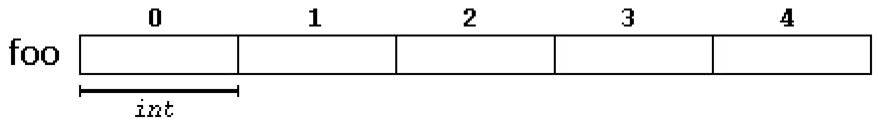
11. Arrays

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

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be 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, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. 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 length of the array in terms of the number of elements.

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

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

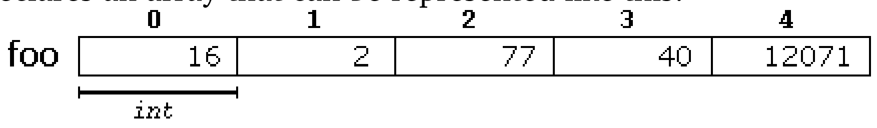
11.1 Initializing arrays

By default, regular arrays of local scope (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

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 };
```

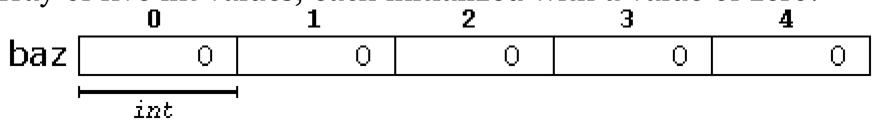
This statement declares an array that can be represented like this:



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

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

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



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 5 int long, since we have provided 5 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 };
```

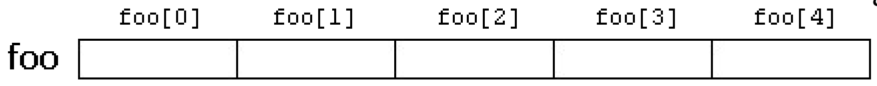
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

11.2 Accessing the values of an array

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 third element of foo to a variable called x:

```
    x = foo[2];
```

Therefore, the expression foo[2] is itself a variable of type int.

Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element is foo[4]. Therefore, 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 will be seen in a later chapter when pointers are introduced.

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

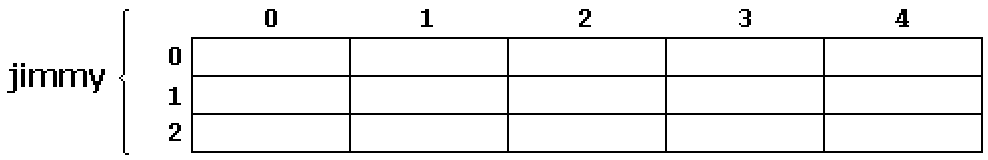
For example:

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

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

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

```
12206
```

11.3 Multidimensional arrays

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

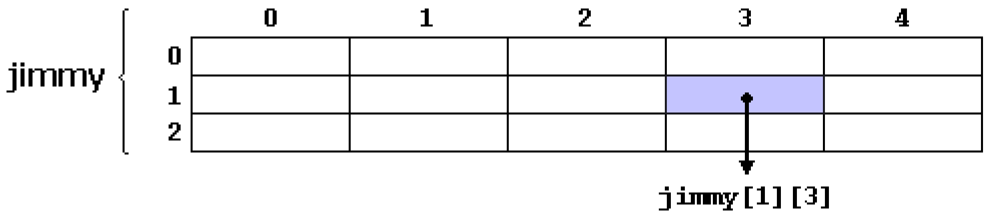


jimmy represents a bi-dimensional array of 3 per 5 elements of type int. The C++ syntax for this is:

```
    int jimmy [3][5];
```

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

```
    jimmy[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!

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:

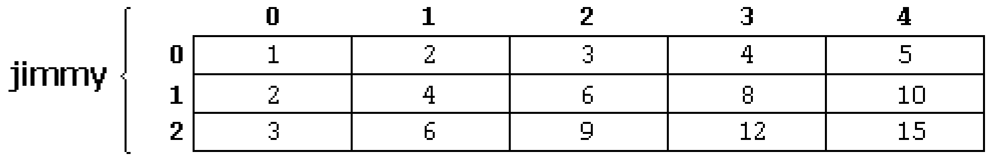
```
    int jimmy [3][5];   // is equivalent to
    int jimmy [15];     // (3 * 5 = 15)
```

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 bidimensional array while the other uses a simple array:

multidimensional array	pseudo-multidimensional array

`#define WIDTH 5`
`#define HEIGHT 3`
 	
`int jimmy [HEIGHT][WIDTH];`
`int n,m;`
 	
`int main ()`
`{`
`  for (n=0; n<HEIGHT; n++)`
`    for (m=0; m<WIDTH; m++)`
`    {`
`      jimmy[n][m]=(n+1)*(m+1);`
`    }`
`}`
	`#define WIDTH 5`
	`#define HEIGHT 3`
	
	`int jimmy [HEIGHT * WIDTH];`
	`int n,m;`
	
	`int main ()`
	`{`
	`  for (n=0; n<HEIGHT; n++)`
	`    for (m=0; m<WIDTH; m++)`
	`    {`
	`      jimmy[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:



jimmy	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 defined 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.

11.4 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 is a complete example:

```
// 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);
}
```

```
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. This allows the for loop that prints out the array to know the range to iterate in the array passed, without going out of range.

