6.6a — An introduction to std::string_view

BY NASCARDRIVER ON NOVEMBER 2ND, 2019 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the previous lesson, we talked about C-style strings, and the dangers of using them. C-style strings are fast, but they're not as easy to use and as safe as `std::string`.

But `std::string` (which we covered in lesson **S.4.4b -- An introduction to std::string**), has some of its own downsides, particularly when it comes to const strings.

Consider the following example:

```
#include <iostream>
#include <string>

int main()
{
  char text[]{ "hello" };
  std::string str{ text };
  std::string more{ str };

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

As expected, this prints

```
hello hello hello
```

Internally, `main` copies the string "hello" 3 times, resulting in 4 copies. First, there is the string literal "hello", which is known at compile-time and stored in the binary. One copy is created when we create the `char[]`. The following two `std::string` objects create one copy of the string each. Because `std::string` is designed to be modifiable, each `std::string` must contain its own copy of the string, so that a given `std::string` can be modified without affecting any other `std::string` object.

This holds true for const `std::string`, even though they can't be modified.

Introducing std::string_view

Consider a window in your house, looking at a car sitting on the street. You can look through the window and see the car, but you can't touch or move the car. Your window just provides a view to the car, which is a completely separate object.

C++17 introduces another way of using strings, `std::string_view`, which lives in the <string_view> header.

Unlike `std::string`, which keeps its own copy of the string, `std::string_view` provides a *view* of a string that is defined elsewhere.

We can re-write the above code to use `std::string_view` by replacing every `std::string` with `std::string_view`.

```
#include <iostream>
#include <string_view>

int main()
```

```
{
  std::string_view text{ "hello" }; // view the text "hello", which is stored in the binary
  std::string_view str{ text }; // view of the same "hello"
  std::string_view more{ str }; // view of the same "hello"

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

The output is the same, but no more copies of the string "hello" are created. When we copy a `std::string_view`, the new `std::string_view` observes the same string as the copied-from `std::string_view` is observing. `std::string_view` is not only fast, but has many of the functions that we know from `std::string`.

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "Trains are fast!" };

  std::cout << str.length() << '\n'; // 16
  std::cout << str.substr(0, str.find(' ')) << '\n'; // Trains
  std::cout << (str == "Trains are fast!") << '\n'; // 1

  // Since C++20
  std::cout << str.starts_with("Boats") << '\n'; // 0
  std::cout << str.ends_with("fast!") << '\n'; // 1

  std::cout << str << '\n'; // Trains are fast!

  return 0;
}
```

Because `std::string_view` doesn't create a copy of the string, if we change the viewed string, the changes are reflected in the `std::string_view`.

```
#include <iostream>
#include <string_view>

int main()
{
  char arr[]{ "Gold" };
  std::string_view str{ arr };

  std::cout << str << '\n'; // Gold

  // Change 'd' to 'f' in arr
  arr[3] = 'f';

  std::cout << str << '\n'; // Golf

  return 0;
}
```

We modified `arr`, but `str` appears to be changing as well. That's because `arr` and `str` share their string. When you use a `std::string_view`, it's best to avoid modifications to the underlying string for the remainder of the `std::string_view`'s life to prevent confusion and errors.

Best practice

Use `std::string_view` instead of C-style strings.

Prefer `std::string_view` over `std::string` for read-only strings, unless you already have a `std::string`.

View modification functions

Back to our window analogy, consider a window with curtains. We can close either the left or right curtain to reduce what we can see. We don't change what's outside, we just reduce the visible area.

Similarly, `std::string_view` contains functions that let us manipulate the *view* of the string. This allows us to change the view without modifying the viewed string.

The functions for this are `remove_prefix`, which removes characters from the left side of the view, and `remove_suffix`, which removes characters from the right side of the view.

```
#include <cstring> // For std::strlen
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "Peach" };

  std::cout << str << '\n';

  // Ignore the first characters.
  str.remove_prefix(1);

  std::cout << str << '\n';

  // Ignore the last 2 characters.
  str.remove_suffix(2);

  std::cout << str << '\n';

  return 0;
}
```

This program produces the following output:

```
Peach
each
ea
```

Unlike real curtains, a `std::string_view` cannot be opened back up. Once you change the visible area, you can't go back (There are tricks which we won't go into).

std::string_view works with non-null-terminated strings

Unlike C-style strings and `std::string`, `std::string_view` doesn't use null terminators to mark the end of the string. Rather, it knows where the string ends because it keeps track of its length.

