28.5 — Stream states and input validation

ALEX[1] SEPTEMBER 9, 2024

Stream states

The ios_base class contains several state flags that are used to signal various conditions that may occur when using streams:

Flag	Meaning
goodbit	Everything is okay
badbit	Some kind of fatal error occurred (e.g. the program tried to read past the end of a file)
eofbit	The stream has reached the end of a file
failbit	A non-fatal error occurred (e.g. the user entered letters when the program was expecting an integer)

Although these flags live in ios_base, because ios is derived from ios_base and ios takes less typing than ios_base, they are generally accessed through ios (e.g. as std::ios::failbit).

ios also provides a number of member functions in order to conveniently access these states:

Member function	Meaning
good()	Returns true if the goodbit is set (the stream is ok)
bad()	Returns true if the badbit is set (a fatal error occurred)
eof()	Returns true if the eofbit is set (the stream is at the end of a file)
fail()	Returns true if the failbit is set (a non-fatal error occurred)
clear()	Clears all flags and restores the stream to the goodbit state
clear(state)	Clears all flags and sets the state flag passed in
rdstate()	Returns the currently set flags
setstate(state)	Sets the state flag passed in

The most commonly dealt with bit is the failbit, which is set when the user enters invalid input. For example, consider the following program:

```
1 std::cout << "Enter your age: ";
2 int age {};
3 std::cin >> age;
```

Note that this program is expecting the user to enter an integer. However, if the user enters non-numeric data, such as “Alex”, cin will be unable to extract anything to age, and the failbit will be set.

If an error occurs and a stream is set to anything other than goodbit, further stream operations on that stream will be ignored. This condition can be cleared by calling the clear() function.

Input validation

Input validation is the process of checking whether the user input meets some set of criteria. Input validation can generally be broken down into two types: string and numeric.

With string validation, we accept all user input as a string, and then accept or reject that string depending on whether it is formatted appropriately. For example, if we ask the user to enter a telephone number, we may want to ensure the data they enter has ten digits. In most languages (especially scripting languages like Perl and PHP), this is done via regular expressions. The C++ standard library has a regular expression library (https://en.cppreference.com/w/cpp/regex)[2] as well. Because regular expressions are slow compared to manual string validation, they should only be used if performance (compile-time and run-time) is of no concern or manual validation is too cumbersome.

With numerical validation, we are typically concerned with making sure the number the user enters is within a particular range (e.g. between 0 and 20). However, unlike with string validation, it's possible for the user to enter things that aren't numbers at all -- and we need to handle these cases too.

To help us out, C++ provides a number of useful functions that we can use to determine whether specific characters are numbers or letters. The following functions live in the cctype header:

Function	Meaning
std::isalnum(int)	Returns non-zero if the parameter is a letter or a digit
std::isalpha(int)	Returns non-zero if the parameter is a letter
std::iscntrl(int)	Returns non-zero if the parameter is a control character
std::isdigit(int)	Returns non-zero if the parameter is a digit
std::isgraph(int)	Returns non-zero if the parameter is printable character that is not whitespace
std::isprint(int)	Returns non-zero if the parameter is printable character (including whitespace)
std::ispunct(int)	Returns non-zero if the parameter is neither alphanumeric nor whitespace
std::isspace(int)	Returns non-zero if the parameter is whitespace
std::isxdigit(int)	Returns non-zero if the parameter is a hexadecimal digit (0-9, a-f, A-F)

String validation

Let's do a simple case of string validation by asking the user to enter their name. Our validation criteria will be that the user enters only alphabetic characters or spaces. If anything else is encountered, the input will be rejected.

When it comes to variable length inputs, the best way to validate strings (besides using a regular expression library) is to step through each character of the string and ensure it meets the validation criteria. That's exactly what we'll do here, or better, that's what `std::all_of` does for us.

```cpp
#include <algorithm> // std::all_of
#include <cctype> // std::isalpha, std::isspace
#include <iostream>
#include <ranges>
#include <string>
#include <string_view>

bool isValidName(std::string_view name)
{
  return std::ranges::all_of(name, [](char ch) {
    return std::isalpha(ch) || std::isspace(ch);
  });

  // Before C++20, without ranges
  // return std::all_of(name.begin(), name.end(), [](char ch) {
  //    return std::isalpha(ch) || std::isspace(ch);
  // });
}

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

  do
  {
    std::cout << "Enter your name: ";
    std::getline(std::cin, name); // get the entire line, including spaces
  } while (!isValidName(name));

  std::cout << "Hello " << name << "!\n";
}
```

Note that this code isn't perfect: the user could say their name was "asf w jweo s di we ao" or some other bit of gibberish, or even worse, just a bunch of spaces. We could address this somewhat by refining our validation criteria to only accept strings that contain at least one character and at most one space.