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.

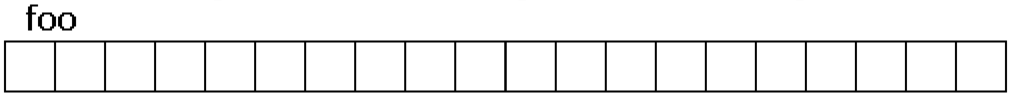
11.5 Character sequences

The string class has been briefly introduced in an earlier chapter. It is a very powerful class to handle and manipulate strings of characters. However, because strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

For example, the following array:

```
    char foo [20];
```

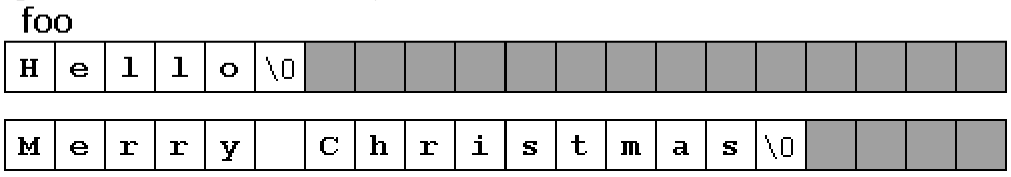
is an array that can store up to 20 elements of type char. It can be represented as:



Therefore, this array has a capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences. For example, at some point in a program, either the sequence "Hello" or the sequence "Merry Christmas" can be stored in foo, since both would fit in a sequence with a capacity for 20 characters.

By convention, the end of strings represented in character sequences is signaled by a special character: the null character, whose literal value can be written as '\0' (backslash, zero).

In this case, the array of 20 elements of type char called foo can be represented storing the character sequences "Hello" and "Merry Christmas" as:



Notice how after the content of the string itself, a null character ('\0') has been added in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

11.6 Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays, they follow the same rules as these. For example, to initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The above declares an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a null character '\0' at the end.

But arrays of character elements have another way to be initialized: using string literals directly.

In the expressions used in some examples in previous chapters, string literals have already shown up several times. These are specified by enclosing the text between double quotes ("). For example:

```
"the result is: "
```

This is a string literal, probably used in some earlier example.

Sequences of characters enclosed in double-quotes (") are literal constants. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

Therefore, the array of char elements called **`myword`** can be initialized with a null-terminated sequence of characters by either one of these two statements:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword[] = "Hello";
```

In both cases, the array of characters **`myword`** is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0'), which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that here we are talking about initializing an array of characters at the moment it is being declared, and not about assigning values to them later (once they have already been declared). In fact, because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.

Expressions (once **myword** has already been declared as above), such as:

```
    myword = "Bye";
    myword[] = "Bye";
```

would not be valid, like neither would be:

```
    myword = { 'B', 'y', 'e', '\0' };
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

```
    myword[0] = 'B';
    myword[1] = 'y';
    myword[2] = 'e';
    myword[3] = '\0';
```

11.7 Strings and null-terminated character sequences

Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as C-strings). In C++, even though the standard library defines a specific type for strings (class string), still, plain arrays with null-terminated sequences of characters (C-strings) are a natural way of representing strings in the language; in fact, string literals still always produce null-terminated character sequences, and not string objects.

In the standard library, both representations for strings (C-strings and library strings) coexist, and most functions requiring strings are overloaded to support both.

For example, cin and cout support null-terminated sequences directly, allowing them to be directly extracted from cin or inserted into cout, just like strings. For example:

```
    // strings and NTCS:
    #include <iostream>
    #include <string>
    using namespace std;

    int main ()
    {
      char question1[] = "What is your name? ";
      string question2 = "Where do you live? ";
      char answer1 [80];
      string answer2;
      cout << question1;
      cin >> answer1;
      cout << question2;
      cin >> answer2;
      cout << "Hello, " << answer1;
      cout << " from " << answer2 << "!\n";
      return 0;
    }
```

```
What is your name? Jafor
Where do you live? Bangladesh
Hello, Jafor from Bangladesh!
```

In this example, both arrays of characters using null-terminated sequences and strings are used. They are quite interchangeable in their use together with cin and cout, but there is a notable difference in their declarations: arrays have a fixed size that needs to be specified either implicit or explicitly when declared; question1 has a size of exactly 20 characters (including the terminating null-characters) and answer1 has a size of 80 characters; while strings are simply strings, no size is specified. This is due to the fact that strings have a dynamic size determined during runtime, while the size of arrays is determined on compilation, before the program runs.

In any case, null-terminated character sequences and strings are easily transformed from one another:

Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string's member functions c_str or data:

```
char myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to string
cout << mystring;          // printed as a library string
cout << mystring.c_str();  // printed as a c-string
```