22.1 — std::string and std::wstring

ALEX AUGUST 12, 2023

The standard library contains many useful classes -- but perhaps the most useful is std::string. std::string (and std::wstring) is a string class that provides many operations to assign, compare, and modify strings. In this chapter, we'll look into these string classes in depth.

Note: C-style strings will be referred to as "C-style strings", whereas std::string (and std::wstring) will be referred to simply as "strings".

Author's note

This chapter is somewhat outdated and will likely be condensed in a future update. Feel free to scan the material for ideas and useful examples, but technical reference sites (e.g. cppreference (https://en.cppreference.com/w/cpp/string/basic_string)) should be preferred for the most up-to-date information.

Motivation for a string class

In a previous lesson, we covered C-style strings (https://www.learncpp.com/cpp-tutorial/66-c-style-strings/), which uses char arrays to store a string of characters. If you've tried to do anything with C-style strings, you'll very quickly come to the conclusion that they are a pain to work with, easy to mess up, and hard to debug.

C-style strings have many shortcomings, primarily revolving around the fact that you have to do all the memory management yourself. For example, if you want to assign the string "hello!" into a buffer, you have to first dynamically allocate a buffer of the correct length:

```
char* strHello { new char[7] };
```

Don't forget to account for an extra character for the null terminator!

Then you have to actually copy the value in:

```
strcpy(strHello, "hello!");
```

Hopefully you made your buffer large enough so there's no buffer overflow!

And of course, because the string is dynamically allocated, you have to remember to deallocate it properly when you're done with it:

```
delete[] strHello;
```

Don't forget to use array delete instead of normal delete!

Furthermore, many of the intuitive operators that C provides to work with numbers, such as assignment and comparisons, simply don't work with C-style strings. Sometimes these will appear to work but actually produce incorrect results -- for example, comparing two C-style strings using == will actually do a pointer comparison, not a string comparison. Assigning one C-style string to another using operator= will appear to work at first, but is actually doing a pointer copy (shallow copy), which is not generally what you want. These kinds of things can lead to program crashes that are very hard to find and debug!

The bottom line is that working with C-style strings requires remembering a lot of nit-picky rules about what is safe/unsafe, memorizing a bunch of functions that have funny names like strcat() and strcmp() instead of using intuitive operators, and doing lots of manual memory management.

Fortunately, C++ and the standard library provide a much better way to deal with strings: the std::string and std::wstring classes. By making use of C++ concepts such as constructors, destructors, and operator overloading, std::string allows you to create and manipulate strings in an intuitive and safe manner! No more memory management, no more weird function names, and a much reduced potential for disaster.

Sign me up!

String overview

All string functionality in the standard library lives in the header file. To use it, simply include the string header:

```
#include <string>
```

There are actually 3 different string classes in the string header. The first is a templated base class named basic_string<>:

```
namespace std
{
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT> >
        class basic_string;
}
```

You won't be working with this class directly, so don't worry about what traits or an Allocator is for the time being. The default values will suffice in almost every imaginable case.

There are two flavors of basic_string<> provided by the standard library:

```
namespace std
{
    typedef basic_string<char> string;
    typedef basic_string<wchar_t> wstring;
}
```

These are the two classes that you will actually use. std::string is used for standard ascii and utf-8 strings. std::wstring is used for wide-character/unicode (utf-16) strings. There is no built-in class for utf-32 strings (though you should be able to extend your own from basic_string<> if you need one).

Although you will directly use std::string and std::wstring, all of the string functionality is implemented in the basic_string<> class. String and wstring are able to access that functionality directly by virtue of being templated. Consequently, all of the functions presented will work for both string and wstring. However, because basic_string is a templated class, it also means the compiler will produce horrible looking template errors when you do something syntactically incorrect with a string or wstring. Don't be intimidated by these errors; they look far worse than they are!

Here's a list of all the functions in the string class. Most of these functions have multiple flavors to handle different types of inputs, which we will cover in more depth in the next lessons.

Function	Effect
	Creation and destruction
(constructor)	Create or copy a string
(destructor)	Destroy a string
	Size and capacity
capacity()	Returns the number of characters that can be held without reallocation
empty()	Returns a boolean indicating whether the string is empty
length(), size()	Returns the number of characters in string
max_size()	Returns the maximum string size that can be allocated
reserve()	Expand or shrink the capacity of the string
	Element access
[], at()	Accesses the character at a particular index
	Modification
=, assign()	Assigns a new value to the string
+=, append(), push_back()	Concatenates characters to end of the string
insert()	Inserts characters at an arbitrary index in string
clear()	Delete all characters in the string
erase()	Erase characters at an arbitrary index in string
replace()	Replace characters at an arbitrary index with other characters
resize()	Expand or shrink the string (truncates or adds characters at end of string)
swap()	Swaps the value of two strings
	Input and Output
>>, getline()	Reads values from the input stream into the string
<<	Writes string value to the output stream
c_str()	Returns the contents of the string as a NULL-terminated C-style string
copy()	Copies contents (not NULL-terminated) to a character array
data()	Same as c_str(). The non-const overload allows writing to the returned string.
	String comparison
==, !=	Compares whether two strings are equal/unequal (returns bool)
<, <=, > >=	Compares whether two strings are less than / greater than each other (returns bool)
compare()	Compares whether two strings are equal/unequal (returns -1, 0, or 1)
	Substrings and concatenation
+	Concatenates two strings
substr()	Returns a substring
	Searching
find()	Find index of first character/substring
find_first_of()	Find index of first character from a set of characters
find_first_not_of()	Find index of first character not from a set of characters
find_last_of()	Find index of last character from a set of characters
find_last_not_of()	Find index of last character not from a set of characters
rfind()	Find index of last character/substring
	Iterator and allocator support
begin(), end()	Forward-direction iterator support for beginning/end of string
get_allocator()	Returns the allocator
rbegin(), rend()	Reverse-direction iterator support for beginning/end of string

While the standard library string classes provide a lot of functionality, there are a few notable omissions:

- Constructors for creating strings from numbers
- Capitalization / upper case / lower case functions
- Case-insensitive comparisons
- Tokenization / splitting string into array
- Easy functions for getting the left or right hand portion of string
- Whitespace trimming
- Formatting a string sprintf style
- Conversion from utf-8 to utf-16 or vice-versa

For most of these, you will have to either write your own functions, or convert your string to a C-style string (using c_str()) and use the C functions that offer this functionality.

In the next lessons, we will look at the various functions of the string class in more depth. Although we will use string for our examples, everything is equally applicable to wstring.

Next lesson
22.2 std::string construction and destruction

Back to table of contents

Previous lesson
21.4 STL algorithms overview

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

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

106 COMMENTS

Newest