In lesson 5.2 -- Literals, we introduced C-style string literals:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!"; // "Hello world!" is a C-style string literal.
    return 0;
}
```

While C-style string literals are fine to use, C-style string variables behave oddly, are hard to work with (e.g. you can't use assignment to assign a C-style string variable a new value), and are dangerous (e.g. if you copy a larger C-style string into the space allocated for a shorter C-style string, undefined behavior will result). In modern C++, C-style string variables are best avoided.

Fortunately, C++ has introduced two additional string types into the language that are much easier and safer to work with: `std::string` and `std::string_view` (C++17). Unlike the types we've introduced previously, `std::string` and `std::string_view` aren't fundamental types (they're class types, which we'll cover in the future). However, basic usage of each is straightforward and useful enough that we'll introduce them here.

Introducing `std::string`

The easiest way to work with strings and string objects in C++ is via the `std::string` type, which lives in the <string> header.

We can create objects of type `std::string` just like other objects:

```
#include <string> // allows use of std::string

int main()
{
    std::string name {}; // empty string

    return 0;
}
```

Just like normal variables, you can initialize or assign values to std::string objects as you would expect:

```
#include <string>

int main()
{
    std::string name { "Alex" }; // initialize name with string literal "Alex"
    name = "John";               // change name to "John"

    return 0;
}
```

Note that strings can be composed of numeric characters as well:

```
std::string myID{ "45" }; // "45" is not the same as integer 45!
```

In string form, numbers are treated as text, not as numbers, and thus they can not be manipulated as numbers (e.g. you can't multiply them). C++ will not automatically convert strings to integer or floating point values or vice-versa (though there are ways to do so that we'll cover in a future lesson).

String output with `std::cout`

`std::string` objects can be output as expected using `std::cout`:

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

int main()
{
    std::string name { "Alex" };
    std::cout << "My name is: " << name << '\n';

    return 0;
}
```

This prints:

```
My name is: Alex
```

Empty strings will print nothing:

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

int main()
{
    std::string empty{ };
    std::cout << '[' << empty << ']';

    return 0;
}
```

Which prints:

```
[]
```

`std::string` can handle strings of different lengths

One of the neatest things that `std::string` can do is store strings of different lengths:

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

int main()
{
    std::string name { "Alex" }; // initialize name with string literal "Alex"
    std::cout << name << '\n';

    name = "Jason";              // change name to a longer string
    std::cout << name << '\n';

    name = "Jay";                // change name to a shorter string
    std::cout << name << '\n';

    return 0;
}
```

This prints:

```
Alex
Jason
Jay
```

In the above example, `name` is initialized with the string `"Alex"`, which contains five characters (four explicit characters and a null-terminator). We then set `name` to a larger string, and then a smaller string. `std::string` has no problem handling this! You can even store really long strings in a `std::string`.

This is one of the reasons that `std::string` is so powerful.

Key insight

If `std::string` doesn't have enough memory to store a string, it will request additional memory (at runtime) using a form of memory allocation known as dynamic memory allocation. This ability to acquire additional memory is part of what makes `std::string` so flexible, but also comparatively slow.

We cover dynamic memory allocation in a future chapter.

String input with `std::cin`

Using `std::string` with `std::cin` may yield some surprises! Consider the following example:

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

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::cin >> name; // this won't work as expected since std::cin breaks on whitespace

    std::cout << "Enter your favorite color: ";
    std::string color{};
    std::cin >> color;

    std::cout << "Your name is " << name << " and your favorite color is " << color << '\n';

    return 0;
}
```

Here's the results from a sample run of this program:

```
Enter your full name: John Doe
Enter your favorite color: Your name is John and your favorite color is Doe
```

Hmmm, that isn't right! What happened? It turns out that when using `operator>>` to extract a string from `std::cin`, `operator>>` only returns characters up to the first whitespace it encounters. Any other characters are left inside `std::cin`, waiting for the next extraction.

So when we used `operator>>` to extract input into variable `name`, only `"John"` was extracted, leaving `" Doe"` inside `std::cin`. When we then used `operator>>` to get extract input into variable `color`, it extracted `"Doe"` instead of waiting for us to input an color. Then the program ends.

Use `std::getline()` to input text

To read a full line of input into a string, you're better off using the `std::getline()` function instead. `std::getline()` requires two arguments: the first is `std::cin`, and the second is your string variable.

Here's the same program as above using `std::getline()`:

```
#include <iostream>
#include <string> // For std::string and std::getline

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::getline(std::cin >> std::ws, name); // read a full line of text into name

    std::cout << "Enter your favorite color: ";
    std::string color{};
    std::getline(std::cin >> std::ws, color); // read a full line of text into color

    std::cout << "Your name is " << name << " and your favorite color is " << color << '\n';

    return 0;
}
```

Now our program works as expected:

