5.11 — std::string_view (part 2)

ALEX JANUARY 9, 2024

In prior lessons, we introduced two string types: `std::string` (5.9 -- Introduction to std::string (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/)) and `std::string_view` (5.10 -- Introduction to std::string_view (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring_view/)).

Because `std::string_view` is our first encounter with a view type, we're going to spend some additional time discussing it further. We will focus on how to use `std::string_view` safely, and provide some examples illustrating how it can be used incorrectly. We'll conclude with some guidelines on when to use `std::string` vs `std::string_view`.

An introduction to owners and viewers

Let's sidebar into an analogy for a moment. Say you've decided that you're going to paint a picture of a bicycle. But you don't have a bicycle! What are you to do?

Well, you could go to the local cycle shop and buy one. You would own that bike. This has some benefits: you now have a bike that you can ride. You can guarantee the bike will always be available when you want it. You can decorate it, or move it. There are also some downsides to this choice. Bicycles are expensive. And if you buy one, you are now responsible for it. You have to periodically maintain it. And when you eventually decide you don't want it any more, you have to properly dispose of it.

Ownership can be expensive. As an owner, it is your responsibility to acquire, manage, and properly dispose of the objects you own.

On your way out of the house, you glance out your window front. You notice that your neighbor has parked their bike across from your window. You could just paint a picture of your neighbor's bike (as seen from your window) instead. There are lots of benefits to this choice. You save the expense of having to go acquire your own bike. You don't have to maintain it. Nor are you responsible for disposing of it. When you are done viewing, you can just shut your curtains and move on with your life. This ends your view of the object, but the object itself is not affected by this. There are also some potential downsides to this choice. You can't paint or customize your neighbors bike. And while you are viewing the bike, your neighbor may decide to change the way the bike looks, or move it out of your view altogether. You may end up with a view of something unexpected instead.

Viewing is inexpensive. As a viewer, you have no responsibility for the objects you are viewing, but you also have no control over those objects.

`std::string` is an owner

You might be wondering why `std::string` makes an expensive copy of its initializer. When an object is instantiated, memory is allocated for that object to store whatever data it needs to use throughout its lifetime. This memory is reserved for the object, and guaranteed to exist for as long as the object does. It is a safe space. `std::string` (and most other objects) copy the initialization value they are given into this memory so that they can have their own independent value to access and manipulate later. Once the initialization value has been copied, the object is no longer reliant on the initializer in any way.

And that's a good thing, because the initializer generally can't be trusted after initialization is complete. If you imagine the initialization process as a function call that initializes the object, who is passing in the initializer? The caller. When initialization is done, control returns back to the

caller. At this point, the initialization statement is complete, and one of two things will typically happen:

- If the initializer was a temporary value or object, that temporary will be destroyed immediately.
- If the initializer was a variable, the caller still has access to that object. The caller can then do whatever they want with the object, including modify or destroy it.

> **Key insight**
>
> An initialized object has no control over what happens to the initialization value after initialization is finished.

Because `std::string` makes its own copy of the value, it doesn't have to worry about what happens after initialization is finished. The initializer can be destroyed, or modified, and it doesn't affect the `std::string`. The downside is that this independence comes with the cost of an expensive copy.

In the context of our analogy, `std::string` is an owner. It has its own data that it manages. And when it is destroyed, it cleans up after itself.

We don't always need a copy

Let's revisit this example from the prior lesson:

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

void printString(std::string str) // str makes a copy of its initializer
{
    std::cout << str << '\n';
}

int main()
{
    std::string s{ "Hello, world!" };
    printString(s);