Author's note

Reader "Waldo" provides a C++20 solution (using std::ranges) that addresses these shortcomings here (#comment-571052)[3]

Now let's take a look at another example where we are going to ask the user to enter their phone number. Unlike a user's name, which is variable-length and where the validation criteria are the same for every character, a phone number is a fixed length but the validation criteria differ depending on the position of the character. Consequently, we are going to take a different approach to validating our phone number input. In this case, we're going to write a function that will check the user's input against a predetermined template to see whether it matches. The template will work as follows:

A # will match any digit in the user input.
A @ will match any alphabetic character in the user input.
A _ will match any whitespace.
A ? will match anything.
Otherwise, the characters in the user input and the template must match exactly.

So, if we ask the function to match the template "(###) ###-####", that means we expect the user to enter a '(' character, three numbers, a ')' character, a space, three numbers, a dash, and four more numbers. If any of these things doesn't match, the input will be rejected.

Here is the code:

```cpp
#include <algorithm> // std::equal
#include <cctype> // std::isdigit, std::isspace, std::isalpha
#include <iostream>
#include <map>
#include <ranges>
#include <string>
#include <string_view>

bool inputMatches(std::string_view input, std::string_view pattern)
{
    if (input.length() != pattern.length())
    {
        return false;
    }

    // This table defines all special symbols that can match a range of user input
    // Each symbol is mapped to a function that determines whether the input is valid
for that symbol
    static const std::map<char, int (*)(int)> validators{
      { '#', &std::isdigit },
      { '_', &std::isspace },
      { '@', &std::isalpha },
      { '?', [](int) { return 1; } }
    };

    // Before C++20, use
    // return std::equal(input.begin(), input.end(), pattern.begin(), [](char ch, char
mask) -> bool {
    // ...

    return std::ranges::equal(input, pattern, [](char ch, char mask) -> bool {
        auto found{ validators.find(mask) };

        if (found != validators.end())
        {
            // The pattern's current element was found in the validators. Call the
            // corresponding function.
            return (*found->second)(ch);
        }

        // The pattern's current element was not found in the validators. The
        // characters have to be an exact match.
        return ch == mask;
        }); // end of lambda
}

int main()
{
    std::string phoneNumber{};

    do
    {
        std::cout << "Enter a phone number (###) ###-####: ";
        std::getline(std::cin, phoneNumber);
    } while (!inputMatches(phoneNumber, "(###) ###-####"));

    std::cout << "You entered: " << phoneNumber << '\n';
}
```

Using this function, we can force the user to match our specific format exactly. However, this function is still subject to several constraints: if #, @, _, and ? are valid characters in the user input, this function won't work, because those symbols have been given special meanings. Also, unlike with regular expressions, there is no template symbol that means "a variable number of characters can be entered". Thus, such a template could not be used to ensure the user enters two words separated by a whitespace, because it can

not handle the fact that the words are of variable lengths. For such problems, the non-template approach is generally more appropriate.

Numeric validation

When dealing with numeric input, the obvious way to proceed is to use the extraction operator to extract input to a numeric type. By checking the failbit, we can then tell whether the user entered a number or not.

Let's try this approach:

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

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail()) // no extraction took place
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use
ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); //
clear out the bad input from the stream
            continue; // try again
        }

        if (age <= 0) // make sure age is positive
            continue;

        break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

If the user enters an integer, the extraction will succeed. std::cin.fail() will evaluate to false, skipping the conditional, and (assuming the user entered a positive number), we will hit the break statement, exiting the loop.

If the user instead enters input starting with a letter, the extraction will fail. std::cin.fail() will evaluate to true, and we will go into the conditional. At the end of the conditional block, we will hit the continue statement, which will jump back to the top of the while loop, and the user will be asked to enter input again.

