5.10 — std::cin, extraction, and dealing with invalid text input

BY ALEX ON APRIL 21ST, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Most programs that have a user interface of some kind need to handle user input. In the programs that you have been writing, you have been using std::cin to ask the user to enter text input. Because text input is so free-form (the user can enter anything), it's very easy for the user to enter input that is not expected.

As you write programs, you should always consider how users will (unintentionally or otherwise) misuse your programs. A well-written program will anticipate how users will misuse it, and either handle those cases gracefully or prevent them from happening in the first place (if possible). A program that handles error cases well is said to be **robust**.

In this lesson, we'll take a look specifically at ways the user can enter invalid text input via std::cin, and show you some different ways to handle those cases.

std::cin, buffers, and extraction

In order to discuss how std::cin and operator>> can fail, it first helps to know a little bit about how they work.

When we use operator>> to get user input and put it into a variable, this is called an "extraction". The >> operator is accordingly called the extraction operator when used in this context.

When the user enters input in response to an extraction operation, that data is placed in a buffer inside of std::cin. A **buffer** (also called a data buffer) is simply a piece of memory set aside for storing data temporarily while it's moved from one place to another. In this case, the buffer is used to hold user input while it's waiting to be extracted to variables.

When the extraction operator is used, the following procedure happens:

- If there is data already in the input buffer, that data is used for extraction.
- If the input buffer contains no data, the user is asked to input data for extraction (this is the case most of the time). When the user hits enter, a '\n' character will be placed in the input buffer.
- operator>> extracts as much data from the input buffer as it can into the variable (ignoring any leading whitespace characters, such as spaces, tabs, or '\n').
- Any data that can not be extracted is left in the input buffer for the next extraction.

Extraction succeeds if at least one character is extracted from the input buffer. Any unextracted input is left in the input buffer for future extractions. For example:

```cpp
int x;
std::cin >> x;
```

If the user enters "5a", 5 will be extracted, converted to an integer, and assigned to variable x. "a\n" will be left in the input stream for the next extraction.

Extraction fails if the input data does not match the type of the variable being extracted to. For example:

```cpp
int x;
std::cin >> x;
```

If the user were to enter 'b', extraction would fail because 'b' can not be extracted to an integer variable.

Validating input

The process of checking whether user input conforms to what the program is expecting is called **input validation**.

There are three basic ways to do input validation:

- Inline (as the user types)
    - Prevent the user from typing invalid input in the first place.
- Post-entry (after the user types)
    - Let the user enter whatever they want into a string, then validate whether the string is correct, and if so, convert the string to the final variable format.
    - Let the user enter whatever they want, let std::cin and operator>> try to extract it, and handle the error cases.

Some graphical user interfaces and advanced text interfaces will let you validate input as the user enters it (character by character). Generally speaking, the programmer provides a validation function that accepts the input the user has entered so far, and returns true if the input is valid, and false otherwise. This function is called every time the user presses a key. If the validation function returns true, the key the user just pressed is accepted. If the validation function returns false, the character the user just input is discarded (and not shown on the screen). Using this method, you can ensure that any input the user enters is guaranteed to be valid, because any invalid keystrokes are discovered and discarded immediately. Unfortunately, std::cin does not support this style of validation.

Since strings do not have any restrictions on what characters can be entered, extraction is guaranteed to succeed (though remember that std::cin stops extracting at the first non-leading whitespace character). Once a string is entered, the program can then parse the string to see if it is valid or not. However, parsing strings and converting string input to other types (e.g. numbers) can be challenging, so this is only done in rare cases.

Most often, we let std::cin and the extraction operator do the hard work. Under this method, we let the user enter whatever they want, have std::cin and operator>> try to extract it, and deal with the fallout if it fails. This is the easiest method, and the one we'll talk more about below.

