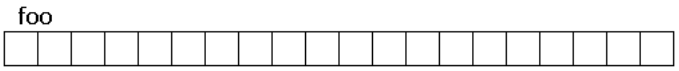
# Character sequences

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];
```

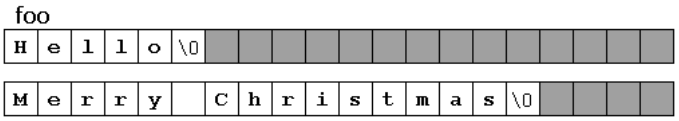
is an array that can store up to 20 elements of type `char`. It can be represented as:




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.

## 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:

```
1 char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 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:

```
1 myword = "Bye";
2 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';
```

## 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? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

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
```

(note: both `c_str` and `data` members of `string` are equivalent)

Previous: **Arrays** Index Next: **Pointers**