```
Enter your full name: John Doe
Enter your favorite color: blue
Your name is John Doe and your favorite color is blue
```

What the heck is `std::ws`?

In lesson 4.8 -- Floating point numbers, we discussed output manipulators, which allow us to alter the way output is displayed. In that lesson, we used the output manipulator function `std::setprecision()` to change the number of digits of precision that `std::cout` displayed.

C++ also supports input manipulators, which alter the way that input is accepted. The `std::ws` input manipulator tells `std::cin` to ignore any leading whitespace before extraction. Leading whitespace is any whitespace character (spaces, tabs, newlines) that occur at the start of the string.

Let's explore why this is useful. Consider the following program:

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

int main()
{
    std::cout << "Pick 1 or 2: ";
    int choice{};
    std::cin >> choice;

    std::cout << "Now enter your name: ";
    std::string name{};
    std::getline(std::cin, name); // note: no std::ws here

    std::cout << "Hello, " << name << ", you picked " << choice << '\n';

    return 0;
}
```

Here's some output from this program:

```
Pick 1 or 2: 2
Now enter your name: Hello, , you picked 2
```

This program first asks you to enter 1 or 2, and waits for you to do so. All good so far. Then it will ask you to enter your name. However, it won't actually wait for you to enter your name! Instead, it prints the "Hello" string, and then exits.

When you enter a value using `operator>>`, `std::cin` not only captures the value, it also captures the newline character (`'\n'`) that occurs when you hit the enter key. So when we type `2` and then hit enter, `std::cin` captures the string `"2\n"` as input. It then extracts the value `2` to variable `choice`, leaving the newline character behind for later. Then, when `std::getline()` goes to extract text to `name`, it sees `"\n"` is already waiting in `std::cin`, and figures we must have previously entered an empty string! Definitely not what was intended.

We can amend the above program to use the `std::ws` input manipulator, to tell `std::getline()` to ignore any leading whitespace characters:

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

int main()
{
    std::cout << "Pick 1 or 2: ";
    int choice{};
    std::cin >> choice;

    std::cout << "Now enter your name: ";
    std::string name{};
    std::getline(std::cin >> std::ws, name); // note: added std::ws here

    std::cout << "Hello, " << name << ", you picked " << choice << '\n';

    return 0;
}
```

Now this program will function as intended.

```
Pick 1 or 2: 2
Now enter your name: Alex
Hello, Alex, you picked 2
```

Best practice

If using `std::getline()` to read strings, use `std::cin >> std::ws` input manipulator to ignore leading whitespace. This needs to be done for each `std::getline()` call, as `std::ws` is not preserved across calls.

Key insight

When extracting to a variable, the extraction operator (`>>`) ignores leading whitespace. It stops extracting when encountering non-leading whitespace.

`std::getline()` does not ignore leading whitespace. If you want it to ignore leading whitespace, pass `std::cin >> std::ws` as the first argument. It stops extracting when encountering a newline.

The length of a `std::string`

If we want to know how many characters are in a `std::string`, we can ask a `std::string` object for its length. The syntax for doing this is different than you've seen before, but is pretty straightforward:

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

int main()
{
    std::string name{ "Alex" };
    std::cout << name << " has " << name.length() << " characters\n";

    return 0;
}
```

This prints:

```
Alex has 4 characters
```

Although `std::string` is required to be null-terminated (as of C++11), the returned length of a `std::string` does not include the implicit null-terminator character.

Note that instead of asking for the string length as `length(name)`, we say `name.length()`. The `length()` function isn't a normal standalone function -- it's a special type of function that is nested within `std::string` called a *member function*. Because the `length()` member function is declared inside of `std::string`, it is sometimes written as `std::string::length()` in documentation.

We'll cover member functions, including how to write your own, in more detail later.

Key insight

With normal functions, we call `function(object)`. With member functions, we call `object.function()`.

Also note that `std::string::length()` returns an unsigned integral value (most likely of type `size_t`). If you want to assign the length to an `int` variable, you should `static_cast` it to avoid compiler warnings about signed/unsigned conversions:

```
int length { static_cast<int>(name.length()) };
```

In C++20, you can also use the `std::ssize()` function to get the length of a `std::string` as a large signed integral type (usually `std::ptrdiff_t`):

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

int main()
{
    std::string name{ "Alex" };
    std::cout << name << " has " << std::ssize(name) << " characters\n";

    return 0;
}
```

Since a `ptrdiff_t` may be larger than an `int`, if you want to store the result of `std::ssize()` in an `int` variable, you should `static_cast` the result to an `int`:

