4.12 — An introduction to std::string

ALEX AUGUST 29, 2021

The very first C++ program you wrote probably looked something like this:

```
#include <iostream>

int main()
{
    std::cout << "Hello,
world!\n";
    return 0;
}
```

So what is “Hello, world!” exactly? “Hello, world!” is a collection of sequential characters called a **string**. In C++, we use strings to represent text such as names, addresses, words, and sentences. String literals (such as “Hello, world!\n”) are placed between double quotes to identify them as strings.

Because strings are commonly used in programs, most modern programming languages include a fundamental string data type. In C++, strings aren’t a fundamental type (they’re actually a `compound type`, and defined in the C++ standard library rather than as part of the core language). But strings are straightforward and useful enough that we’ll introduce them here rather than wait until the chapter on compound types (chapter 9).

std::string

To use strings in C++, we first need to #include the <string> header to bring in the declarations for std::string. Once that is done, we can define variables of type std::string.

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

std::string myName {}; // empty string
```

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

```
std::string myName{ "Alex" }; // initialize myName with string literal
"Alex"
myName = "John"; // assign variable myName the string literal "John"
```

Note that strings can hold numbers as well:

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

In string form, numbers are treated as text, not numbers, and thus they can not be manipulated as numbers (e.g. you can't multiply them). C++ will not automatically convert string numbers to integer or floating point values.

String output

Strings can be output as expected using std::cout:

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

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

```
[]
```

String input with std::cin

Using strings 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 age: ";
    std::string age{};
    std::cin >> age;

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

    return 0;
}
```

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

```
Enter your full name: John Doe
Enter your age: Your name is John and your age is Doe
```

Hmmm, that isn't right! What happened? It turns out that when using operator>> to extract a string from 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 a string into variable `name`, only `"John"` was extracted, leaving `" Doe"` inside std::cin. When we then used operator>> to get variable `age`, it extracted `"Doe"` instead of waiting for us to input an age. 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() takes two parameters: the first is std::cin, and the second is your string variable.

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

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

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 age: ";
    std::string age{};
    std::getline(std::cin >> std::ws, age); // read a full line of text into age

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

    return 0;
}
```

Now our program works as expected:

```
Enter your full name: John Doe
Enter your age: 23
Your name is John Doe and your age is 23
```

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.

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

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

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. What happened?

It turns out, 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 gets the string `"2\n"`. It then extracts the `2` to variable `choice`, leaving the newline character behind for later. Then, when std::getline() goes to read the name, it sees `"\n"` is already in the stream, 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 <string>
#include <iostream>

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 the `std::ws` `input manipulator` to ignore leading whitespace.

Key insight

Using the extraction operator (>>) with std::cin ignores leading whitespace.
std::getline does not ignore leading whitespace unless you use input manipulator std::ws.

String length

If we want to know how many characters are in a std::string, we can ask the std::string 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 myName{ "Alex" };
    std::cout << myName << " has " << myName.length() << " characters\n";
    return 0;
}
```

This prints:

```
Alex has 4 characters
```

Note that instead of asking for the string length as `length(myName)`, we say `myName.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 `length()` lives within 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.

Also note that std::string::length() returns an unsigned integral value (most likely 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:

```
1 int length =
  static_cast<int>(myName.length());
```

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. For simplicity, count 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: std::string::length() returns an unsigned int. You should static_cast this to an int before adding the age so you don't mix signed and unsigned values.

Show Solution

Next lesson
4.13 Literals

Back to table of contents

Previous lesson
4.11 Chars

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest