5.10 — Introduction to std::string_view

Consider the following program:

```
#include <iostream>

int main()
{
    int x { 5 }; // x makes a copy of its initializer
    std::cout << x << '\n';

    return 0;
}
```

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

Now consider this similar program:

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

int main()
{
    std::string s{ "Hello, world!" }; // s makes a copy of its initializer
    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 and 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:

```
#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!" }; // s makes a copy of its initializer
    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, a `std::string`, or another `std::string_view`) without making a copy. **Read-only** means that we can access and use the value being viewed, but we can not modify it.

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

```
#include <iostream>
#include <string_view> // C++17

// str provides read-only access to whatever argument is passed in
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.

`std::string_view` can be initialized with many different types of strings

One of the neat things about a `std::string_view` is how flexible it is. A `std::string_view` object can be initialized with a C-style string, a `std::string`, or another `std::string_view`:

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

int main()
{
    std::string_view s1 { "Hello, world!" }; // initialize with C-style string
literal
    std::cout << s1 << '\n';

    std::string s{ "Hello, world!" };
    std::string_view s2 { s };  // initialize with std::string
    std::cout << s2 << '\n';

    std::string_view s3 { s2 }; // initialize with std::string_view
    std::cout << s3 << '\n';

    return 0;
}
```

`std::string_view` parameters will accept many different types of string arguments

Both a C-style string and a `std::string` will implicitly convert to a `std::string_view`. Therefore, a `std::string_view` parameter will accept arguments of type C-style string, a `std::string`, or `std::string_view`:

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

void printSV(std::string_view str)
{
    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;
}
```

`std::string_view` will not implicitly convert to `std::string`

Because `std::string` makes a copy of its initializer (which is expensive), C++ won't allow implicit conversion of a `std::string_view` to a `std::string`. This is to prevent accidentally passing a `std::string_view` argument to a `std::string` parameter, and inadvertently making an expensive copy where such a copy may not be required.

However, if this is desired, we have two options:

1. Explicitly create a `std::string` with a `std::string_view` initializer (which is allowed, since this will rarely be done unintentionally)
2. Convert an existing `std::string_view` to a `std::string` using `static_cast`

The following example shows both options:

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

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

int main()
{
	std::string_view sv{ "Hello, world!" };

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

	std::string s{ sv }; // okay: we can create std::string using
std::string_view initializer
	printString(s);      // and call the function with the std::string

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

	return 0;
}
```

Assignment changes what the `std::string_view` is viewing

Assigning a new string to a `std::string_view` causes the `std::string_view` to view the new string. It does not modify the prior string being viewed in any way.

The following example illustrates this:

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

int main()
{
    std::string name { "Alex" };
    std::string_view sv { name }; // sv is now viewing name
    std::cout << sv << '\n'; // prints Alex

    sv = "John"; // sv is now viewing "John" (does not change name)
    std::cout << sv << '\n'; // prints John

    std::cout << name << '\n'; // prints Alex

    return 0;
}
```

In the above example, `sv = "John"` causes `sv` to now view the string `"John"`. It does not change the value held by `name` (which is still `"Alex"`).

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. The `sv` must be lower case.

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

int main()
{
    using namespace std::string_literals;      // access the s suffix
    using namespace std::string_view_literals; // access the sv suffix

    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;
}
```

Related content

We discuss this use of `using namespace` in lesson 5.9 -- Introduction to std::string. The same advice applies here.

It's fine to initialize a `std::string_view` object with a C-style string literal (you don't need to initialize it with a `std::string_view` literal).

That said, initializing a `std::string_view` using a `std::string_view` literal won't cause problems (as such literals are actually C-style string literals in disguise).

constexpr `std::string_view`

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

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

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

    return 0;
}
```

This makes `constexpr std::string_view` the preferred choice when string symbolic constants are needed.

We will continue discussing `std::string_view` in the next lesson.