    return 0;
}
```

When `printString(s)` is called, `str` makes an expensive copy of `s`. The function prints the copied string and then destroys it.

Note that `s` is already holding the string we want to print. Could we just use the string that `s` is holding instead of making a copy? The answer is possibly -- there are three criteria we need to assess:

- Could `s` be destroyed while `str` is still using it? No, `str` dies at the end of the function, and `s` exists in the scope of the caller and can't be destroyed before the function returns.

- Could `s` be modified while `str` is still using it? No, `str` dies at the end of the function, and the caller has no opportunity to modify the `s` before the function returns.
- Does `str` modify the string in some way that the caller would not expect? No, the function does not modify the string at all.

Since all three of these criteria are false, there is no risk in using the string that `s` is holding instead of making a copy. And since string copies are expensive, why pay for one that we don't need?

`std::string_view` is a viewer

`std::string_view` takes a different approach to initialization. Instead of making an expensive copy of the initialization string, `std::string_view` creates an inexpensive view of the initialization string. The `std::string_view` can then be used whenever access to the string is required.

In the context of our analogy, `std::string_view` is a viewer. It views an object that already exists elsewhere, and cannot modify that object. When the view is destroyed, the object being viewed is not affected.

It is important to note that a `std::string_view` remains dependent on the initializer through its lifetime. If the string being viewed is modified or destroyed while the view is still being used, unexpected or undefined behavior will result.

Whenever we use a view, it is up to us to ensure these possibilities do not occur.

> **Warning**
>
> A view is dependent on the object being viewed. If the object being viewed is modified or destroyed while the view is still being used, unexpected or undefined behavior will result.

A `std::string_view` that is viewing a string that has been destroyed is sometimes called a **dangling** view.

`std::string_view` is best used as a read-only function parameter

The best use for `std::string_view` is as a read-only function parameter. This allows us to pass in a C-style string, `std::string`, or `std::string_view` argument without making a copy, as the `std::string_view` will create a view to the argument.

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

void printSV(std::string_view str) // now a std::string_view, creates a view of the argument
{
    std::cout << str << '\n';
}

int main()
{
    printSV("Hello, world!"); // call with C-style string literal

    std::string s2{ "Hello, world!" };
    printSV(s2); // call with std::string

    std::string_view s3 { s2 };
    printSV(s3); // call with std::string_view

    return 0;
}
```

Because the `str` function parameter is created, initialized, used, and destroyed before control returns to the caller, there is no risk that the string being viewed (the function argument) will be modified or destroyed before our `str` parameter.

Should I prefer `std::string_view` or `const std::string&` function parameters?" Advanced

Prefer `std::string_view` in most cases. We cover this topic further in lesson 12.6 -- Pass by const lvalue reference (https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#stringparameter).

Improperly using `std::string_view`

Let's take a look at a few cases where misusing `std::string_view` gets us into trouble.

Here's our first example:

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

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

    { // create a nested block
        std::string s{ "Hello, world!" }; // create a std::string local to this nested block
        sv = s; // sv is now viewing s
    } // s is destroyed here, so sv is now viewing an invalid string

    std::cout << sv << '\n'; // undefined behavior

    return 0;
}
```

In this example, we're creating `std::string s` inside a nested block (don't worry about what a nested block is yet). Then we set `sv` to view `s`. `s` is then destroyed at the end of the nested block. `sv` doesn't know that `s` has been destroyed. When we then use `sv`, we are accessing an invalid object, and undefined behavior results.

Here's another variant of the same issue, where we initialize a `std::string_view` with the `std::string` return value of a function:

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

std::string getName()
{
    std::string s { "Alex" };
    return s;
}

int main()
{
  std::string_view name { getName() }; // name initialized with return value of function
  std::cout << name << '\n'; // undefined behavior

  return 0;
}
```

This behaves similarly to the prior example. The `getName()` function is returning a `std::string` containing the string "Alex". Return values are temporary objects that are destroyed at the end of the full expression containing the function call. We must either use this return value immediately, or copy it to use later.

But `std::string_view` doesn't make copies. Instead, it creates a view to the temporary return value, which is then destroyed. That leaves our `std::string_view` dangling (viewing an invalid object), and printing the view results in undefined behavior.

The following is a less-obvious variant of the above:

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

int main()
{
    using namespace std::string_literals;
    std::string_view name { "Alex"s }; // "Alex"s creates a temporary std::string
    std::cout << name << '\n'; // undefined behavior

    return 0;
}
```

A `std::string` literal (created via the `s` literal suffix) creates a temporary `std::string` object. So in this case, `"Alex"s` creates a temporary `std::string`, which we then use as the initializer for `name`. At this point, `name` is viewing the temporary `std::string`. Then the temporary `std::string` is destroyed, leaving `name` dangling. We get undefined behavior when we then use `name`.

Warning

Do not initialize a `std::string_view` with a `std::string` literal, as this will leave the `std::string_view` dangling.

It is okay to initialize a `std::string_view` with a C-style string literal or a `std::string_view` literal. It's also okay to initialize a `std::string_view` with a C-style string object, a `std::string` object, or a `std::string_view` object, as long as that string object outlives the view.

We can also get undefined behavior when the underlying string is modified:

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

int main()
{
    std::string s { "Hello, world!" };
    std::string_view sv { s }; // sv is now viewing s

    s = "Hello, universe!";    // modifies s, which invalidates sv (s is still valid)
    std::cout << sv << '\n';   // undefined behavior

    return 0;
}
```

In this example, `sv` is again set to view `s`. `s` is then modified. When a `std::string` is modified, all views into that `std::string` are **invalidated**, meaning those views are now invalid. Using an invalidated view will result in undefined behavior. Thus, when we print `sv`, undefined behavior results.

