10.6 — C-style strings

ALEX AUGUST 27, 2021

In lesson 4.12 -- An introduction to std::string we defined a string as a collection of sequential characters, such as "Hello, world!". Strings are the primary way in which we work with text in C++, and std::string makes working with strings in C++ easy.

Modern C++ supports two different types of strings: std::string (as part of the standard library), and C-style strings (natively, as inherited from the C language). It turns out that std::string is implemented using C-style strings. In this lesson, we'll take a closer look at C-style strings.

C-style strings

A C-style string is simply an array of characters that uses a null terminator. A null terminator is a special character ('\0', ascii code 0) used to indicate the end of the string. More generically, A C-style string is called a null-terminated string.

To define a C-style string, simply declare a char array and initialize it with a string literal:

```
char myString[]{ "string"
};
```

Although "string" only has 6 letters, C++ automatically adds a null terminator to the end of the string for us (we don't need to include it ourselves). Consequently, myString is actually an array of length 7!

We can see the evidence of this in the following program, which prints out the length of the string, and then the ASCII values of all of the characters:

```
#include <iostream>
#include <iterator> // for std::size

int main()
{
    char myString[]{ "string" };
    const int length{ static_cast<int>(std::size(myString)) };
//  const int length{ sizeof(myString) / sizeof(myString[0]) }; // use instead if not C++17 capable
    std::cout << myString << " has " << length << " characters.\n";

    for (int index{ 0 }; index < length; ++index)
        std::cout << static_cast<int>(myString[index]) << ' ';

    std::cout << '\n';

    return 0;
}
```

This produces the result:

```
string has 7 characters.
115 116 114 105 110 103 0
```

That 0 is the ASCII code of the null terminator that has been appended to the end of the string.

When declaring strings in this manner, it is a good idea to use [] and let the compiler calculate the length of the array. That way if you change the string later, you won't have to manually adjust the array length.

One important point to note is that C-style strings follow *all* the same rules as arrays. This means you can initialize the string upon creation, but you can not assign values to it using the assignment operator after that!

```
1 char myString[]{ "string" }; // ok
  myString = "rope"; // not ok!
```

Since C-style strings are arrays, you can use the [] operator to change individual characters in the string:

```
1 #include <iostream>
2
3 int main()
4 {
5     char myString[]{ "string" };
      myString[1] = 'p';
      std::cout << myString << '\n';
6
      return 0;
  }
```

This program prints:

```
spring
```

When printing a C-style string, std::cout prints characters until it encounters the null terminator. If you accidentally overwrite the null terminator in a string (e.g. by assigning something to myString[6]), you'll not only get all the characters in the string, but std::cout will just keep printing everything in adjacent memory slots until it happens to hit a 0!

Note that it's fine if the array is larger than the string it contains:

```
1 #include <iostream>
2
3 int main()
4 {
5     char name[20]{ "Alex" }; // only use 5 characters (4 letters + null terminator)
      std::cout << "My name is: " << name << '\n';

      return 0;
  }
```

In this case, the string "Alex" will be printed, and std::cout will stop at the null terminator. The rest of the characters in the array are ignored.

C-style strings and std::cin

There are many cases where we don't know in advance how long our string is going to be. For example, consider the problem of writing a program where we need to ask the user to enter their name. How long is their name? We don't know until they enter it!

In this case, we can declare an array larger than we need:

```cpp
#include <iostream>

int main()
{
    char name[255] {}; // declare array large enough to hold 254 characters + null terminator
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

In the above program, we've allocated an array of 255 characters to name, guessing that the user will not enter this many characters. Although this is commonly seen in C/C++ programming, it is poor programming practice, because nothing is stopping the user from entering more than 254 characters (either unintentionally, or maliciously).

The recommended way of reading C-style strings using `std::cin` is as follows:

```cpp
#include <iostream>
#include <iterator> // for std::size

int main()
{
    char name[255] {}; // declare array large enough to hold 254 characters + null terminator
    std::cout << "Enter your name: ";
    std::cin.getline(name, std::size(name));
    std::cout << "You entered: " << name << '\n';

    return 0;
}
```

This call to cin.getline() will read up to 254 characters into name (leaving room for the null terminator!). Any excess characters will be discarded. In this way, we guarantee that we will not overflow the array!

Manipulating C-style strings

C++ provides many functions to manipulate C-style strings as part of the <cstring> header. Here are a few of the most useful:

strcpy() allows you to copy a string to another string. More commonly, this is used to assign a value to a string:

```cpp
#include <cstring>
#include <iostream>

int main()
{
    char source[]{ "Copy this!" };
    char dest[50];
    std::strcpy(dest, source);
    std::cout << dest << '\n'; // prints "Copy this!"

    return 0;
}
```

However, strcpy() can easily cause array overflows if you're not careful! In the following program, dest isn't big enough to hold the entire

string, so array overflow results.