A sample program

Consider the following calculator program that has no error handling:

```
#include <iostream>

double getDouble()
{
    std::cout << "Enter a double value: ";
    double x;
    std::cin >> x;
    return x;
}

char getOperator()
{
    std::cout << "Enter one of the following: +, -, *, or /: ";
    char op;
    std::cin >> op;
    return op;
}

void printResult(double x, char op, double y)
{
    if (op == '+')
        std::cout << x << " + " << y << " is " << x + y << '\n';
    else if (op == '-')
        std::cout << x << " - " << y << " is " << x - y << '\n';
    else if (op == '*')
        std::cout << x << " * " << y << " is " << x * y << '\n';
    else if (op == '/')
        std::cout << x << " / " << y << " is " << x / y << '\n';
}

```

```
int main()
{
    double x = getDouble();
    char op = getOperator();
    double y = getDouble();

    printResult(x, op, y);

    return 0;
}
```

This simple program asks the user to enter two numbers and a mathematical operator.

```
Enter a double value: 5
Enter one of the following: +, -, *, or /: *
Enter a double value: 7
5 * 7 is 35
```

Now, consider where invalid user input might break this program.

First, we ask the user to enter some numbers. What if they enter something other than a number (e.g. 'q')? In this case, extraction will fail.

Second, we ask the user to enter one of four possible symbols. What if they enter a character other than one of the symbols we're expecting? We'll be able to extract the input, but we don't currently handle what happens afterward.

Third, what if we ask the user to enter a symbol and they enter a string like "*q hello". Although we can extract the '*' character we need, there's additional input left in the buffer that could cause problems down the road.

Types of invalid text input

We can generally separate input text errors into four types:

- Input extraction succeeds but the input is meaningless to the program (e.g. entering 'k' as your mathematical operator).
- Input extraction succeeds but the user enters additional input (e.g. entering '*q hello' as your mathematical operator).
- Input extraction fails (e.g. trying to enter 'q' into a numeric input).
- Input extraction succeeds but the user overflows a numeric value.

Thus, to make our programs robust, whenever we ask the user for input, we ideally should determine whether each of the above can possibly occur, and if so, write code to handle those cases.

Let's dig into each of these cases, and how to handle them using std::cin.

Error case 1: Extraction succeeds but input is meaningless

This is the simplest case. Consider the following execution of the above program:

```
Enter a double value: 5
Enter one of the following: +, -, *, or /: k
Enter a double value: 7
```

In this case, we asked the user to enter one of four symbols, but they entered 'k' instead. 'k' is a valid character, so std::cin happily extracts it to variable op, and this gets returned to main. But our program wasn't expecting this to happen, so it doesn't properly deal with this case (and thus never outputs anything).

The solution here is simple: do input validation. This usually consists of 3 steps:

1) Check whether the user's input was what you were expecting.
2) If so, return the value to the caller.
3) If not, tell the user something went wrong and have them try again.

Here's an updated getOperator() function that does input validation.

```
char getOperator()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char op;
        std::cin >> op;

        // Check whether the user entered meaningful input
        if (op == '+' || op == '-' || op == '*' || op == '/')
            return op; // return it to the caller
        else // otherwise tell the user what went wrong
            std::cout << "Oops, that input is invalid.  Please try again.\n";
    } // and try again
}
```

As you can see, we're using a while loop to continuously loop until the user provides valid input. If they don't, we ask them to try again until they either give us valid input, shutdown the program, or destroy their computer.

Error case 2: Extraction succeeds but with extraneous input

Consider the following execution of the above program:

```
Enter a double value: 5*7
```

What do you think happens next?

```
Enter a double value: 5*7
Enter one of the following: +, -, *, or /: Enter a double value: 5 * 7 is 35
```

The program prints the right answer, but the formatting is all messed up. Let's take a closer look at why.

When the user enters "5*7" as input, that input goes into the buffer. Then operator>> extracts the 5 to variable x, leaving "*7\n" in the buffer. Next, the program prints "Enter one of the following: +, -, *, or /:". However, when the extraction operator was called, it sees "*7\n" waiting in the buffer to be extracted, so it uses that instead of asking the user for more input. Consequently, it extracts the '*' character, leaving "7\n" in the buffer.

After asking the user to enter another double value, the "7" in the buffer gets extracted without asking the user. Since the user never had an opportunity to enter additional data and hit enter (causing a newline), the output prompts all get run together on the same line, even though the output is correct.

Although the above problem works, the execution is messy. It would be better if any extraneous characters entered were simply ignored. Fortunately, that's easy to do:

```
std::cin.ignore(32767, '\n');  // clear (up to 32767) characters out of the buffer until a '\n'
```

Since the last character the user entered must be a '\n', we can tell std::cin to ignore buffered characters until it finds a newline character (which is removed as well).

