4.18 — Introduction to std::string_view

NASCARDRIVER AUGUST 6, 2022

Consider the following program:

```
#include <iostream>

int main()
{
    int x { 5 };
    std::cout << x <<
'\n';

    return 0;
}
```

When the definition for `x` is executed, the initialization value `5` is copied into the memory allocated for `int x`. For fundamental types, initializing (or copying) a variable is fast.

Now consider this similar program:

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

int main()
{
    std::string s{ "Hello, world!"
};
    std::cout << s << '\n';

    return 0;
}
```

When `s` is initialized, the C-style string literal `"Hello, world!"` is copied into memory allocated for `std::string s`. Unlike fundamental types, initializing (or copying) a `std::string` is slow.

In the above program, all we do with `s` is print the value to the console, and then `s` is destroyed. We've essentially made a copy of "Hello, world!" just to print and then destroy that copy. That's inefficient.

We see something similar in this example:

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

void printString(std::string str)
{
    std::cout << str << '\n';
}

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

    return 0;
}
```

This example makes two copies of the C-style string "Hello, world!": one when we initialize `s` in `main()`, and another when we initialize parameter `str` in `printString()`. That's a lot of needless copying just to print a string!

---

std::string_view C++17

To address the issue with `std::string` being expensive to initialize (or copy), C++17 introduced `std::string_view` (which lives in the <string_view> header). `std::string_view` provides read-only access to an existing string (a C-style string literal, a `std::string`, or a char array) without making a copy.

The following example is identical to the prior one, except we've replaced `std::string` with `std::string_view`.

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

void printSV(std::string_view str) // now a std::string_view
{
    std::cout << str << '\n';
}

int main()
{
    std::string_view s{ "Hello, world!" }; // now a
std::string_view
    printSV(s);

    return 0;
}
```

This program produces the same output as the prior one, but no copies of the string "Hello, world!" are made.

When we initialize `std::string_view s` with C-style string literal `"Hello, world!"`, `s` provides read-only access to "Hello, world!" without making a copy of the string. When we pass `s` to `printSV()`, parameter `str` is initialized from `s`. This allows us to access "Hello, world!" through `str`, again without making a copy of the string.

> **Best practice**
>
> Prefer `std::string_view` over `std::string` when you need a read-only string, especially for function parameters.

---

constexpr std::string_view

Unlike `std::string`, `std::string_view` has full support for constexpr:

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

int main()
{
    constexpr std::string_view s{ "Hello, world!" };
    std::cout << s << '\n'; // s will be replaced with "Hello, world!" at compile-
time

    return 0;
}
```

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

A `std::string_view` can be created using a `std::string` initializer, and a `std::string` will implicitly convert to a `std::string_view`:

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

void printSV(std::string_view str)
{
    std::cout << str << '\n';
}

int main()
{
    std::string s{ "Hello, world" };
    std::string_view sv{ s }; // Initialize a std::string_view from a std::string
    std::cout << sv << '\n';

    printSV(s); // implicitly convert a std::string to std::string_view

    return 0;
}
```

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

Because `std::string` makes a copy of its initializer, C++ won't allow implicit conversion of a `std::string` from a `std::string_view`. However, we can explicitly create a `std::string` with a `std::string_view` initializer, or we can convert an existing `std::string_view` to a `std::string` using `static_cast`:

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

void printString(std::string str)
{
    std::cout << str << '\n';
}

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

  std::string str{ sv }; // okay, we can create std::string using std::string_view initializer

  // printString(sv);   // compile error: won't implicitly convert std::string_view to a std::string

  printString(static_cast<std::string>(sv)); // okay, we can explicitly cast a std::string_view to a std::string

  return 0;
}
```

Literals for `std::string_view`

Double-quoted string literals are C-style string literals by default. We can create string literals with type `std::string_view` by using a `sv` suffix after the double-quoted string literal.

```cpp
#include <iostream>
#include <string>      // for std::string
#include <string_view> // for std::string_view

int main()
{
    using namespace std::literals; // easiest way to access the s and sv
suffixes

    std::cout << "foo\n";   // no suffix is a C-style string literal
    std::cout << "goo\n"s;  // s suffix is a std::string literal
    std::cout << "moo\n"sv; // sv suffix is a std::string_view literal

    return 0;
};
```

Tip

The "sv" suffix lives in the namespace `std::literals::string_view_literals`. The easiest way to access the literal suffixes is via using directive `using namespace std::literals`. We discuss using directives in lesson 6.12 -- Using declarations and using directives (https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/). This is one of the exception cases where `using` an entire namespace is okay.

Do not return a std::string_view

Returning a std::string_view from a function is usually a bad idea. We'll explore why in lesson 11.7 -- std::string_view (part 2) (https://www.learncpp.com/cpp-tutorial/stdstring_view-part-2/). For now, avoid doing so.

Next lesson
4.x Chapter 4 summary and quiz

Back to table of contents

Previous lesson
4.17 Introduction to std::string

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

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

Notify me about replies: POST COMMENT

210 COMMENTS Newest