S.4.4b — An introduction to std::string

BY ALEX ON MAY 8TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

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 << '\n';

    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 << '\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 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 then used operator>> to get variable age, it extracted "Doe" instead of waiting for us to input an age. We are never given a chance to enter an age.

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 << '\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
```

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{};
    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 value using cin, cin not only captures the 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 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{};
    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;
```

```
16 }
```

Rule: If reading 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 charact
```

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 (returning a new string), or operator+= to append a string to the end of an existing string).

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 concatenated
    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) **Show Solution**

 **S.4.4c -- Using a language reference**

 **Index**

 **6.x -- Chapter 6 summary and quiz**

C++ TUTORIAL | PRINT THIS POST

473 comments to S.4.4b — An introduction to std::string

« Older Comments 1 ... 4 5 6

raavann
January 16, 2020 at 6:02 am · Reply

i tried removing #include <string>
from the above examples,
and they ran fine??

nascardriver
January 16, 2020 at 6:13 am · Reply

Your might be including itself. This isn't guaranteed and your code won't work with other compilers. If you use something, include its header.

kavin
January 9, 2020 at 7:36 am · Reply

myName.length() defaults to "unsigned int" because length is always a +ve value. So every time i have to use .length() for a calculation involving int, i have to use static_cast to convert it to int ?

nascardriver
January 9, 2020 at 7:40 am · Reply

Yes

Mercedes
January 2, 2020 at 11:35 pm · Reply

I don't know if this is the ideal way but its working I guess.
Is there anything I should change?

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

int main() {

    std::cout << "Enter your full name: ";
    std::string name{};
    std::getline(std::cin, name);

    std::cout << "Enter your age: ";
    int age{};
    std::cin >> age;

    double userLived{static_cast<double>(age) / name.length()};

    std::cout << "You've lived " << userLived << " years for each letter in your name.\n";

    return 0;
}
```

nascardriver
January 3, 2020 at 6:52 am · Reply

Looks good :)

Jake
December 30, 2019 at 6:07 pm · Reply

Hi Alex and company.

My first draft of the code using getline and ignore looked like this:

```
std::cout << "Enter your full name: ";   // Prompt for full name
std::getline(std::cin, fullName);   // Get full name from user
std::cin.ignore(10000, '\n');       // Flush new-line from the buffer
```

When it ran it failed to prompt for age - it just hung. ON impulse I just pressed <Enter> and got the age prompt. I then commented out the line with the cin.ignore() call and it worked a I would have expected. So I guess the release of VS I'm using (2019, Version 4.8.03761) does not leave the \n as warned in your section on "Mixing std::cin and std::getline()".

Just thought that might be useful to know.

nascardriver
December 31, 2019 at 5:32 am · Reply

`std::cin >>` leaves the '\n' in the input stream.
`std::getline` doesn't.
You only need to clear the stream if you use `std::cin >>` before `std::getline`, not the other way around.

Micael Starfire
December 7, 2019 at 12:40 pm · Reply

For ignore(), would this work to remove both the magic number and the difficult to read function call?

```cpp
#include<limits>
```

```
...

const long maxStream{std::numeric_limits<streamsize>::max()};
```

and later calling ignore() with:

```
std::cin.ignore(maxStream, '\n');
```

Also, if not long, which data type should I be using for the constant?

nascardriver
December 8, 2019 at 4:30 am · Reply

The type is `std::streamsize`. Use `constexpr` for compile-time constants.

```
constexpr std::streamsize maxStream{ std::numeric_limits<std::streamsize>::max() };
```

You're right, this removes the magic number and the long function call.

Ged
November 2, 2019 at 4:10 am · Reply

This code works as well. Why do we need another variable for myName.length()? Isn't it already and integer type? If it's lower than int won't it just get promoted?

