10.7 — An introduction to std::string_view

NASCARDRIVER AUGUST 25, 2021

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 lesson4.12 -- 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. The string “hello” is stored in the binary and is not allocated at run-time. `text` is only a view onto the string “hello”, so no copy has to be 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. This means that neither `str` nor `more` create any copies. They are views onto the existing string “hello”.

`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`.

```cpp
#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.

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

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

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

  // Ignore the first
character.
  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.

```cpp
#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. The string that a `std::string_view` is viewing has to have been created somewhere else. It might be a string literal that lives as long as the program does or it was created by a `std::string`, in which case the string lives until the `std::string` decides to destroy it or the `std::string` dies. `std::string_view` can't create any strings on its own, because it's just a view.

```cpp
#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@
```

In function `askForName()`, we create `str` and fill it with data from `std::cin`. Then we create `view`, which can view that string. At the end of the function, we return `view`, but the string it is viewing (`str`) is destroyed, so `view` is now pointing to deallocated memory. The function returns a dangling `std::string_view`.

Accessing the returned `std::string_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 then 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:

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

void print(const 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 }; // okay

  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`:

```cpp
#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.
  const char* 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()`:

```cpp
#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:

```cpp
#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
// the std::string_view as a C-style string.
std::cout << (s + v.data()) << '\n';
std::cout << (v.data() + s) << '\n';

// Ok, but ugly and wasteful because we have to construct a new
std::string.
std::cout << (s + std::string{ v }) << '\n';
std::cout << (std::string{ v } + s) << '\n';
std::cout << (s + static_cast<std::string>(v)) << '\n';
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.

Next lesson
10.8 Introduction to pointers

Back to table of contents

Previous lesson
10.6 C-style strings