However, there's one more case we haven't tested for, and that's when the user enters a string that starts with numbers but then contains letters (e.g. "34abcd56"). In this case, the starting numbers (34) will be extracted into age, the remainder of the string ("abcd56") will be left in the input stream, and the failbit will NOT be set. This causes two potential problems:

1. If you want this to be valid input, you now have garbage in your stream.
2. If you don't want this to be valid input, it is not rejected (and you have garbage in your stream).

Let's fix the first problem. This is easy:

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

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail()) // no extraction took place
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out any additional input from the stream

        if (age <= 0) // make sure age is positive
            continue;

      break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

If you don't want such input to be valid, we'll have to do a little extra work. Fortunately, the previous solution gets us half way there. We can use the gcount() function to determine how many characters were ignored. If our input was valid, gcount() should return 1 (the newline character that was discarded). If it returns more than 1, the user entered something that wasn't extracted properly, and we should ask them for new input. Here's an example of this:

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

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail()) // no extraction took place
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out any additional input from the stream
        if (std::cin.gcount() > 1) // if we cleared out more than one additional character
        {
            continue; // we'll consider this input to be invalid
        }

        if (age <= 0) // make sure age is positive
        {
            continue;
        }

        break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

Numeric validation as a string

The above example was quite a bit of work simply to get a simple value! Another way to process numeric input is to read it in as a string, then try to convert it to a numeric type. The following program makes use of that methodology:

```cpp
#include <charconv> // std::from_chars
#include <iostream>
#include <limits>
#include <optional>
#include <string>
#include <string_view>

// std::optional<int> returns either an int or nothing
std::optional<int> extractAge(std::string_view age)
{
    int result{};
    const auto end{ age.data() + age.length() }; // get end iterator of underlying C-
style string

    // Try to parse an int from age
    // If we got an error of some kind...
    if (std::from_chars(age.data(), end, result).ec != std::errc{})
    {
        return {}; // return nothing
    }

    if (result <= 0) // make sure age is positive
    {
        return {}; // return nothing
    }

    return result; // return an int value
}

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::string strAge{};

        // Try to get a line of input
        if (!std::getline(std::cin >> std::ws, strAge))
        {
            // If we failed, clean up and try again
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
        }

        // Try to extract the age
        auto extracted{ extractAge(strAge) };

        // If we failed, try again
        if (!extracted)
            continue;

        age = *extracted; // get the value
        break;
    }

  std::cout << "You entered: " << age << '\n';
}
```

Whether this approach is more or less work than straight numeric extraction depends on your validation parameters and restrictions.

As you can see, doing input validation in C++ is a lot of work. Fortunately, many such tasks (e.g. doing numeric validation as a string) can be easily turned into functions that can be reused in a wide variety of

situations.

Next lesson
28.6 Basic file I/O

4

Back to table of contents

5

Previous lesson
28.4 Stream classes for strings

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

120 COMMENTS

Newest

whoever

June 26, 2025 5:08 am PDT

std::stoi is much easier to use than std::from_chars (but it uses exceptions to indicate invalid input, which is slow)

0 Reply

Nidhi Gupta

February 23, 2025 9:49 pm PST

The ios_base class contains certain state flags—goodbit, badbit, eofbit, and failbit—to represent the current state of a stream, and member functions like good(), bad(), eof(), fail(), clear(), rdstate(), and setstate() to query and set these states. This facility is essential to input validation, the activity of checking user input according to certain standards. For strings, checking can be done by comparing each character using functions from the cctype library (like std::isalpha or std::isdigit), most commonly optimized through algorithms like std::all_of to make sure the input consists of a certain form or pattern (like a name with only letters and blanks or a telephone number that matches a fixed pattern). For numeric input, the extraction operator directly reads values, making use of error checking in terms of failbit checking, bad input erased with clear() and ignore(), and even using std::gcount() to scan for stray characters. Alternatively, reading numeric values in the form of a string and converting the latter through routines like std::from_chars allows for accurate validation. Although the process can be complex, encapsulating these verifications within reusable subroutines is one method of coping with input errors and ensuring that only correct, well-formed data is utilized.

0 Reply

SomeoneVeryConfused

January 16, 2025 9:44 am PST

Shouldn't fail() return true if failbit OR badbit is set? Im reading Stroustrup's book (PPP 2nd edition) and he makes that same mistake. I am very confused with all the contradiction coming from reference websites and others.