Let's update our getDouble() function to ignore any extraneous input:

```
double getDouble()
{
    std::cout << "Enter a double value: ";
    double x;
    std::cin >> x;
    std::cin.ignore(32767, '\n'); // clear (up to 32767) characters out of the buffer until a '
    return x;
}
```

Now our program will work as expected, even if we enter “5*7” for the first input -- the 5 will be extracted, and the rest of the characters will be removed from the input buffer. Since the input buffer is now empty, the user will be properly asked for input the next time an extraction operation is performed!

Error case 3: Extraction fails

Now consider the following execution of the calculator program:

```
Enter a double value: a
```

You shouldn’t be surprised that the program doesn’t perform as expected, but how it fails is interesting:

```
Enter a double value: a
Enter one of the following: +, -, *, or /: Enter a double value:
```

and the program suddenly ends.

This looks pretty similar to the extraneous input case, but it’s a little different. Let’s take a closer look.

When the user enters ‘a’, that character is placed in the buffer. Then operator>> tries to extract ‘a’ to variable x, which is of type double. Since ‘a’ can’t be converted to a double, operator>> can’t do the extraction. Two things happen at this point: ‘a’ is left in the buffer, and std::cin goes into “failure mode”.

Once in ‘failure mode’, future requests for input extraction will silently fail. Thus in our calculator program, the output prompts still print, but any requests for further extraction are ignored. The program simply runs to the end and then terminates (without printing a result, because we never read in a valid mathematical operation).

Fortunately, we can detect whether an extraction has failed and fix it:

```
if (std::cin.fail()) // has a previous extraction failed?
{
    // yep, so let's handle the failure
    std::cin.clear(); // put us back in 'normal' operation mode
    std::cin.ignore(32767,'\n'); // and remove the bad input
}
```

That’s it!

Let’s integrate that into our getDouble() function:

```
double getDouble()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter a double value: ";
        double x;
        std::cin >> x;

        if (std::cin.fail()) // has a previous extraction failed?
        {
            // yep, so let's handle the failure
            std::cin.clear(); // put us back in 'normal' operation mode
            std::cin.ignore(32767,'\n'); // and remove the bad input
        }
        else // else our extraction succeeded
        {
            std::cin.ignore(32767, '\n'); // clear (up to 32767) characters out of the buffer
            return x; // so return the value we extracted
        }
    }
}
```

Note: Prior to C++11, a failed extraction would not modify the variable being extracted to. This means that if a variable was uninitialized, it would stay uninitialized in the failed extraction case. However, as of C++11, a failed extraction due to invalid input will cause the variable to be zero-initialized. Zero initialization means the variable is set to 0, 0.0, “”, or whatever value 0 converts to for that type.

Error case 4: Extraction succeeds but the user overflows a numeric value

Consider the following simple example:

```
#include <cstdint>
#include <iostream>

int main()
{
    std::int16_t x { 0 }; // x is 16 bits, holds from -32768 to 32767
    std::cout << "Enter a number between -32768 and 32767: ";
    std::cin >> x;

    std::int16_t y { 0 }; // y is 16 bits, holds from -32768 to 32767
    std::cout << "Enter another number between -32768 and 32767: ";
    std::cin >> y;

    std::cout << "The sum is: " << x + y << '\n';
    return 0;
}
```

What happens if the user enters a number that is too large (e.g. 40000)?

```
Enter a number between -32768 and 32767: 40000
Enter another number between -32768 and 32767: The sum is: 32767
```

In the above case, std::cin goes immediately into “failure mode”, but also assigns the closest in-range value to the variable. Consequently, x is left with the assigned value of 32767. Additional inputs are skipped, leaving y with the initialized value of 0. We can handle this kind of error in the same way as a failed extraction.

Note: Prior to C++11, a failed extraction would not modify the variable being extracted to. This means that if a variable was uninitialized, it would stay uninitialized in the failed extraction case. However, as of C++11, an out-of-range failed extraction will cause the variable to be set to the closest in-range value.