> **Key insight**
>
> Modifying a `std::string` invalidates all views into that `std::string`.

Revalidating an invalid `std::string_view`

Invalidated objects can often be revalidated (made valid again) by setting them back to a known good state. For an invalidated `std::string_view`, we can do this by assigning the invalidated `std::string_view` object a valid string to view.

Here's the same example as prior, but we'll revalidate `sv`:

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

int main()
{
    std::string s { "Hello, world!" };
    std::string_view sv { s }; // sv is now viewing s

    s = "Hello, universe!";    // modifies s, which invalidates sv (s is still valid)
    std::cout << sv << '\n';   // undefined behavior

    sv = s;                    // revalidate sv: sv is now viewing s again
    std::cout << sv << '\n';   // prints "Hello, universe!"

    return 0;
}
```

After `sv` is invalidated by the modification of `s`, we revalidate `sv` via the statement `sv = s`, which causes `sv` to become a valid view of `s` again. When we print `sv` the second time, it prints "Hello, universe!".

Be careful returning a `std::string_view`

`std::string_view` can be used as the return value of a function. However, this is often dangerous.

Because local variables are destroyed at the end of the function, returning a `std::string_view` to a local variable will result in the returned `std::string_view` being invalid, and further use of that `std::string_view` will result in undefined behavior. For example:

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

std::string_view getBoolName(bool b)
{
    std::string t { "true" };  // local variable
    std::string f { "false" }; // local variable

    if (b)
        return t;  // return a std::string_view viewing t

    return f; // return a std::string_view viewing f
} // t and f are destroyed at the end of the function

int main()
{
    std::cout << getBoolName(true) << ' ' << getBoolName(false) << '\n'; // undefined behavior

    return 0;
}
```

In the above example, when `getBoolName(true)` is called, the function returns a `std::string_view` that is viewing `t`. However, `t` is destroyed at the end of the function. This means the returned `std::string_view` is viewing an object that has been destroyed. So when the returned `std::string_view` is printed, undefined behavior results.

Your compiler may or may not warn you about such cases.

There are two main cases where a `std::string_view` can be returned safely. First, because C-style string literals exist for the entire program, it's fine to return C-style string literals from a function that has a return type of `std::string_view`.

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

std::string_view getBoolName(bool b)
{
    if (b)
        return "true";  // return a std::string_view viewing "true"

    return "false"; // return a std::string_view viewing "false"
} // "true" and "false" are not destroyed at the end of the function

int main()
{
    std::cout << getBoolName(true) << ' ' << getBoolName(false) << '\n'; // ok

    return 0;
}
```

This prints:

```
true false
```

When `getBoolName(true)` is called, the function will return a `std::string_view` viewing the C-style string `"true"`. Because `"true"` exists for the entire program, there's no problem when we use the returned `std::string_view` to print `"true"` within `main()`.

Second, it is generally okay to return a function parameter of type `std::string_view`:

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

std::string_view firstAlphabetical(std::string_view s1, std::string_view s2)
{
    return s1 < s2 ? s1: s2; // uses operator?: (the conditional operator)
}

int main()
{
    std::string a { "World" };
    std::string b { "Hello" };

    std::cout << firstAlphabetical(a, b) << '\n'; // prints "Hello"

    return 0;
}
```

It may be less obvious why this is okay. First, note that arguments `a` and `b` exist in the scope of the caller. When the function is called, function

parameter `s1` is a view into `a`, and function parameter `s2` is a view into `b`. When the function returns either `s1` or `s2`, it is returning a view into `a` or `b` back to the caller. Since `a` and `b` still exist at this point, it's fine to use the returned `std::string_view` into `a` or `b`.

There is one important subtlety here. If the argument is a temporary object (that will be destroyed at the end of the full expression containing the function call), the `std::string_view` return value must be used in the same expression. After that point, the temporary is destroyed and the std::string_view is left dangling.

Warning

If an argument is a temporary that is destroyed at the end of the full expression containing the function call, the returned `std::string_view` must be used immediately, as it will be left dangling after the temporary is destroyed.

