28.5 — Stream states and input validation

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:

```
std::cout << "Enter your age: ";
int age {};
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 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.

```
#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

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:

```
#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:

```
#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:

```
#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:

```
#include <charconv> // std::from_chars
#include <iostream>
#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 (std::from_chars(age.data(), end, result).ptr != end)
  {
    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{};
    std::getline(std::cin >> std::ws, strAge);

    auto extracted{ extractAge(strAge) };
    if (extracted) // if extracted has a value
    {
      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.