Putting it all together

Here’s our example calculator with full error checking:

```
#include <iostream>

double getDouble()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter a double value: ";
        double x;
        std::cin >> x;

        // Check for failed extraction
        if (std::cin.fail()) // has a previous extraction failed?
        {
            // yep, so let's handle the failure
            std::cin.clear(); // put us back in 'normal' operation mode
            std::cin.ignore(32767,'\n'); // and remove the bad input
            std::cout << "Oops, that input is invalid.  Please try again.\n";
        }
        else
        {
            std::cin.ignore(32767,'\n'); // remove any extraneous input

            // the user can't enter a meaningless double value, so we don't need to worry abou
            return x;
        }
    }
}

char getOperator()
{
    while (true) // Loop until user enters a valid input
    {
        std::cout << "Enter one of the following: +, -, *, or /: ";
        char op;
        std::cin >> op;

        // Chars can accept any single input character, so no need to check for an invalid ext

        std::cin.ignore(32767,'\n'); // remove any extraneous input

        // Check whether the user entered meaningful input
        if (op == '+' || op == '-' || op == '*' || op == '/')
            return op; // return it to the caller
        else // otherwise tell the user what went wrong
            std::cout << "Oops, that input is invalid.  Please try again.\n";
    } // and try again
}

void printResult(double x, char op, double y)
{
    if (op == '+')
        std::cout << x << " + " << y << " is " << x + y << '\n';
    else if (op == '-')
        std::cout << x << " - " << y << " is " << x - y << '\n';
    else if (op == '*')
        std::cout << x << " * " << y << " is " << x * y << '\n';
    else if (op == '/')
        std::cout << x << " / " << y << " is " << x / y << '\n';
    else // Being robust means handling unexpected parameters as well, even though getOperator
        std::cout << "Something went wrong: printResult() got an invalid operator.";

}

int main()
{
    double x = getDouble();
    char op = getOperator();
    double y = getDouble();

    printResult(x, op, y);

    return 0;
}
```

Conclusion

As you write your programs, consider how users will misuse your program, especially around text input. For each point of text input, consider:

- Could extraction fail?
- Could the user enter more input than expected?
- Could the user enter meaningless input?
- Could the user overflow an input?

You can use if statements and boolean logic to test whether input is expected and meaningful.

The following code will test for and fix failed extractions or overflow:

```
if (std::cin.fail()) // has a previous extraction failed or overflowed?
{
    // yep, so let's handle the failure
    std::cin.clear(); // put us back in 'normal' operation mode
    std::cin.ignore(32767,'\n'); // and remove the bad input
}
```

The following will also clear any extraneous input:

```
    std::cin.ignore(32767,'\n'); // and remove the bad input
```

Finally, use loops to ask the user to re-enter input if the original input was invalid.

Note: Input validation is important and useful, but it also tends to make examples more complicated and harder to follow. Accordingly, in future lessons, we will generally not do any kind of input validation unless it's relevant to something we're trying to teach.

5.11 -- Introduction to testing your code

Index

5.9 -- Random number generation

C++ TUTORIAL | PRINT THIS POST

198 comments to 5.10 — std::cin, extraction, and dealing with invalid text input

« Older Comments 1 2 3

Eric
February 3, 2020 at 1:19 pm · Reply

My program does not correctly return 'y' or 'n' at "choice = getChoice ();" in the main () (line 55). What is wrong?

```
1   #include <iostream>
2   #include <ctime>
3   #include <cstdlib>
4
5   int getRandomNumber()
6   {
7
8       int randomNumber{ std::rand() % 100 };
9       return randomNumber;
10  }
11
12  void doActivity()
13  {
14      int total{};
15      for (int counter = 1; counter <= 2; ++counter)
16      {
17          int randomNumber{ getRandomNumber() };
18          std::cout << "Random number #: " << counter << " = " << randomNumber << '\n';
19          total += randomNumber;
20      }
21
22      if (total < 100)
23          std::cout << "These total " << total << ".  Yes, this is less than 100. \n " << std
24      else
25          std::cout << "These total " << total << ".  No, this is more than 100. \n " << std:
26
27  }
28
29  char getChoice ()
```

```
30 {
31     while (true) // Loop until user enters a valid input
32     {
33         std::cout << "Do you want to do the activity? (y or n)";
34         char choice;
35         std::cin >> choice;
36 
37         // Chars can accept any single input character, so no need to check for an invalid
38 
39         std::cin.ignore(32767, '\n'); // remove any extraneous input
40 
41         // Check whether the user entered meaningful input
42         if (choice == 'y' || choice == 'n')
43             return choice; // return it to the caller
44         else // otherwise tell the user what went wrong
45             std::cout << "Oops, that input is invalid.  Please try again.\n";
46     } // and try again
47 }
48 
49 int main()
50 {
51     std::srand(static_cast<unsigned int>(std::time(nullptr))); // set initial seed value to
52     char choice{};
53     while (choice == 'y')
54     {
55         choice = getChoice ();
56         if (choice == 'y')
57             doActivity();
58     }
59     return (0);
60 }
```

nascardriver
February 4, 2020 at 8:31 am · Reply

Your loop is never entered. In line 53, you set choice to 0, in line 53 you check if it's 'y', which it isn't, so the loop doesn't run.