```
#include <iostream>
#include <iterator> // For std::size
#include <string_view>

int main()
```

```
{
  // No null-terminator.
  char vowels[]{ 'a', 'e', 'i', 'o', 'u' };

  // vowels isn't null-terminated. We need to pass the length manually.
  // Because vowels is an array, we can use std::size to get its length.
  std::string_view str{ vowels, std::size(vowels) };

  std::cout << str << '\n'; // This is safe. std::cout knows how to print std::string_views.

  return 0;
}
```

This program prints:

```
aeiou
```

Ownership issues

Being only a view, a `std::string_view`'s lifetime is independent of that of the string it is viewing. If the viewed string goes out of scope, `std::string_view` has nothing to observe and accessing it causes undefined behavior.

```
#include <iostream>
#include <string>
#include <string_view>

std::string_view askForName()
{
  std::cout << "What's your name?\n";

  // Use a std::string, because std::cin needs to modify it.
  std::string str{};
  std::cin >> str;

  // We're switching to std::string_view for demonstrative purposes only.
  // If you already have a std::string, there's no reason to switch to
  // a std::string_view.
  std::string_view view{ str };

  std::cout << "Hello " << view << '\n';

  return view;
} // str dies, and so does the string that str created.

int main()
{
  std::string_view view{ askForName() };

  // view is observing a string that already died.
  std::cout << "Your name is " << view << '\n'; // Undefined behavior

  return 0;
}
```

```
What's your name?
nascardriver
Hello nascardriver
Your name is ◆P@◆P@
```

When we created `str` and filled it with `std::cin`, it created its internal string in dynamic memory. When `str` goes out of scope at the end of `askForName`, the internal string dies along with `str`. The `std::string_view` doesn't know that the string no longer exists and allows us to access it. Accessing the released string through `view` in `main` causes undefined behavior, which on the author's machine produced weird characters.

The same can happen when we create a `std::string_view` from a `std::string` and modify the `std::string`. Modifying a `std::string` can cause its internal string to die and be replaced with a new one in a different place. The `std::string_view` will still look at where the old string was, but it's not there anymore.

Warning

Make sure that the underlying string viewed with a `std::string_view` does not go out of scope and isn't modified while using the std::string_view.

Converting a `std::string_view` to a `std::string`

An std::string_view will not implicitly convert to a `std::string`, but can be explicitly converted:

```
#include <iostream>
#include <string>
#include <string_view>

void print(std::string s)
{
  std::cout << s << '\n';
}

int main()
{
  std::string_view sv{ "balloon" };

  sv.remove_suffix(3);

  // print(sv); // compile error: won't implicitly convert

  std::string str{ sv }; // explicit conversion

  print(str); // okay

  print(static_cast<std::string>(sv)); // okay

  return 0;
}
```

This prints:

```
ball
ball
```

Converting a `std::string_view` to a C-style string

Some old functions (such as the old strlen function) still expect C-style strings. To convert a `std::string_view` to a C-style string, we can do so by first converting to a `std::string`:

```
#include <cstring>
```

```
#include <iostream>
#include <string>
#include <string_view>

int main()
{
  std::string_view sv{ "balloon" };

  sv.remove_suffix(3);

  // Create a std::string from the std::string_view
  std::string str{ sv };

  // Get the null-terminated C-style string.
  auto szNullTerminated{ str.c_str() };

  // Pass the null-terminated string to the function that we want to use.
  std::cout << str << " has " << std::strlen(szNullTerminated) << " letter(s)\n";

  return 0;
}
```

This prints:

```
ball has 4 letter(s)
```

However, creating a `std::string` every time we want to pass a `std::string_view` as a C-style string is expensive, so this should be avoided if possible.

Opening the window (kinda) via the data() function

The string being viewed by a `std::string_view` can be accessed by using the `data()` function, which returns a C-style string. This provides fast access to the string being viewed (as a C-string). But it should also only be used if the `std::string_view`'s view hasn't been modified (e.g. by `remove_prefix` or `remove_suffix`) and the string being viewed is null-terminated.

In the following example, `std::strlen` doesn't know what a `std::string_view` is, so we need to pass it `str.data()`:

```
#include <cstring> // For std::strlen
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "balloon" };

  std::cout << str << '\n';

  // We use std::strlen because it's simple, this could be any other function
  // that needs a null-terminated string.
  // It's okay to use data() because we haven't modified the view, and the
  // string is null-terminated.
  std::cout << std::strlen(str.data()) << '\n';

  return 0;
}
```

```
balloon
7
```

When a `std::string_view` has been modified, `data()` doesn't always do what we'd like it to. The following example demonstrates what happens when we access `data()` after modifying the view:

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

int main()
{
  std::string_view str{ "balloon" };

  // Remove the "b"
  str.remove_prefix(1);
  // remove the "oon"
  str.remove_suffix(3);
  // Remember that the above doesn't modify the string, it only changes
  // the region that str is observing.

  std::cout << str << " has " << std::strlen(str.data()) << " letter(s)\n";
  std::cout << "str.data() is " << str.data() << '\n';
  std::cout << "str is " << str << '\n';

  return 0;
}
```

```
all has 6 letter(s)
str.data() is alloon
str is all
```

Clearly this isn't what we'd intended, and is a consequence of trying to access the data() of a `std::string_view` that has been modified. The length information about the string is lost when we access `data()`. `std::strlen` and `std::cout` keep reading characters from the underlying string until they find the null-terminator, which is at the end of "balloon".

