What is the best way to do input validation in C++ with cin?

Asked 15 years, 10 months ago Modified 8 years ago Viewed 27k times

8

My brother recently started learning C++. He told me a problem he encountered while trying to validate input in a simple program. He had a text menu where the user entered an integer `choice`, if they entered an invalid choice, they would be asked to enter it again (do while loop). However, if the user entered a string instead of an int, the code would break. I read various questions on stackoverflow and told him to rewrite his code along the lines of:

```
#include<iostream>
using namespace std;

int main()
{
    int a;
    do
    {
    cout<<"\nEnter a number:"
    cin>>a;
        if(cin.fail())
        {
            //Clear the fail state.
            cin.clear();
            //Ignore the rest of the wrong user input, till the end of the
line.
            cin.ignore(std::numeric_limits<std::streamsize>::max(),\
                                                    '\n');
        }
    }while(true);
    return 0;
}
```

While this worked ok, I also tried a few other ideas:
1. Using a try catch block. It didn't work. I think this is because an exception is not raised due to bad input. 2. I tried `if(! cin){//Do Something}` which didn't work either. I haven't yet figured this one out.
3. Thirdly, I tried inputting a fixed length string and then parsing it. I would use atoi(). Is this standards compliant and portable? Should I write my own parsing function?
4. If write a class that uses cin, but dynamically does this kind of error detection, perhaps by determining the type of the input variable at runtime, would it have too much overhead? Is it even possible?

I would like to know what is the best way to do this kind of checking, what are the best practices?

I would like to add that while I am not new to writing C++ code, I am new to writing good standards compliant code. I am trying to unlearn bad practices and learn the right ones. I would be much obliged if answerers give a detailed explanation.

EDIT: I see that litb has answered one of my previous edits. I'll post that code here for reference.

```
#include<iostream>
using namespace std;

int main()
{
    int a;
    bool inputCompletionFlag = true;
    do
    {
    cout<<"\nEnter a number:"
    cin>>a;
        if(cin.fail())
        {
            //Clear the fail state.
            cin.clear();
            //Ignore the rest of the wrong user input, till the end of the
line.
            cin.ignore(std::numeric_limits<std::streamsize>::max(),\
                                                    '\n');
        }
        else
        {
            inputCompletionFlag = false;
        }
    }while(!inputCompletionFlag);
    return 0;
}
```

This code fails on input like "1asdsdf". I didn't know how to fix it but litb has posted a great answer. :)

c++ coding-style validation

Share
Improve this question
Follow

edited Oct 15, 2009 at 13:36

community wiki
14 revs, 5 users 100%
batbrat

8 Answers

Sorted by: Highest score (default)

Here is code you could use to make sure you also reject things like

19

```
42crap
```

Where non-number characters follow the number. If you read the whole line and then parse it and execute actions appropriately it will possibly require you to change the way your program works. If your program read your number from different places until now, you then have to put one central place that parses one line of input, and decides on the action. But maybe that's a good thing too - so you could increase the readability of the code that way by having things separated: **I**nput - **P**rocessing - **O**utput

Anyway, here is how you can reject the number-non-number of above. Read a line into a string, then parse it with a `stringstream`:

```
std::string getline() {
  std::string str;
  std::getline(std::cin, str);
  return str;
}

int choice;
std::istringstream iss(getline());
iss >> choice >> std::ws;
if(iss.fail() || !iss.eof()) {
  // handle failure
}
```

It eats all trailing whitespace. When it hits the end-of-file of the stringstream while reading the integer or trailing whitespace, then it sets the eof-bit, and we check that. If it failed to read any integer in the first place, then the fail or bad bit will have been set.

Earlier versions of this answer used `std::cin` directly - but `std::ws` won't work well together with `std::cin` connected to a terminal (it will block instead waiting for the user to input something), so we use a `stringstream` for reading the integer.

Answering some of your questions:

Question: 1. Using a try catch block. It didn't work. I think this is because an exception is not raised due to bad input.