Ged
November 6, 2019 at 9:11 am · Reply

Code works, except for overflow. For some reason it allows me to continue.

```
#include <iostream>
#include <cstdint>

int16_t getValue()
{
    while(true)
    {
    std::cout << "Enter a number from -32768 to 32767 ";
    int x{};
    std::cin >> x;

    if(std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(32767,'\n');
        std::cout << "Incorrect input please try again" << '\n';
    }
    else
    {
        std::cin.ignore(32767,'\n');
        return x;
    }
    }
}

char getSymbol()
{
    while(true)
    {
    std::cout << "Enter a symbol * / + - ";
    char ch{};
    std::cin >> ch;

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

    if(ch == '+' || ch == '-' || ch == '/' || ch == '*')
    {
        return ch;
    }
    else
    {
        std::cout << "Wrong symbol, please try again" << '\n';
    }
    }
}

void printResult(int16_t x, char ch, int16_t y)
{
    switch(ch)
    {
        case '+':
        std::cout << "The answer is " << x + y;
        break;
        case '-':
        std::cout << "The answer is " << x - y;
        break;
        case '*':
        std::cout << "The answer is " << x * y;
        break;
        case '/':
        std::cout << "The answer is " << x / y;
        break;

    }
}

int main()
{
    int16_t x{};
    x = getValue();

    char ch{};
    ch = getSymbol();

    int16_t y{};
    y = getValue();

    printResult(x,ch,y);
    return 0;
}
```

nascardriver
November 7, 2019 at 1:49 am · Reply

- Don't use `std::int*_t`, they might not exist. Use `std::int_least*_t` or `std::int_fast*_t` instead.
- Initialize your variables instead of assigning to them.
- Use your editor's auto-formatter.
- If your program prints anything, the last thing it prints should be a line feed ('\n').

> except for overflow
Your code detects overflowed input, but your message is wrong. -32768 and 32767 are magic numbers, they don't mean anything. Valid inputs are the message should be

```
std::cout << "Enter a number from " << std::numeric_limits<int>::lowest() << " to " <<
```

This gets you the actual highest and lowest values that are allowed for `int`. Entering a number higher or lower than the ones printed will set `std::cin`'s fail state.

Gacrux
October 27, 2019 at 7:35 pm · Reply

Is this good to force user to make a valid input and avoid previous cin buffer to show up in next extractions in case of the extraction succeed, but carry bad characters? I mean, if I comment out line 20, if I input "10xx" in the first extraction, the second one will carry the "xx", trigger the failure message and the format gets messed up.

```
1   #include <iostream>
2   #include <limits>
3
4   int getNumber()
5   {
6       int x{};
7       while (true)
8       {
9           std::cout << "Enter an integer: ";
10          std:: cin >> x;
11          if (std::cin.fail())
12          {
13              std::cin.clear();
14              std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
15              std::cout << "Invalid input, try again.\n";
16              continue;
17          }
18          else
19          {
20              std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
21              return x;
22          }
23      }
24  }
25
26  int main()
27  {
28      int first{getNumber()};
29      std::cout << "The first extraction was " << first << '\n';
30      int second{getNumber()};
```

```
    std::cout << "The second extraction was " << second << '\n';
    return 0;
}
```

Gacrux
October 27, 2019 at 7:56 pm · Reply

Or maybe

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

bool checkInput()
{
    if (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Invalid input, try again.\n";
        return true;
    }
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return false;
}

int getNumber()
{
    int x{};
    do
    {
        std::cout << "Enter an integer: ";
        std::cin >> x;
    }
    while (checkInput());
    return x;
}

int main()
{
    int first{getNumber()};
    std::cout << "The first extraction was " << first << '\n';
    int second{getNumber()};
    std::cout << "The second extraction was " << second << '\n';
    return 0;
}
```

nascardriver
October 28, 2019 at 4:13 am · Reply

Hi!

It depends on how you want your program to be used. The way your code looks, you want to enter the user to enter one integer after the other, with a message being printed in between. Clearing the stream after a successful extraction is fine in that case.
Sometimes you want to allow multiple inputs to be entered at the same time. Either because the user already knows what kind of input your program expects or to allow piped input, eg. from a file. If you don't clear the stream, the user can do this