Warning

Only use `std::string_view::data()` if the `std::string_view`'s view hasn't been modified and the string being viewed is null-terminated. Using `std::string_view::data()` of a non-null-terminated string can cause undefined behavior.

Incomplete implementation

Being a relatively recent feature, `std::string_view` isn't implemented as well as it could be.

```
std::string s{ "hello" };
std::string_view v{ "world" };

// Doesn't work
std::cout << (s + v) << '\n';
std::cout << (v + s) << '\n';

// Potentially unsafe, or not what we want, because we're treating
```

```
9  // the std::string_view as a C-style string.
10 std::cout << (s + v.data()) << '\n';
11 std::cout << (v.data() + s) << '\n';
12
13 // Ok, but ugly and wasteful because we have to construct a new std::string.
14 std::cout << (s + std::string{ v }) << '\n';
15 std::cout << (std::string{ v } + s) << '\n';
16 std::cout << (s + static_cast<std::string>(v)) << '\n';
17 std::cout << (static_cast<std::string>(v) + s) << '\n';
```

There's no reason why line 5 and 6 shouldn't work. They will probably be supported in a future C++ version.

6.7 -- Introduction to pointers

Index

6.6 -- C-style strings

C++ TUTORIAL | PRINT THIS POST

26 comments to 6.6a — An introduction to std::string_view

Jordy
February 10, 2020 at 3:42 pm · Reply

std::string_view's function seems to overlap a lot with what constant references are used for. Why is it preferred?

kitabski
February 9, 2020 at 8:56 pm · Reply

Good day everybody!!!
Wanted to clarify one thing.
The following code works just fine even though i don't include string or string_view

```
#include <iostream>
//#include <string>
//#include <string_view>

int main()
{
    using namespace std;
    string_view one {"Hello"};
    string_view two {one};
    cout << one << ' ' << two << endl;
    return 0;
}
```

kavin
January 21, 2020 at 10:17 am · Reply

Hi, Under Opening the window (kinda) via the data() function:
how do we know std::string_view str{ "balloon" }; is null-terninated string?

From what i understood is it because all strings(refers to group of characters here) default to null-terminated strings? If the characters are separated by "," they naturally default to non-null-terminated and c-style, which can only be printed to screen by converting them to std::string_view like in vowels example ? Is it correct ?

nascardriver
January 22, 2020 at 3:56 am · Reply

"balloon" is a string literal, string literals are zero-terminated. If you have an array of characters, then that array isn't zero-terminated (unless you added a terminator).

kavin
January 22, 2020 at 11:52 pm · Reply

Thank you, we add an terminator to array of characters like this right??

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

int main()
{
    char vowels[]{'a', 'e', 'i', 'o', 'u', '\0'};
    const int length{static_cast<int>(std::size(vowels))};
        std::cout << "length is " << length << '\n'; //prints 6 including \0

    for (int index{0}; index < length; ++index)
        std::cout << vowels[index] << " ";  // prints a e i o u

    std::cout << '\n';

    return 0;
}
```

Here it prints 0 after aeiou but its not shown in output right ?

nascardriver
January 23, 2020 at 1:18 am · Reply

Right. When you print a 0-valued byte, you see nothing. You could print the array without a loop, because now `std::cout` can know where the string ends.

```
1 std::cout << vowels << '\n';
```

chai
December 27, 2019 at 12:17 pm · Reply

[code]
char arr[]{ "Gold" };
char name[] = "Jason";
[code]

I learned in lesson 6.8 that I haven't got '\0' at the end of "Gold" but got one with "Jason". Is the assignment initialisation better?

nascardriver
December 28, 2019 at 2:16 am · Reply

Both arrays are zero-terminated