Answer: Well, you can tell the stream to throw exceptions when you read something. You use the `istream::exceptions` function, which you tell for which kind of error you want to have an exception thrown:

```
iss.exceptions(ios_base::failbit);
```

I did never use it. If you do that on `std::cin`, you will have to remember to restore the flags for other readers that rely on it not throwing. Finding it way easier to just use the functions *fail*, *bad* to ask for the state of the stream.

Question: 2. I tried `if(!cin){ //Do Something }` which didn't work either. I haven't yet figured this one out.

Answer: That could come from the fact that you gave it something like "42crap". For the stream, that is completely valid input when doing an extraction into an integer.

Question: 3. Thirdly, I tried inputting a fixed length string and then parsing it. I would use atoi(). Is this standards compliant and portable? Should I write my own parsing function?

Answer: atoi is Standard Compliant. But it's not good when you want to check for errors. There is no error checking, done by it as opposed to other functions. If you have a string and want to check whether it contains a number, then do it like in the initial code above.

There are C-like functions that can read directly from a C-string. They exist to allow interaction with old, legacy code and writing fast performing code. One should avoid them in programs because they work rather low-level and require using raw naked pointers. By their very nature, they can't be enhanced to work with user defined types either. Specifically, this talks about the function "strtol" (string-to-long) which is basically atoi with error checking and capability to work with other bases (hex for example).

Question: 4. If I write a class that uses cin, but dynamically do this kind of error detection, perhaps by determining the type of the input variable at runtime, will it have too much overhead? Is it even possible?

Answer: Generally, you don't need to care too much about overhead here (if you mean runtime-overhead). But it depends specifically on where you use that class. That question will be very important if you are writing a high performance system that processes input and needs to have high throughout. But if you need to read input from a terminal or a file, you already see what this comes down to: Waiting for the user to input something takes really so long, you don't need to watch runtime costs at this point anymore on this scale.

If you mean code overhead - well it depends on how the code is implemented. You would need to scan your string that you read - whether it contains a number or not, whether some arbitrary string. Depending on what you want to scan (maybe you have a "date" input, or a "time" input format too. Look into `boost.date_time` for that), your code can become arbitrarily complex. For simple things like classifying between number or not, I think you can get away with small amount of code.

Share

Improve this answer

Follow

edited Dec 21, 2016 at 16:48

abcoep

636 ● 11 ● 15

answered Feb 13, 2009 at 15:57

Johannes Schaub - litb

506k ● 131 ● 917 ● 1.2k

Thanks for the detailed explanation litb, but I looked up and found that Boost isn't a standard library. In light of that, would it be better to roll my own code if I can or should I stick with boost? – batbrat Feb 13, 2009 at 16:44

I just noticed that it is going to be standardized to a large extent in C++0x. So I'll use Boost. I'll be glad if you confirm. Thanks. – batbrat Feb 13, 2009 at 16:47

yeah, next c++ will include some libraries designed after boost ones (shared_ptr, thread, system (error_code, ...), array, bind, function). but not date_time. boost however is highly recommended. it's better than rolling your own really – Johannes Schaub - litb Feb 13, 2009 at 17:46

Thanks for the response litb. I'll go with Boost – batbrat Feb 14, 2009 at 11:04

12

This is what I do with C but it's probably applicable for C++ as well.

Input everything as a string.

Then, and only then, parse the string into what you need. It's sometimes better to code your own than try to bend someone else's to your will.

Share Improve this answer Follow

answered Feb 13, 2009 at 13:29

paxdiablo

880k ● 241 ● 1.6k ● 2k

Thanks for replying Pax. I had thought of doing what you said, but was wondering whether it was the right approach. Thanks for the answer. – batbrat Feb 13, 2009 at 16:45

4

- In order to get the exceptions with iostreams you need to set the proper exception flag for the stream.
- And I would use *get_line* to get the whole line of input and then handle it accordingly - use lexical_cast, regular expressions (for example Boost Regex or Boost Xpressive, parse it with Boost Spirit, or just use some kind of appropriate logic

Share

Improve this answer

Follow

edited Feb 13, 2009 at 13:39

answered Feb 13, 2009 at 13:31

Anonymous

18.6k ● 2 ● 41 ● 64

Thanks for the answer. I would like to know if there is a standard library equivalent to the boost alternatives mentioned here. – batbrat Feb 13, 2009 at 13:39

3

What I would do is twofold: First, try to validate the input, and extract the data, using a regular expression, if the input is somewhat not trivial. It can be very helpful also even if the input is just a series of numbers.