```
Enter an integer: 10 10
```

```
The first extraction was 10
Enter an integer: The second extraction was 10
```

Granted, the output doesn't look nice because you relied on the user to press enter when prompted the second time, but that way the user doesn't have to wait for your prompts.

Samira Ferdi
August 25, 2019 at 5:41 pm · Reply

Hi, Alex!

I'm confuse the context of 'extracted' in "If the user enters “5a”, 5 will be extracted, converted to an integer, and assigned to variable x. “a\n” will be left in the input stream for the next extraction."

You give the definition of extraction: get user input and put into (assigned to) a variable. So, if I apply this definition, the statement above is like this:
If the user enters "5a", 5 will be take from user, (the user input placed into data buffer) and put into (assigned to) a variable x (extracted), converted to an integer, and assigned to variable x. So, it sounds like twice assignment occur. If what you really mean is twice assignment occur, then your statement is clear for me, but I'm not sure about it.

when type conversion occur? before user input move from input buffer into a variable or after user input assigned to a variable?

Alex
August 27, 2019 at 1:12 pm · Reply

It works something like this:
1) If the data buffer is empty:
1a) The program waits for user input
1b) The user submits the input data
1c) The data gets added to the input buffer
2) Based on the type of the variable, a specific extraction function is called to do the extraction
3) If this is successful, data from the front of the buffer is extracted
4) If this is not successful, the failure flags are set and the variable is zero'd.

If there's any kind of type conversion, it happens inside the extraction function itself, prior to assigning the value to the variable.

Samira Ferdi
August 27, 2019 at 4:21 pm · Reply

Thanks, Alex!

alfonso
August 24, 2019 at 11:46 pm · Reply

I tried to make the final program less redundant (cleaning the input buffer before any usage) but it looks like executing

```
std::cin.ignore ()
```

before using std::cin for extraction at least once, breaks the program. The first double value is ignored and the program waits to reenter the first double value.

```
#include <iostream>
```

```
#include <limits>

void resetInputBuffer () {
    std::cin.clear ();
    std::cin.ignore (std::numeric_limits <std::streamsize>::max (), '\n');
}

double getDouble() {
    double x {};

    while (true) {
        std::cout << "Enter a double value: ";

        // make sure the input buffer is ready
        resetInputBuffer ();
        std::cin >> x;

        if (std::cin.fail ()) {
            continue;
        }

        return x;
    }
}

char getOperator() {
    char op {};

    while (true) {
        std::cout << "Enter one of the following: +, -, *, or /: ";

        // make sure the input buffer is ready
        resetInputBuffer ();
        std::cin >> op;

        if (std::cin.fail ()) {
            continue;
        }

        if (op == '+' || op == '-' || op == '*' || op == '/') {
            return op;
        }
    }
}

void printResult(double x, char op, double y) {
    double result {};

    if (op == '+') {
        result = x + y;
    } else if (op == '-') {
        result = x - y;
    } else if (op == '*') {
        result = x * y;
    } else if (op == '/') {
        if (y != 0.0) {
            result = x / y;
        } else {
            std::cout << "Error: divison by 0!\n";
            return;
        }
    } else {
```