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

int main()
{
    std::cout << "Enter your name: ";
    std::string myName = "";
    getline(std::cin, myName);

    std::cout <<  "Enter your age: ";
    int age = 0;
    std::cin >> age;

    double division = static_cast<double>(age) / myName.length();

    std::cout << "You have lived " << division << " for every character in your name.";

    return 0;
}
```

nascardriver
November 2, 2019 at 4:23 am · Reply

We don't need a separate variable for the length, we also don't need one for the result, but having them there can make the code easier to follow for beginners.

- Initialize your variables with brace initializers.
- If your program prints anything, the last thing it prints should be a line feed ('\n').

Jesse
October 30, 2019 at 1:39 pm · Reply

In the solution for the quiz, I notice that all of the program is in the main() function.
Through these lessons, it was my understanding that we wanted to have functions do as little as possible, per function.
In my code, I have done such, which has made it much, much, longer, though it still does exactly the same thing.
My question is, would my code, while understandly not as efficient, be considered "wrong?"
[Code] #include <string>
#include <iostream>

```
std::string getUserName() {
    std::cout << "Please enter your full name. \n";
    std::string userName{};
    std::getline(std::cin, userName);

    std::cin.ignore(32767, '\n');

    return userName;
}

float getUserAge() {
    std::cout << "Please enter your age. \n";
    float userAge {};
    std::cin >> userAge;

    return userAge;
}

float calculateYearsPerLetters(float nameLength, float userAge) {
    float yearsPerLttrs { userAge / nameLength };

    return yearsPerLttrs;
}

int printResults(float yrsPerLtrs, float userAge, std::string userName) {
    std::cout << "You full name is: " << userName << ". \n";
    std::cout << "You are " << userAge << " years old. \n";
    std::cout << "You have lived " << yrsPerLtrs << " years for each letter in your name. \n";

}

int main() {
    std::string userName { getUserName() };
    float userAge { getUserAge() };
    float nameLength { static_cast<float>(userName.length()) };
    float yrsPerLtrs { calculateYearsPerLetters(nameLength, userAge) };
    printResults(yrsPerLtrs, userAge, userName);

    return 0;

