4.4b — An introduction to std::string

BY ALEX ON MAY 8TH, 2015 | LAST MODIFIED BY ALEX ON JUNE 7TH, 2018

What is a string?

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

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    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!”) are placed between double quotes to identify them as a string.

Because strings are commonly used in programs, most modern languages include a built-in string data type. C++ includes one, not as part of the core language, but as part of the standard library.

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>

std::string myName;
```

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 input and output

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

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

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

    return 0;
}
```

This prints:

```
My name is: Alex
```

However, using strings with std::cin may yield some surprises! Consider the following example:

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

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;
}
```

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 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, waiting for the next extraction. When we used operator>> again to extract a string into variable age, we got "Doe" instead of "23". If we had done a third extraction, we would have gotten "23".

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():

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

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

    std::cout << "Enter your age: ";
    std::string age;
    std::getline(std::cin, age); // read a full line of text into age

    std::cout << "Your name is " << name << " and your age is " << age;
}
```

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
```

Mixing std::cin and std::getline()

Reading inputs with both std::cin and std::getline may cause some unexpected behavior. Consider the following:

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

```

```
int main()
{
    std::cout << "Pick 1 or 2: ";
    int choice { 0 };
    std::cin >> choice;

    std::cout << "Now enter your name: ";
    std::string name;
    std::getline(std::cin, name);

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

    return 0;
}
```

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" line, and then exits. What happened?

It turns out, when you enter a numeric value using cin, cin not only captures the numeric value, it also captures the newline. So when we enter 2, cin actually gets the string "2\n". It then extracts the 2 to variable choice, leaving the newline stuck in the input stream. Then, when std::getline goes to read the name, it sees "\n" is already in the stream, and figures we must have entered an empty string! Definitely not what was intended.

A good rule of thumb is that after reading a numeric value with std::cin, remove the newline from the stream. This can be done using the following:

```
std::cin.ignore(32767, '\n'); // ignore up to 32767 characters until a \n is removed
```

If we insert this line directly after reading variable choice, the extraneous newline will be removed from the stream, and the program will work as expected!

```
int main()
{
    std::cout << "Pick 1 or 2: ";
    int choice { 0 };
    std::cin >> choice;

    std::cin.ignore(32767, '\n'); // ignore up to 32767 characters until a \n is removed

    std::cout << "Now enter your name: ";
    std::string name;
    std::getline(std::cin, name);

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

    return 0;
}
```

Rule: If reading numeric values with std::cin, it's a good idea to remove the extraneous newline using std::cin.ignore().

What's that 32767 magic number in your code?

That tells std::cin.ignore() how many characters to ignore up to. We picked that number because it's the largest signed value guaranteed to fit in a (2-byte) integer on all platforms.

Technically, the correct way to ignore an unlimited amount of input is as follows:

```
#include <limits>

...

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // ignore unlimited characters until
 a \n is removed
```

But this requires remembering (or looking up) that horrendous line of code, as well as remembering what header to include. Most of the time you won't need to ignore more than a line or two of buffered input, so for practical purposes, 32767 works about as well, and has the benefit of being something you can actually remember in your head.

Throughout these tutorials, we use 32767 for this reason. However, it's your choice of whether you want to do it the "obscure, complex, and correct" way or the "easy, practical, but not ideal" way.

Appending strings

You can use operator+ to concatenate two strings together, or operator+= to append one string to another.

Here's an example of both, also showing what happens if you try to use operator+ to add two numeric strings together:

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

int main()
{
    std::string a("45");
    std::string b("11");

    std::cout << a + b << "\n"; // a and b will be appended, not added
    a += " volts";
    std::cout << a;

    return 0;
}
```

This prints:

```
4511
45 volts
```

Note that operator+ concatenated the strings "45" and "11" into "4511". It did not add them as numbers.

String length

If we want to know how long a string is, we can ask the string for its length. The syntax for doing this is different than you've seen before, but is pretty straightforward:

```
#include <string>
#include <iostream>
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 like we've used up to this point -- it's a special type of function that belongs to std::string called a member function. We'll cover member functions, including how to write your own, in more detail later.

Conclusion

std::string is complex, leveraging many language features that we haven't covered yet. It also has a lot of other capabilities that we haven't touched on here. 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

1) Write a program that asks the user to enter their full name and their age. As output, tell the user how many years they've lived for each letter in their name (for simplicity, count spaces as a letter).

Sample output:

```
Enter your full name: John Doe
Enter your age: 46
You've lived 5.75 years for each letter in your name.
```

Quiz solutions

1) **Hide Solution**

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

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

    std::cout << "Enter your age: ";
    int age { 0 }; // age needs to be an integer, not a string, so we can do math with it
    std::cin >> age;

    int letters = name.length(); // get number of letters in name (including spaces)
    double agePerLetter = static_cast<double>(age) / letters; // static cast age to double to avoid int
eger division
    std::cout << "You've lived " << agePerLetter << " years for each letter in your name.\n";

    return 0;
}
```



4.5 -- Enumerated types


Index


4.4a -- Explicit type conversion (casting)

272 comments to 4.4b — An introduction to std::string

« Older Comments 1 2 3 4

Arumikaze
July 17, 2018 at 1:14 pm · Reply