```
64          std::cout << "Error: invalid operator!\n";
65          return;
66      }
67
68      std::cout << x << ' ' << op << ' ' << y << " = " << result << '\n';
69  }
70
71  int main()
72  {
73      double x = getDouble();
74      char op = getOperator();
75      double y = getDouble();
76
77      printResult(x, op, y);
78
79      return 0;
80  }
```

nascardriver
August 25, 2019 at 1:23 am · Reply

- Initialize your variables with brace initializers.
- Clearing the input buffer before using it doesn't make sense, it's already clean. You brush your teeth after eating, not before.

> executing `std::cin.ignore ()` before using std::cin for extraction at least once, breaks the program
You're telling `std::cin.ignore` to ignore everything until it finds a line feed. Since you're calling it before taking any input, it doesn't find a line feed, so it blocks until you enter something.

alfonso
August 26, 2019 at 11:12 pm · Reply

> Initialize your variables with brace initializers.

Ah, those 73, 74, and 75 lines come from (outdated) example program of this page. I just modified the program and I did not look there.

> Clearing the input buffer before using it doesn't make sense, it's already clean. You brush your teeth after eating, not before.

In this short program you can know it is clean. But in large programs where you do not know easily where the input buffer was used last ... I sow this kind of practice in programs - do not rely on other parts of the program for cleaning, or do not presume something is as it should be. First make sure it is all right and then proceed. Generally speaking, but here clearly something went wrong.

> You're telling `std::cin.ignore` to ignore everything until it finds a line feed. Since you're calling it before taking any input, it doesn't find a line feed, so it blocks until you enter something.

Now I see.

alfonso
August 24, 2019 at 10:36 pm · Reply

Error Case 5: The input is valid standalone but may be meaningless in some possible contexts: division by 0. For example x is a valid double 2.8, y is also a valid double 0.0 but (2.8 / 0.0) makes no sense. Ok, maybe this is not really about user input but about managing errors.

Alex
August 27, 2019 at 9:56 am · Reply

I'd consider this a variant of case 1. Much like 'k' isn't a valid mathematical operator, 0 isn't a valid divisor.

Samira Ferdi
August 22, 2019 at 9:51 pm · Reply

Hi, Alex and Nascardriver!

Any unextracted input is left in the input buffer for future extractions. Is there a way to print out all unextracted input that left in the input buffer?

nascardriver
August 23, 2019 at 2:29 am · Reply

I don't think you can access more than 1 character without extracting them.
If you're fine with extracting the characters, you can loop `std::cin.get()` until you find a line feed or end of file.

Haider
August 20, 2019 at 8:52 am · Reply

I have two questions:
What would be the maximal value that "std::cin.ignore()" can ignore? (e.g. you chose 32767.)
Similarly, how many extraction values can the buffer hold?
Why doesn't "std::cin.clear()" have any arguments?

nascardriver
August 20, 2019 at 10:50 am · Reply

```
std::numeric_limits<std::streamsize>::max()
```

disables the count check, ie. an all characters up to `delim` (the second argument) are ignored.

```
std::numeric_limits<std::streamsize>::max() - 1
```

is the maximum value if you want to keep the count check, which doesn't make sense.

> how many extraction values can the buffer hold?
Do you mean how large the buffer can be? That's implementation-defined. There is probably no built-in limit, but you'll run out of memory.

> Why doesn't "std::cin.clear()" have any arguments?
It clears the internal error flags. What would you like to pass? :)

Haider
August 20, 2019 at 11:09 am · Reply

Thank-you.

Benur21
August 5, 2019 at 11:46 am · Reply

"Since the last character the user entered must be a '\n', we can tell std::cin to ignore buffered characters until it finds a newline character (which is removed as well)."

There is also the case where the user copies text that includes multiple new lines from another place, and pastes it to the program. Then this fix will not work.

Samira Ferdi
July 21, 2019 at 5:30 pm · Reply

Hi, Alex and Nascardriver!
I have questions.

first, so, the process of getting user input is like this:
1) user enter the input
2) the user input goes to buffer, and
3) this user input from buffer extract to (move to) a variable?

second, why std::cin by default do not capture whitespace? are there any reasons or uses why this happen?

nascardriver
July 22, 2019 at 5:39 am · Reply

> the process of getting user input is like this
Correct.

> why std::cin by default do not capture whitespace?
Not extracting whitespace allows you to extract the input word by word without having to manually split the string.

```
std::string strName{};
int iAge{};
std::string strFavoriteFood{};

std::cin >> strName >> iAge >> strFavoriteFood;

// tom 12 strawberries
// strName: tom
// iAge: 12
// strFavoriteFood: strawberries
```

If you want to extract an entire line, you can use `std::getline`.

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

// tom 12 strawberries
// str: tom 12 strawberries
```

DEEPAK
May 29, 2019 at 3:55 am · Reply

when complier ask for char input, it treat 'y' and 121 (ASCII of 'y') same. but i dont want to accept 121 only 'y' is allowed what should i made.

nascardriver
May 29, 2019 at 4:35 am · Reply

You can't change it.
I guess you're mixing up the compiler with your running program.
Have a look at this snippet

```
#include <iostream>

int main(void)
{
  char ch{};

  std::cin >> ch;

  std::cout << ch << '\n';

  return 0;
}
```

We could initialize @ch with 121, which is the same as 'y'. But if the user enters 121 when they're asked for the input, the 121 isn't treated as a number, but 3 characters ('1', '2', '1'). Thus, only '1' is extracted and stored in @ch.