2 Reply

Konstantin

Reply to SomeoneVeryConfused [10] January 30, 2025 1:13 am PST

According to cppreference, you are correct.
Returns true if an error has occurred on the associated stream. Specifically, returns true if badbit or failbit is set in rdstate()

0 Reply

EmtyC

January 4, 2025 10:12 am PST

I was about to ask why did you use std::map instead of std::unordered_map but nvm, I think std::map is better for this case

0 Reply

EmtyC

January 4, 2025 10:07 am PST

There is an other option for this:

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

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail()) // no extraction took place
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out the bad input from the stream
            continue; // try again
        }

        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // clear out any additional input from the stream
        if (std::cin.gcount() > 1) // if we cleared out more than one additional character
        {
            continue; // we'll consider this input to be invalid
        }

        if (age <= 0) // make sure age is positive
        {
            continue;
        }

        break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

And it is:

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

int main()
{
    int age{};

    while (true)
    {
        std::cout << "Enter your age: ";
        std::cin >> age;

        if (std::cin.fail() || std::cin.peek() != '\n') // other way of handling
trailing garbage
        {
            std::cin.clear(); // reset the state bits back to goodbit so we can use
ignore()
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); //
clear out the bad input from the stream
            continue; // try again
        }

        // removed

        if (age <= 0) // make sure age is positive
        {
            continue;
        }

        break;
    }

    std::cout << "You entered: " << age << '\n';
}
```

Although both have an issue(of needing to enter an other '\n') against the eof char.

Last edited 6 months ago by EmtyC

1 Reply

immissingknowledge

October 18, 2024 8:33 am PDT

for some reason, 28.4 gives me a database error no matter what device i use

is it just me?

0 Reply

imgainingknowledge

Reply to immissingknowledge [11] October 20, 2024 1:09 am PDT

working now nvm

0 Reply

7LI

September 8, 2024 8:38 am PDT

The "std::from_chars" example doesn't handle overflow number properly. In the age context it's fine since you value init "result" and doesn't accept value 0, but if people default init "result" with a different value or accept value 0(and value init) then the program will succeed and return either of those values.

The failbit example handles overflow just fine.

0 Reply

Alex Author

Reply to 7LI [12] September 9, 2024 10:06 pm PDT

Fixed. Thanks for pointing this out!

0 Reply

SpaghetiInBrain

July 9, 2024 7:09 am PDT

I did it by switch. But, there is any option of create a map which takes lambda functions with different number of arguments? Only ellipse (...)?
You did great job!

```cpp
#include <algorithm> // std::equal
#include <cctype> // std::isdigit, std::isspace, std::isalpha
#include <iostream>
#include <string>
#include <vector>
#include <cassert>

class MatchPattern {
private:
    std::string m_templ{};      // only types
    std::string m_pattern{};    // chars to compare (if case "same")
    bool checkChar(int c, int d, int e) const;
public:
    MatchPattern(const std::string& templ, const std::string& pattern)
        :m_templ{ templ }, m_pattern{ pattern } {
        assert(m_templ.size() == m_pattern.size()); //check lengths
    }
    bool inputMatches(std::string_view input);
    const std::string& getTempl() const { return m_templ; }
    const std::string& getPattern() const { return m_pattern; }
};

bool MatchPattern::checkChar(const int c, const int d, const int e) const {
    // c - type of checking
    // d - char from number
    // e - char for comparison if c=='s';
    switch (c) {
    case '#':
        return std::isdigit(d) != 0;
        break;
    case '_':
        return std::isspace(d) != 0;
        break;
    case '@':
        return std::isalpha(d) != 0;
        break;
    case 's':
        return d == e;          //that's because use switch, not map (2 args)
        break;
    case '?':
        return true;
        break;
    default:
        std::cout << "Wrong encryption number" << std::endl;
        return false;
        break;
    }
}

bool MatchPattern::inputMatches(std::string_view input)
{
    if (m_templ.size() != input.size()) return false;
    int it{ -1 };
    return std::equal(m_templ.begin(), m_templ.end(), input.begin(), [this,&it](const
int& a, const int& b)
        {
            it++;
            return checkChar(a, b, m_pattern[it]);
        }
    );
}

int main()
{
    MatchPattern matching{
        "s@#????_?",
        "+        "
    };

    std::string phoneNumber{};
```

```
    do
    {
        std::cout << "Enter a phone number \n"
            << matching.getTempl() << "\n"
            << matching.getPattern() << ": \n";
        std::getline(std::cin, phoneNumber);
    } while (!matching.inputMatches(phoneNumber));