```
#include <cstring>
#include <iostream>

int main()
{
    char source[]{ "Copy this!" };
    char dest[5]; // note that the length of dest is only 5 chars!
    std::strcpy(dest, source); // overflow!
    std::cout << dest << '\n';

    return 0;
}
```

Many programmers recommend using strncpy() instead, which allows you to specify the size of the buffer, and ensures overflow doesn't occur. Unfortunately, strncpy() doesn't ensure strings are null terminated, which still leaves plenty of room for array overflow.

In C++11, strcpy_s() is preferred, which adds a new parameter to define the size of the destination. However, not all compilers support this function, and to use it, you have to define STDC_WANT_LIB_EXT1 with integer value 1.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <cstring> // for strcpy_s
#include <iostream>

int main()
{
    char source[]{ "Copy this!" };
    char dest[5]; // note that the length of dest is only 5 chars!
    strcpy_s(dest, 5, source); // A runtime error will occur in debug mode
    std::cout << dest << '\n';

    return 0;
}
```

Because not all compilers support strcpy_s(), strlcpy() is a popular alternative -- even though it's non-standard, and thus not included in a lot of compilers. It also has its own set of issues. In short, there's no universally recommended solution here if you need to copy a C-style string.

Another useful function is the strlen() function, which returns the length of the C-style string (without the null terminator).

```
#include <iostream>
#include <cstring>
#include <iterator> // for std::size

int main()
{
    char name[20]{ "Alex" }; // only use 5 characters (4 letters + null terminator)
    std::cout << "My name is: " << name << '\n';
    std::cout << name << " has " << std::strlen(name) << " letters.\n";
    std::cout << name << " has " << std::size(name) << " characters in the array.\n"; // use sizeof(name) / sizeof(name[0]) if not C++17 capable

    return 0;
}
```

The above example prints:

```
My name is: Alex
Alex has 4 letters.
Alex has 20 characters in the array.
```

Note the difference between strlen() and std::size(). strlen() prints the number of characters before the null terminator, whereas std::size (or the sizeof() trick) returns the size of the entire array, regardless of what's in it.

Other useful functions:
strcat() -- Appends one string to another (dangerous)
strncat() -- Appends one string to another (with buffer length check)
strcmp() -- Compare two strings (returns 0 if equal)
strncmp() -- Compare two strings up to a specific number of characters (returns 0 if equal)

Here's an example program using some of the concepts in this lesson:

```
#include <cstring>
#include <iostream>
#include <iterator> // for std::size

int main()
{
    // Ask the user to enter a string
    char buffer[255] {};
    std::cout << "Enter a string: ";
    std::cin.getline(buffer, std::size(buffer));

    int spacesFound{ 0 };
    int bufferLength{ static_cast<int>(std::strlen(buffer))
};
    // Loop through all of the characters the user entered
    for (int index{ 0 }; index < bufferLength; ++index)
    {
        // If the current character is a space, count it
        if (buffer[index] == ' ')
            ++spacesFound;
    }

    std::cout << "You typed " << spacesFound << "
spaces!\n";

    return 0;
}
```

Note that we put `strlen(buffer)` outside the loop so that the string length is only calculated once, not every time the loop condition is checked.

Don't use C-style strings

It is important to know about C-style strings because they are used in a lot of code. However, now that we've explained how they work, we're going to recommend that you avoid them altogether whenever possible! Unless you have a specific, compelling reason to use C-style strings, use std::string (defined in the <string> header) instead. std::string is easier, safer, and more flexible. In the rare case that you do need to work with fixed buffer sizes and C-style strings (e.g. for memory-limited devices), we'd recommend using a well-tested 3rd party string library designed for the purpose, or `std::string_view`, which is covered in the next lesson, instead.

Rule

Use `std::string` or `std::string_view` (next lesson) instead of C-style strings.

Next lesson
10.7 An introduction to std::string_view

Back to table of contents

Previous lesson
10.5 Multidimensional Arrays

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies:

POST COMMENT

260 COMMENTS

Newest