DEEPAK
May 30, 2019 at 10:22 am · Reply

thank you so much! its really very helpfull.

Alireza
April 12, 2019 at 3:01 am · Reply

Is this code good to use ?
Are invalid texts handled well ?

```
void calculator()
{
    cout << "Form: X+Y ( + - * / ): " ;

    double numberOne, numberTwo ;

    char theOperator ;

    cin >> numberOne >> theOperator >> numberTwo ;

    if(std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(32767,'\n');
    }

    system("cls") ;

    cout << numberOne << " "<< theOperator << " " << numberTwo << " = " ;

    try{
        switch (theOperator) {
        case '+':
```

```
24              cout << numberOne + numberTwo << endl ; break ;
25          case '-':
26              cout << numberOne - numberTwo << endl ; break ;
27          case '*':
28              cout << numberOne * numberTwo << endl ; break ;
29          case '/':
30              if(numberOne == 0.0 || numberTwo == 0.0) throw (0) ;
31              cout << numberOne / numberTwo << endl ; break ;
32          default:
33              cout << "Unavailable operator"
34                           "\n" ;
35      }
36      }
37      catch(...){
38          cout << "Zero division error,"
39                  " Enter an available value greater than 0"
40                  "\n\n" ;
41          exit(0) ;
42      }
43  }
```

nascardriver
April 12, 2019 at 3:07 am · Reply

* Line 5, 7: Initialize your variables with brace initializers.
* Line 14: Don't pass 32767 to @std::cin.ignore. Pass @std::numeric_limits<std::streamsize>::max().
* Don't use "using namespace". It can lead to name conflicts.
* Don't use @std::system, it won't work on other OSs.
* Don't use @std::exit. It makes control flow harder to understand.
* There's no need for exceptions, they slow down your program.
* 0 / x is legal

You're detecting invalid input, but don't ask the user to correct themselves.

Alireza
April 12, 2019 at 6:30 am · Reply

Thank you so much for making me aware of my mistakes

Alireza
April 12, 2019 at 6:41 am · Reply

[quote]
* Line 5, 7: Initialize your variables with brace initializers.
[/quote]

How should I initialize my variables with brace initializers when I want them to be initialized with @std::cin >> numberOne >> theOperator >> numberTwo ; ???

nascardriver
April 12, 2019 at 6:43 am · Reply

@std::cin doesn't initialize, it assigns. Before the call to @std::cin::operator>>, your variables have undefined values. That can make debugging more difficult and might result in problems when you remove code later.

```
double numberOne{};
double numberTwo{};

char theOperator{};

std::cin >> numberOne >> theOperator >> numberTwo;

// ...
```

Now the program is in a predictable state at every time.

Benur21
August 5, 2019 at 12:58 pm · Reply

That is the same as zero-initializing them, right? Like this:

```
double numberOne = 0;
double numberTwo{0};

char theOperator{''};

std::cin >> numberOne >> theOperator >> numberTwo;
```

nascardriver
August 6, 2019 at 2:43 am · Reply

`0` is an integer, it should be `0.0`.
`''` doesn't exist, it should be `'\0'`.
But yes, empty curly braces default-initialize.

Paulo Filipe
March 25, 2019 at 10:27 am · Reply

With what we know, so far, assuming we're using std::cin to get input from the user, is it possible to detect if the user presses enter with no input, and if the user inserted extraneous input, for example user inputs 10a into an int variable, instead of 10 going to the variable and ignoring the rest of cin, throw a message: "Invalid input". ?

nascardriver
March 26, 2019 at 12:57 am · Reply

```
if (std::cin.get() == '\n')
{
  // User pressed enter. @std::cin.get() extracts the line feed.
  // If you don't care whether or not something has been entered,
  // you can skip to check and @std::cin.ignore after the call to
  // @std::cin.get().
}
```

```
int i{};

std::cin >> i;

```

```
if (std::cin.peek() != '\n')
{
  std::cout << "Invalid input.\n";
}
```

Both snippets without error handling.

« Older Comments 1 2 3