    std::cout << "You entered: " << phoneNumber << '\n';
}
```

0 Reply

EmtyC

Reply to SpaghetiInBrain [13] January 4, 2025 10:27 am PST

New feature named fold parameters (c++ 17). I have little knowledge about them as for now, but I know that they can be used with templates type and non-type parameters as well as function parameters (template<typename... T> or sum like this).
Sir Alex say to favor them over ellipsis.

Last edited 6 months ago by EmtyC

0 Reply

Roman

April 3, 2024 1:18 pm PDT

In the example above you use cool feature of the 'if' - init statement can be added into condition! Personally I didn't know that and there is no mention of it in '8.2 — If statements and blocks' . Consider to add it, please.
I double checked cppreference (https://en.cppreference.com/w/cpp/language/if) [14] and tried this (the small code sample below) in VS (with language standard set to c++20)

```
int someExpensiveCalc()
{
    // say, some expensive calculation goes here
    return 10;
}

int main()
{

    if (int i{ someExpensiveCalc()}; i > 0 && i < 100)
        std::cout << "Criteria are met\n";
    else
        std::cout << "Criteria aren't met\n";
}
```

It works! I didn't know in C++ it's possible to use inits right in if condition!

0 Reply

Alex Author

Reply to Roman [15] April 4, 2024 1:58 pm PDT

Yeah, it's a neat feature. That said, I've removed it from the example for now since it's not covered in a prior lesson. Covering it is on my todo, but a lower priority than some other things.

0 Reply

EmtyC

Reply to Alex [16] January 4, 2025 10:30 am PST

Um it's covered in 13.y Using a language reference as an example to using cppreference

0 Reply

Dongbin

March 21, 2024 6:07 am PDT

In the validator example, there is a line

```
1 | *found->second
```

should it be `found->second` instead? Dereferencing the iterator gives direct access to the `std::pair` right?

EDIT: never mind. Just saw that it the result is used in a function call. I was just not used to `*` having lower precedence than `->`

Last edited 1 year ago by Dongbin

1 Reply

Roman

Reply to Dongbin [17] April 3, 2024 1:32 pm PDT

Actually, IMO, you are right!
Modified slightly example works for me in VS (language standard is set to C++20), you can omit ampersand in the map initialization with function pointers and can skip deference during the invocation via the pointer

```cpp
bool inputMatches(std::string_view input, std::string_view pattern)
{
    if (input.length() != pattern.length())
    {
        return false;
    }

    // This table defines all special symbols that can match a range of user
input
    // Each symbol is mapped to a function that determines whether the input
is valid for that symbol
    static const std::map<char, int (*)(int)> validators{
      { '#', std::isdigit },  // <----------------- NOTE: no & here, just
store pointer to function
      { '_', std::isspace },
      { '@', std::isalpha },
      { '?', [](int) { return 1; } }
    };

    return std::ranges::equal(input, pattern, [](char ch, char mask) -> bool
{
        if (auto found{ validators.find(mask) }; found != validators.end())
        {
            // The pattern's current element was found in the validators.
Call the
            // corresponding function.
            return found->second(ch); // <----------------- NOTE: no * here;
just invoke the function via pointer
        }

        // The pattern's current element was not found in the validators. The
        // characters have to be an exact match.
        return ch == mask;
        }); // end of lambda
}

int main()
{
    std::string phoneNumber{};

    do
    {
        std::cout << "Enter a phone number (###) ###-####: ";
        std::getline(std::cin, phoneNumber);
    } while (!inputMatches(phoneNumber, "(###) ###-####"));

    std::cout << "You entered: " << phoneNumber << '\n';
}
```

I guess they are used to make the intentions more clear.

Last edited 1 year ago by Roman

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://en.cppreference.com/w/cpp/regex
3. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-571052
4. https://www.learncpp.com/cpp-tutorial/basic-file-io/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/stream-classes-for-strings/
7. https://www.learncpp.com/stream-states-and-input-validation/
8. https://www.learncpp.com/wordpress/tiga-and-wordpress-25/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-606638
11. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-603292
12. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-601750
13. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-599352
14. https://en.cppreference.com/w/cpp/language/if
15. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-595411
16. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-595436
17. https://www.learncpp.com/cpp-tutorial/stream-states-and-input-validation/#comment-594916
18. https://g.ezoic.net/privacy/learncpp.com