}
```

[\Code]

nascardriver
October 31, 2019 at 4:13 am · Reply

Hi!

Closing code tags use a forward slash (/)

- Inconsistent formatting. Use your editor's auto-formatting feature.
- `printResults` is missing a return statement. Your compiler should've warned you. Make sure you enabled all warnings and read them.
- You don't need `yearsPerLttrs`, you can return right away.

> would my code, while understandly not as efficient, be considered "wrong?"
No, not at all. Your code is easier to maintain and more reusable than the quiz's solution.

HangUp
October 21, 2019 at 2:41 am · Reply

Your "Rule: If reading values with std::cin, it's a good idea to remove the extraneous newline using std::cin.ignore()." is missing a green box around it

Alex
October 22, 2019 at 6:29 pm · Reply

This lesson hasn't been updated to the new style. I'll make sure it does when it gets converted. Thanks!

Mike
October 8, 2019 at 10:41 am · Reply

I noticed for the quiz you didn't use the usually recommended uniform initialization for the following statement:

int letters = name.length();

If I use uniform initialization, I get an error b/c the value of name.length is unsigned, while int is signed. Ok, that's fine! But if I use direct or copy, then there is no error.

Why is this?

Name.length() is still unsigned, and int is still signed. How does using copy or direct initialization change this behavior?

nascardriver
October 9, 2019 at 2:32 am · Reply

Brace initialization enforces stricter type conversion rules than copy and direct initialization. It doesn't implicitly change the signedness.
Either add a `static_cast<int>` or declare `letters` as `std::size_t`.

Mike
October 9, 2019 at 7:16 am · Reply

Ah, I should've remembered that as it was discussed way back in a previous lesson. I've just got in the habit of using uniform initialization by default, and unfortunately, just forgot why I was using them. Thanks for the reminder and all your help!

HurricaneHarry
October 6, 2019 at 2:18 am · Reply

Greetings!

After testing a bit the quirks of cin, cin.ignore() and getline(), it's where I could need a helping hand understanding the "why" of the resulsts. Using codeblocks 17.12, the code of the examples reacts a tad differently for me.

The example after "cin may yield some surprises", actually yields less surprise for me regarding the output, as it works as expected just culling the second name:

Enter your full name: John Doe
Enter your age: 88
Your name is John and your age is 88

That's the result I get. If then I substitute the "cin-line" of the name-part with the "getline()-line", like in the example after, I receive the result of the first example (no second prompt).
If then I add an additional line containing a cin.ignore() like:

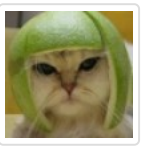
"std::cin.ignore(30000, '\n');"

it works like your second example. I checked my settings like described in the first chapter and it's set. Still I don't really understand the behaviour.

Similar it goes for the appearance of a prompt regarding cin.ignore. As it sometimes produces a prompt and sometimes not. After testing my guess is:

If I make use of cin and enter simple input like a number, cin actually queues not just the number but also the "enter character" / '\n' used to activate it inside, and just delivers back out the number, keeping the '\n'-char. That would explain the getline() behaviour as it streams the '\n' the moment the queue is used again, cancelling the input as it receives the '\n' without userinput from the queue.
Sadly, this doesn't explain the behaviour of using solely cin without getline().
And finally if the queue is empty the cin.ignore() has nothing to work on and calls a cin input before going on.

maybe someone can help me shedding a bit of light on that matter.
Thanks in advance!



Alex
October 8, 2019 at 11:57 am · Reply

I'm not sure why you're getting a different outcome. I've tested the code on quite a few different compilers and none of them produced what you're seeing. Anybody else seeing the same outcome?

HurricaneHarry
October 9, 2019 at 3:22 am · Reply

Thanks for the comment.
Searched the web for quite some time to find some more indephth documentation on the workflow of cin and cin.ignore, but only found the ever same standard stuff n examples.

But did I get the described normal worklflow right regarding those? (insert and extraction stuff)
Also that both only create a user prompt if the queue is empty and otherwise simply stream in the remainders in the queue?

Right now using getline() and cin.ignore() in tandem does work as needed, and only seems to produce one deficiency: if used together with the last call for cin it produces 2 prompts to close/finish the program instead of just one.
(I need to use the small workaround you described in the beginning to delay the closing of the

window)
Still... hate using stuff I don't fully understand.

Thanks again, for your engagement and a great project!

nascardriver
October 9, 2019 at 3:34 am · Reply

Your flow explanation is correct.
`std::cin >>` leaves the trailing line feed in the input stream.
`std::getline` extracts and discards the trailing line feed.
If you call `std::cin.ignore` on an empty stream, it will block until there is something in the stream (ie. you have to write something and press enter).
Same for `std::cin >>` and `std::getline`. If there is something in the input stream already, that data will be used (That's what the "John Doe" example tries to show).

As for your extraction issue, see if you can change or update your compiler. What you're experiencing seems like non-standard behavior.
If changing the compiler doesn't help, run your program in a terminal that you opened manually (On Windows: Shift + Right Click in the executable's directory -> "Open Command Prompt (Or powershell) here" -> then enter the executables name). If this fixes it, your IDE is messing up the input.

Parsa
September 22, 2019 at 8:25 am · Reply

Should we always be favouring std::getline() over std::cin?

nascardriver
September 23, 2019 at 8:07 am · Reply

No. Only use `std::getline` when you want to extract an entire line.

Parsa
September 23, 2019 at 4:49 pm · Reply

Is it because std::getline is slow? (or slower)

nascardriver
September 23, 2019 at 11:06 pm · Reply

It's because you should use what you need. Doing so allows the compiler to optimize your program and it shows your intentions to readers. Mixing `std::cin >>` and `std::getline` adds extra work, because they treat the trailing line feed differently.

White Russsian
September 21, 2019 at 2:06 am · Reply

I have put together this little cin/cout exercise and am struggling to understand why when I enter anything other than an integer (e.g. a letter A-Z) on the first prompt, this programme start racing

whereas if I enter an integer, it moves on to the next step, asking me whether I would like to enter another integer, as intended. Any ideas what's going on?