```
1  #include <iostream>
2  #include <iterator>
3
4  int main()
5  {
6      char arr[]{ "Gold" };
7      char name[] = "Gold";
8
9      std::cout << std::size(arr) << '\n' << std::size(name) << '\n';
10
11     return 0;
12 }
```

Output
5
5

Lesson 6.6 says so. I couldn't find anything stating the opposite in lesson 6.8. If there's misinformation in lesson 6.8, please point it out as to not confuse other readers.

BakaTensi
November 26, 2019 at 12:10 am · Reply

So basically, std::string_view is kinda incomplete, and should only be used when we need constant strings at this point if I understand correctly?

nascardriver
November 26, 2019 at 1:10 am · Reply

Correct. If you're not going to modify a string and you can initialize it, use a `std::string_view`.

Vuk
November 20, 2019 at 11:56 am · Reply

Hi, I think there is a missing parenthesis in the third code example for the substr call.

nascardriver
November 21, 2019 at 2:32 am · Reply

It was missing indeed. Thanks for pointing out the omission!

Wallace
November 15, 2019 at 2:29 pm · Reply

Minor typo: In "When you use a std::string_view, it's best to avoid modifications the to underlying string," "the" and "to" seem swapped.

Fan
November 13, 2019 at 3:16 am · Reply

OK. I saw that the first example has been modified to use char[] instead of const char *, but then should there also be a copy of the string allocated on the stack frame of main()?

nascardriver
November 13, 2019 at 4:09 am · Reply

Right, I missed that when updating the lesson to use `char[]`, thanks for pointing it out!.

Fan
November 13, 2019 at 4:51 am · Reply

Then it comes up to 4 copies of the string?

nascardriver
November 13, 2019 at 4:53 am · Reply

I don't think so. When you have 1 paper and copy it, you have 2 papers but only 1 copy, right?
The string is there 4 times, but only 3 of the strings are copies. That's how I look at it, I don't know if it's correct.

Alex
November 14, 2019 at 7:23 pm · Reply

In C++, the term "copies" is often used to mean the "number of" something. So if you have a string, and you copy it, you now have two copies of that string.

I think it's more correct to say that the first program has 4 copies of the string -- but function main only creates 3 of those copies. Perhaps the text should be amended to say, "main creates 3 copies" rather than "this program creates 3 copies"? That way we avoid the ambiguity about whether the string in read-only memory counts as a "copy" or not.

nascardriver
November 15, 2019 at 12:31 am · Reply

Updated to "main copies the string “hello” 3 times, resulting in 4 copies"

Omar Abdelazeem
November 10, 2019 at 4:08 am · Reply

when I'm trying to compile this code , the compiler complain that
'string_view': is not a member of 'std'

[/

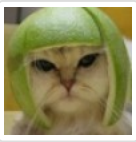
```
#include <iostream>
#include <string_view>

int main()
{
   std::string_view text{ "hello" }; // view the text "hello", which is stored in the binary
   std::string_view str{ text }; // view of the same "hello"
   std::string_view more{ str }; // view of the same "hello"

   std::cout << text << ' ' << str << ' ' << more << '\n';

   return 0;
}
```

]

nascardriver
November 10, 2019 at 4:16 am · Reply

Hi!

`std::string_view` was added in C++17. Make sure you enabled C++17 or higher in your project settings.



Alex
November 13, 2019 at 1:33 pm · Reply

For future readers, setting C++17 (or higher) is now covered in lesson 0.12.

Omar Abdelazeem
November 10, 2019 at 4:47 am · Reply

Thank you very much

ErwanDL
November 7, 2019 at 2:23 am · Reply

Hey nascardriver,

From the start of this article, you introduce "const char*" to define strings but I don't think we have already seen what it is and what it does earlier in the tutorial. Is it like using "std::string" ?

Another thing is I have played around a bit with std::string_view and there are some pretty bizarre behaviours, like in the following code :

```
1  int main()
2  {
3      std::string str{"Hello"};
4      std::string_view view{str};
5
6      std::cout << str << " " << view << "\n";
7
8      str = "Hi";
9
10     std::cout << str << " " << view << "\n";
11
12     return 0;
13 }
```

This prints :

Hello Hello
Hi Hilo

What is happening here ? Maybe this could be explained in the article as well ?

Cheers

ErwanDL
November 7, 2019 at 3:34 am · Reply

Update : I've advanced to lesson 6.8b and the syntax "const char*" is explained there (C-style string symbolic constants). Maybe this lesson on std::string_view should be moved further down the chapter ?

nascardriver
November 7, 2019 at 3:40 am · Reply

> you introduce "const char*" to define strings but I don't think we have already seen what it is
It's a `const char[]` with a different syntax, I though it was a part of the previous lesson, but it isn't. I changed every `const char*` to `char[]` or "C-style string". Thanks for letting me know!

> like in the following code
I wanted to include an example that shows that `std::string_view` reflects the changes made to its underlying string, but I'd have to move this lesson further back, which I don't want to do, because it would motivate the use of C-style strings.
Your code invokes undefined behavior (I added a paragraph to the lesson).
When you assign "Hi" to `str` in line 8, `str`'s old string "Hello" can be invalidated. In your case, since "Hi" is shorter than "Hello", the `std::string` re-used the memory it was using the store "Hello". That memory is now "Hi\0lo" (\0 is a null-terminator).
If you assigned something longer, the old string could die completely.
`view` still looks at where the old string was, but `view`'s size isn't changed (The curtains are where they were before). Since `std::string_view` doesn't use null-terminators, the \0 is ignored and "Hilo" is printed.