRAYS AS PARAMETERS

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

CODE

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
        cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

SOLUTION

```
5 10 15
2 4 6 8 10
```

In the code above, the first parameter (`int arg[]`) accepts any array whose elements are of type int, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter.

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

RELATED VIDEOS

Passing Array to function

Multidimensional Arrays

Print out Multidemensioanl Arrays

Arrays

Create an array using Loop

Using Arrays in calculation

✓ **RETURN TO TOP**

Go back to top menu