```
#include <iostream>

int main() {
    using std::cin;
    using std::cout;
    int x{};
    char choice{ 'y' };
    while (choice == 'y') {

        cout << "Please enter integer: ";
        cin >> x;
        cout << "You entered: " << x << '\n';

        cout << "Enter another integer? (y/n): ";
        cin >> choice;

        if (choice != 'y' and choice != 'n') {
            cout << "Invalid choice. Bye Bye!\n";
            return 0;
        }
        cout << "Choice is : " << choice << '\n';
    }
    cout << "Bye Bye!\n";

    return 0;
}
```

Output:

Please enter integer: 5
You entered: 5
Enter another integer? (y/n): y
Choice is : y
Please enter integer: H
You entered: 0
Enter another integer? (y/n): Choice is : y
Please enter integer: You entered: 0
Enter another integer? (y/n): Choice is : y
Please enter integer: You entered: 0
Enter another integer? (y/n): Choice is : y
....

nascardriver
September 21, 2019 at 2:12 am · Reply

Hi!

When extraction fails, `std::cin` enters a failed state. When `std::cin` is in a failed state, all further calls are ignored. Lesson C.5.10 deals with error handling of `std::cin`.

Kashanzulfiqar
November 19, 2019 at 5:44 pm · Reply

Be a cause you are entering a char in integer which is not possible

Hai Linh
September 5, 2019 at 6:09 am · Reply

Aside from using a magic number (32767), one could also declare a static const (or constexpr) int storing the max streamsize.

```
#include <iostream>
#include <limits>

int main()
{
    static constexpr int s_streamsize_max{std::numeric_limits<std::streamsize>::max()};
    // now we can use s_streamsize_max instead
    // the rest of the code is copied from the unexpected behaviour section
    std::cout << "Pick 1 or 2: ";
    int choice { 0 };
    std::cin >> choice;

    std::cin.ignore(s_streamsize_max, '\n'); // note s_streamsize_max

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

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

    return 0;
}
```

Additionally, we can use a typedef (or type alias):

```
typedef std::numeric_limits<std::streamsize> streamsize;
```

or

```
using streamsize = std::numeric_limits<std::streamsize>; // if your compiler was C++11 compa
```

Note that we still have to call function max() (streamsize::max())

nascardriver
September 5, 2019 at 6:13 am · Reply

`std::numeric_limits<std::streamsize>::max()` doesn't return an `int`. `s_streamsize_max` should be of type `std::streamsize`.

VerticalLimit
August 9, 2019 at 12:45 pm · Reply

Hi Nascardriver
Any suggestions to make it better ?

Quiz 1

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

int main()
{
    using namespace std;
    cout << "Now enter your full name: ";
```

```
8       string name;
9       getline(cin, name);
10
11      cout << "Enter your age: ";
12      int age{ 0 };
13      cin >> age;
14
15      double yearsLived = ((double)age / name.size());
16      cout << "Hi " << name << '\n' << "Your name has " << name.size() << " characters in it
17           << " & you've lived " << yearsLived << " years for each letter in your name \n";
18
19      return 0;
20  }
```

nascardriver
August 10, 2019 at 2:06 am · Reply

- Line 12: Initialize with empty curly braces. Only explicitly initialize if you're using the value.
- Line 15: Brace initialization.
- Line 15: Don't use C-style casts, they're unsafe.
- Limit your lines to 80 characters in length for better readability.

Piyush Pranjal
July 30, 2019 at 9:03 am · Reply

```
1   std::string bmiCheck(float x)
2   {
3       if (x >= 16.0f && x <= 18.5f)
4           return "Underweight";
5
6       if (x >= 18.5f && x <= 25.0f)
7           return "Normal";
8
9       if (x >= 25.0f && x <= 40.0f)
10          return "Over Weight";
11
12  }
```

if this function call, what we get.

My expectation to get, Underweight, Normal, Over Weight.

Am I right or wrong?

Because my program breaks when this function's call occurred.

nascardriver
July 30, 2019 at 9:08 am · Reply

Your function has to return, no matter what happens.
If `x < 16.0f` or `x > 40.0f` your function doesn't return.

You don't need all those duplicate comparisons, use `else if`.

Piyush

July 30, 2019 at 9:19 am · Reply

```
1  std::string bmiCheck(float x)
2  {
3      if (x >= 16.0f && x <= 18.5f)
4      {
5          return "Underweight";
6      }
7      else if (x >= 18.5f && x <= 25.0f)
8      {
9          return "Normal";
10     }
11     else if (x >= 25.0f && x <= 40.0f)
12     {
13         return "Over Weight";
14     }
15     else {
16         return "Unknown";
17     }
18
19 }
```

Thanks Bro.
Successfully updated my code and getting unknown at this moment.
Now I need to find what is wrong with the argument.

nascardriver
July 30, 2019 at 9:21 am · Reply

You don't need to compare `x` to `18.5f` in line 7 or `25.0f` in line 11, you already checked those.