View modification functions

Consider a window in your house, looking at an electric 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.

Many windows have curtains, which allow us to modify our view. 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.

Because `std::string_view` is a view, it contains functions that let us modify our view by "closing the curtains". This does not modify the string being viewed in any way, just the view itself.

- The `remove_prefix()` member function removes characters from the left side of the view.
- The `remove_suffix()` member function removes characters from the right side of the view.

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

int main()
{
    std::string_view str{ "Peach" };
    std::cout << str << '\n';

    // Remove 1 character from the left side of the view
    str.remove_prefix(1);
    std::cout << str << '\n';

    // Remove 2 characters from the right side of the view
    str.remove_suffix(2);
    std::cout << str << '\n';

    str = "Peach"; // reset the view
    std::cout << str << '\n';

    return 0;
}
```

This program produces the following output:

```
Peach
each
ea
Peach
```

Unlike real curtains, once `remove_prefix()` and `remove_suffix()` have been called, the only way to reset the view is by reassigning the source string to it again.

`std::string_view` can view a substring

This brings up an important use of `std::string_view`. While `std::string_view` can be used to view an entire string without making a copy, they are also useful when we want to view a substring without making a copy. A **substring** is a contiguous sequence of characters within an existing string. For example, given the string "snowball", some substrings are "snow", "all", and "now". "owl" is not a substring of "snowball" because these characters do not appear contiguously in "snowball".

`std::string_view` may or may not be null-terminated

The ability to view just a substring of a larger string comes with one consequence of note: a `std::string_view` may or may not be null-terminated. Consider the string "snowball", which is null-terminated. If a `std::string_view` views the whole string, then it is viewing a null-terminated string. However, if `std::string_view` is only viewing the substring "now", then that substring is not null-terminated (the next

character is a 'b').

Key insight

A C-style string literal and a `std::string` are always null-terminated.
A `std::string_view` may or may not be null-terminated.

In almost all cases, this doesn't matter -- a `std::string_view` keeps track of the length of the string or substring it is viewing, so it doesn't need the null-terminator. Converting a `std::string_view` to a `std::string` will work regardless of whether or not the `std::string_view` is null-terminated.

Warning

Take care not to write any code that assumes a `std::string_view` is null terminated.

Tip

If you have a non-null-terminated `std::string_view` and you need a null-terminated string for some reason, convert the `std::string_view` into a `std::string`.

A quick guide on when to use `std::string` vs `std::string_view`

This guide is not meant to be comprehensive, but is intended to highlight the most common cases:

Use a `std::string` variable when:

- You need a string that you can modify.
- You need to store user-inputted text.
- You need to store the return value of a function that returns a `std::string`.

Use a `std::string_view` variable when:

- You need read-only access to part or all of a string that already exists elsewhere and will not be modified or destroyed before use of the `std::string_view` is complete.
- You need a symbolic constant for a C-style string.
- You need to continue viewing the return value of a function that returns a C-style string or a non-dangling `std::string_view`.

Use a `std::string` function parameter when:

- The function needs to modify the string passed in as an argument without affecting the caller. This is rare.
- You are using a language standard older than C++17.
- You meet the criteria of the pass-by-reference cases covered in lesson 12.5 -- Pass by lvalue reference.

Use a `std::string_view` function parameter when:

- The function needs a read-only string.

Use a `std::string` return type when:

- The return value is a `std::string` local variable.
- The return value is a function call or operator that returns a `std::string` by value.
- You meet the criteria of the return-by-reference cases covered in lesson 12.12 -- Return by reference and return by address.

Use a `std::string_view` return type when:

- The function returns a C-style string literal or local `std::string_view` that has been initialized with a C-style string literal.
- The function returns a `std::string_view` parameter.

Things to remember about `std::string`:

- Initializing and copying `std::string` is expensive, so avoid this as much as possible.
- Avoid passing `std::string` by value, as this makes a copy.
- If possible, avoid creating short-lived `std::string` objects.
- Modifying a `std::string` will invalidate any views to that string.

Things to remember about `std::string_view`:

- Because C-style string literals exist for the entire program, it is always okay to set a `std::string_view` to a C-style string literal.

- When a string is destroyed, all views to that string are invalidated.
- Using an invalidated view (other than using assignment to revalidate the view) will cause undefined behavior.
- A `std::string_view` may or may not be null-terminated.

Next lesson
5.x Chapter 5 summary and quiz

Back to table of contents

Previous lesson
5.10 Introduction to std::string_view