Then, I like to use boost::lexical_cast, that can raise a bad_ lexical_ cast exception if the input cannot be converted.

In your example:

```
std::string in_str;
cin >> in_str;

// optionally, test if it conforms to a regular expression, in case the input
is complex

// Convert to int? this will throw bad_lexical_cast if cannot be converted.
int my_int = boost::lexical_cast<int>(in_str);
```

Share Improve this answer Follow

answered Feb 13, 2009 at 13:28

Diego Sevilla
29k ● 4 ● 61 ● 89

2

Forget about using formatted input (the >> operator) directly in real code. You will always need to read raw text with std::getline or similar and then use your own input parsing routines (which may use of the >> operator) to parse the input.

Share Improve this answer Follow

answered Feb 13, 2009 at 13:30

anon

2

How about a combination of the various approaches:

1. Snag the input from `std::cin` using `std::getline(std::cin, strObj)` where `strObj` is a `std::string` object.
2. Use `boost::lexical_cast` to perform a lexical translation from `strObj` to either a signed or unsigned integer of largest width (e.g., `unsigned long long` or something similar)

3. Use `boost::numeric_cast` to cast the integer down to the expected range.

You could just fetch the input with `std::getline` and then call `boost::lexical_cast` to the appropriately narrow integer type as well depending on where you want to catch the error. The three step approach has the benefit of accepting any integer data and then catch narrowing errors separately.

Share Improve this answer Follow

answered Feb 13, 2009 at 14:58

D.Shawley

59.5k ● 10 ● 105 ● 116

1

I agree with Pax, the simplest way to do this is to read everything as string, then use TryParse to verify the input. If it is in the right format, then proceed, otherwhise just notify the user and use continue on the loop.

Share Improve this answer Follow

answered Feb 13, 2009 at 15:28

David Božjak

17.6k ● 18 ● 70 ● 98

TryParse is a .NET-specific idiom. Portable, standards-based C++ can't use it.
– Harper Shelby Feb 13, 2009 at 16:20

I was wondering about that harper. thanks for clarifying. Thanks for answering Rekreativc
– batbrat Feb 13, 2009 at 16:42

1

One thing that hasn't been mentioned yet is that it is usually important that you test to see if the cin >> operation worked before using the variable that supposedly got something from the stream.

This example is similar to yours, but makes that test.

```
#include <iostream>
#include <limits>
using namespace std;
int main()
{
   while (true)
   {
      cout << "Enter a number: " << flush;
      int n;
      if (cin >> n)
      {
         // do something with n
         cout << "Got " << n << endl;
      }
      else
```

```
      {
         cout << "Error! Ignoring..." << endl;
         cin.clear();
         cin.ignore(numeric_limits<streamsize>::max(), '\n');
      }
   }
   return 0;
}
```

This will use the usual operator >> semantics; it will skip whitespace first, then try to read as many digits as it can and then stop. So "42crap" will give you the 42 then skip over the "crap". If that isn't what you want, then I agree with the previous answers, you should read it into a string and then validate it (perhaps using a regular expression - but that may be overkill for a simple numeric sequence).

Share

Improve this answer

Follow

answered Feb 13, 2009 at 17:03

community wiki
Brian Neal

> It is what I want. Thanks. however, the above answers made me rethink the way I was doing things and I have a better idea about how to write good standards compliant code. I'll use both your and the others suggestions in the future. – batbrat Feb 13, 2009 at 17:16