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

BY ALEX ON APRIL 21ST, 2016 | LAST MODIFIED BY ALEX ON JUNE 7TH, 2018

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". 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 can be extracted from the input buffer. Any unextracted input is left in the input buffer for future extractions. For example:

```
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:

```
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:

```cpp
#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 user.
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:

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

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:

```cpp
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 '\n' character is removed
    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
            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 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: 0
```

In the above case, std::cin goes immediately into “failure mode”, so it does not assign a value to variable x. Consequently, x is left with the initialized value of 0. Additional inputs are skipped, leaving y with the initialized value of 0 as well. We can handle this kind of error in the same way as a failed extraction.

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 about validat
ing that
            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 extraction h
ere

        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() guarantees op is valid in this particular program
        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