```
int len { static_cast<int>(std::ssize(name)) };
```

Initializing a `std::string` is expensive

Whenever a std::string is initialized, a copy of the string used to initialize it is made. Making copies of strings is expensive, so care should be taken to minimize the number of copies made.

Do not pass `std::string` by value

When a `std::string` is passed to a function by value, the `std::string` function parameter must be instantiated and initialized with the argument. This results in an expensive copy. We'll discuss what to do instead (use `std::string_view`) in lesson 5.10 -- Introduction to std::string_view.

Best practice

Do not pass `std::string` by value, as it makes an expensive copy.

Tip

In most cases, use a `std::string_view` parameter instead (covered in lesson 5.10 -- Introduction to std::string_view).

Returning a `std::string`

When a function returns by value to the caller, the return value is normally copied from the function back to the caller. So you might expect that you should not return `std::string` by value, as doing so would return an expensive copy of a `std::string`.

However, as a rule of thumb, it is okay to return a `std::string` by value when the expression of the return statement resolves to any of the following:

- A local variable of type `std::string`.
- A `std::string` that has been returned by value from another function call or operator.
- A `std::string` temporary that is created as part of the return statement.

For advanced readers

`std::string` supports a capability called move semantics, which allows an object that will be destroyed at the end of the function to instead be returned by value without making a copy. How move semantics works is beyond the scope of this introductory article, but is something we introduce in lesson 16.5 -- Returning std::vector, and an introduction to move semantics.

In most other cases, prefer to avoid returning a `std::string` by value, as doing so will make an expensive copy.

Tip

If returning a C-style string literal, use a `std::string_view` return type instead (covered in lesson 5.11 -- std::string_view (part 2)).

For advanced readers

In certain cases, `std::string` may also be returned by (const) reference, which is another way to avoid making a copy. We discuss this further in lessons 12.12 -- Return by reference and return by address and 14.6 -- Access functions.

Literals for `std::string`

Double-quoted string literals (like “Hello, world!”) are C-style strings by default (and thus, have a strange type).

We can create string literals with type `std::string` by using a `s` suffix after the double-quoted string literal. The `s` must be lower case.

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

int main()
{
    using namespace std::string_literals; // easy access to the s 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

    return 0;
}
```

Tip

The "s" suffix lives in the namespace `std::literals::string_literals`.

The most concise way to access the literal suffixes is via using-directive `using namespace std::literals`. However, this imports *all* of the standard library literals into the scope of the using-directive, which brings in a bunch of stuff you probably aren't going to use.

We recommend `using namespace std::string_literals`, which imports only the literals for `std::string`.

We discuss using-directives in lesson 7.12 -- Using declarations and using directives. This is one of the exception cases where `using` an entire namespace is generally okay, because the suffixes defined within are unlikely to collide with any of your code. Avoid such using-directives outside of functions in header files.

You probably won't need to use `std::string` literals very often (as it's fine to initialize a `std::string` object with a C-style string literal), but we'll see a few cases in future lessons (involving type deduction) where using `std::string` literals instead of C-style string literals makes things easier (see 10.8 -- Type deduction for objects using the auto keyword for an example).

For advanced readers

`"Hello"s` resolves to `std::string { "Hello", 5 }` which creates a temporary `std::string` initialized with C-style string literal "Hello" (which has a length of 5, excluding the implicit null-terminator).

Constexpr strings

If you try to define a `constexpr std::string`, your compiler will probably generate an error:

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

int main()
{
    using namespace std::string_literals;

    constexpr std::string name{ "Alex"s }; // compile error

    std::cout << "My name is: " << name;

    return 0;
}
```

This happens because `constexpr std::string` isn't supported at all in C++17 or earlier, and only works in very limited cases in C++20/23. If you need constexpr strings, use `std::string_view` instead (discussed in lesson 5.10 -- Introduction to std::string_view).

Conclusion

`std::string` is complex, leveraging many language features that we haven't covered yet. Fortunately, you don't need to understand these complexities to use `std::string` for simple tasks, like basic string input and output. We encourage you to start experimenting with strings now, and we'll cover additional string capabilities later.

Quiz time

Question #1

Write a program that asks the user to enter their full name and their age. As output, tell the user the sum of their age and the number of letters in their name (use the `std::string::length()` member function to get the length of the string). For simplicity, count any spaces in the name as a letter.

Sample output:

```
Enter your full name: John Doe
Enter your age: 32
Your age + length of name is: 40
```

Reminder: We need to be careful not to mix signed and unsigned values. `std::string::length()` returns an unsigned value. If you're C++20 capable, use `std::ssize()` to get the length as a signed value. Otherwise, static_cast the return value of `std::string::length()` to an int.