Piyush
July 30, 2019 at 9:27 am · Reply

What happens if we get 6.0 as a parameter value. So, 9th line will be executed or not?

```
1  std::string bmiCheck(float x)
2  {
3      if (x >= 16.0f && x <= 18.5f)
4      {
5          return "Underweight";
6      }
7      else if (x <= 25.0f)
8      {
9          return "Normal";
10     }
11     else if (x <= 40.0f)
12     {
13         return "Over Weight";
14     }
15     else {
16         return "Unknown";
17     }
18
```

```
19 }
```

if this function gets 6.0 as parameter value than 9th line will be executed or not?

nascardriver
July 30, 2019 at 9:30 am · Reply

Yep, didn't see that. You can check for the lower bound first

```
1  if (x >= 16.0f)
2  {
3    if (x <= 18.5f)
4    {}
5    else if (x <= 25.0f)
6    // ...
7
8  }
9
10 return "unknown";
```

Piyush
July 30, 2019 at 9:32 am · Reply

hmmm, this one is perfect. Thank you.

Goodnight :)

you are the partner of the admin ??

Parsa
July 26, 2019 at 10:10 am · Reply

Is there a particular reason why string variables are initialized using parenthesis?

nascardriver
July 26, 2019 at 10:20 am · Reply

No, use brace initialization. You only need parenthesis if you want to call the repeat-constructor

```
1 std::string str(10, '-'); // Good, @str is "----------"
2 std::string str{ 10, '-' }; // Bad, @str is "\n-"
```

Parsa
July 26, 2019 at 10:55 am · Reply

Ok, thank you.

Parsa
July 25, 2019 at 11:32 am · Reply

Is the string library really necessary?

The type string is already defined for me without the library.

Alex
July 25, 2019 at 2:17 pm · Reply

Yes. It's likely iostream is including string on your implementation, so when you include iostream, you're getting string as a transient include. Always explicitly include the headers you use.

Parsa
July 26, 2019 at 9:52 am · Reply

Ok thank you.

Samira Ferdi
July 16, 2019 at 8:05 pm · Reply

Hi, Alex and Nascardriver!
My compiler throw an error if I do this:

```
1 int letters = name.length()
```

But, if I do this:

```
1 unsigned int letters = name.length()
```

My program works! Why is that?

nascardriver
July 17, 2019 at 4:53 am · Reply

Assuming `name` is an `std::string`.
`std::string::length` return an `std::size_t`, which is an unsigned integer type. Converting and unsigned integer to signed causes loss of precision.
Add a `static_cast<int>`. Use brace initialization.

Samira Ferdi
July 17, 2019 at 4:28 pm · Reply

Thanks, for reply!
So, you mean is

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

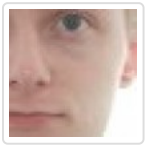
or

```
1 unsigned int letters{ name.length() } ?
```

My next question is why 32767 in std::ignore(32767, '\n')?
Why don't let's say 30, 300, 230 or 1?

nascardriver
July 17, 2019 at 11:55 pm · Reply

The first version.
Don't pass 32767 to `std::cin.ignore`. It's a magic number Alex made up because he's too lazy to type. It doesn't have any special meaning to the function.
Pass `std::numeric_limits<std::streamsize>::max()`. It causes `std::cin.ignore` to ignore everything until the specified character ('\n') is found.



Grego
July 4, 2019 at 12:48 am · Reply

Hello All,

In "Appending strings" you state that "You can use operator+ to concatenate two strings together, or operator+= to append one string to another.", however in the following code block, you say this:

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

Concatenate and append are both used when referring to '+' operator.
Hence my question is : Should I worry about the functional difference between the two at this point and if yes, what is it?

nascardriver
July 4, 2019 at 5:46 am · Reply

Hello,

I don't know which word means what. What you have to remember is that `+=` modifies the left operand, `+` creates a new string.

Grego
July 4, 2019 at 10:28 pm · Reply

Great, thanks!

« Older Comments 1 ... 4 5 6