C++ Reference Material
Strings in C and C++

This page summarizes many of the things you may find it useful to know when working with either C-strings or objects of the C++ `string` class.

The term *string* generally means an ordered sequence of characters, with a first character, a second character, and so on, and in most programming languages such strings are enclosed in either single or double quotes. In C++ the enclosing delimiters are double quotes. In this form the string is referred to as a *string literal* and we often use such string literals in output statements when we wish to display text on the screen for the benefit of our users. For example, the usual first C++ program displays the string literal "Hello, world!" on the screen with the following output statement:

```
cout << "Hello, world!" << endl;
```

However, without string variables about all we can do with strings is output string literals to the screen, so we need to expand our ability to handle string data. When we talk about strings in C++, we must be careful because the C language, with which C++ is meant to be backward compatible, had one way of dealing with strings, while C++ has another, and to further complicate matters there are many non-standard implementations of C++ strings. These should gradually disappear as compiler vendors update their products to implement the string component of the C++ Standard Library.

As a programmer, then, you must distinguish between the following three things:

1. An "ordinary" array of characters, which is just like any other array and has no special properties that other arrays do not have.
2. A C-string, which consists of an array of characters terminated by the null character '\0', and which therefore is different from an ordinary array of characters. There is a whole library of functions for dealing with strings represented in this form. Its header file is `<cstring>`. In some implementations this library may be automatically included when you include other libraries such as the `<iostream>` library. Note that the null character may very well not be the very last character in the C-string array, but it will be the first character beyond the last character of the actual string data in in that array. For example if you have a C-string storing "Hello" in a character array of size 10, then the letters of the word "Hello" will be in positions with indices 0 to 4, there will be a null character at index 5, and the locations with indices 6 to 9 will contain who-knows-what. In any case, it's the null character at index 5 that makes this otherwise ordinary character array a C-string.
3. A C++ string object, which is an instance of a "class" data type whose actual internal representation you need not know or care about, as long as you know what you can and can't do with variables (and constants) having this data type. There is a library of C++ string functions as well, available by including the `<string>` header file.

Both the C-string library functions and the C++ string library functions are available to C++ programs. But, don't forget that these are two *different* function libraries, and the functions of the first library have a different notion of what a string is from the corresponding notion held by the functions of the second library. There are two further complicating aspects to this situation:

first, though a function from one of the libraries may have a counterpart in the other library (i.e., a function in the other library designed to perform the same operation), the functions may not be used in the same way, and may not even have the same name; second, because of backward compatibility many functions from the C++ string library can be expected to work fine and do the expected thing with C-style strings, but not the other way around.

The last statement above might seem to suggest we should use C++ strings and forget about C-strings altogether, and it is certainly true that there is a wider variety of more intuitive operations available for C++ strings. However, C-strings are more primitive, you may therefore find them simpler to deal with (provided you remember a few simple rules, such as the fact that the null character must always terminate such strings), and certainly if you read other, older programs you will see lots of C-strings. So, use whichever you find more convenient, but if you choose C++ strings and occasionally need to mix the two for some reason, be extra careful. Finally, there are certain situations in which C-strings *must* be used.

To understand strings, you will have to spend some time studying sample programs. This study must include the usual prediction of how you expect a program to behave for given input, followed by a compile, link and run to test your prediction, as well as subsequent modification and testing to investigate questions that will arise along the way. In addition to experimenting with any supplied sample programs, you should be prepared to make up your own.

In the following examples we attempt to draw the distinction between the two string representations and their associated operations. The list is not complete, but we do indicate how to perform many of the more useful kinds of tasks with each kind of string. The left-hand column contains examples relevant to C-strings and the right-hand column shows analogous examples in the context of C++ strings.

```
C-strings  (#include <cstring>)        C++ strings  (#include <string>)
===============================        ================================
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Declaring a C-string variable          Declaring a C++ string object
-----------------------------          -----------------------------
char str[10];                          string str;

Initializing a C-string variable       Initializing a C++ string object
--------------------------------       --------------------------------
char str1[11] = "Call home!";          string str1("Call home!");
char str2[] = "Send money!";           string str2 = "Send money!";
char str3[] = {'O', 'K', '\0'};        string str3("OK");
Last line above has same effect as:
char str3[] = "OK";
                                       string str4(10, 'x');

Assigning to a C-string variable       Assigning to a C++ string object
--------------------------------       --------------------------------
Can't do it, i.e., can't do this:      string str;
char str[10];                          str = "Hello";
str = "Hello!";                        str = otherString;

Concatenating two C-strings            Concatenating two C++ string objects
---------------------------            ------------------------------------
strcat(str1, str2);                    str1 += str2;
strcpy(str, strcat(str1, str2));       str = str1 + str2;

Copying a C-string variable            Copying a C++ string object
```

```
---------------------------               ---------------------------
char str[20];                             string str;
strcpy(str, "Hello!");                    str = "Hello";
strcpy(str, otherString);                 str = otherString;

Accessing a single character              Accessing a single character
----------------------------              ----------------------------
str[index]                                str[index]
                                          str.at(index)
                                          str(index, count)

Comparing two C-strings                   Comparing two C++ string objects
-----------------------                   --------------------------------
if (strcmp(str1, str2) < 0)               if (str1 < str2)
    cout << "str1 comes 1st.";                cout << "str1 comes 1st.";
if (strcmp(str1, str2) == 0)              if (str1 == str2)
    cout << "Equal strings.";                 cout << "Equal strings.";
if (strcmp(str1, str2) > 0)               if (str1 > str2)
    cout << "str2 comes 1st.";                cout << "str2 comes 1st.";

Finding the length of a C-string          Finding the length of a C++ string object
--------------------------------          -----------------------------------------
strlen(str)                               str.length()

Output of a C-string variable             Output of a C++ string object
-----------------------------             -----------------------------
cout << str;                              cout << str;
cout << setw(width) << str;               cout << setw(width) << str;
```

In what follows, keep in mind that `cin` ignores white space when reading a string, while `cin.get()`, `cin.getline()` and `getline()` do not. Remember too that `cin.getline()` and `getline()` consume the delimiter while `cin.get()` does not. Finally, `cin` can be replaced with any open input stream, since file input with `inFile`, say, behaves in a manner completely analogous to the corresponding behavior of `cin`. Analogously, in the output examples given immediately above, `cout` could be replaced with any text output stream variable, say `outFile`. In all cases, `numCh` is the maximum number of characters that will be read.

```
Input of a C-style string variable           Input of a C++ string object
----------------------------------           ----------------------------
cin >> s;                                    cin >> s;
cin.get(s, numCh+1);
cin.get(s, numCh+1,'\n');
cin.get(s, numCh+1,'x');
cin.getline(s, numCh+1);                     getline(cin, s);
cin.getline(s, numCh+1, '\n');
cin.getline(s, numCh+1, 'x');                getline(cin, s, 'x');
```

A useful naming convention for C-strings is illustrated by examples like

```
typedef char String80[81];
typedef char String20[21];
```

in which the two numbers in each definition differ by 1 to allow for the null character '\0' to be stored in the array of characters, but to *not* be considered as part of the string stored there. No analog to this naming convention is necessary for C++ strings, since for all practical purposes, each C++ string variable may contain a string value of virtually unlimited length.