C++ Formatted Input Made Easy

cth027, 12 Apr 2015 Zlib

4.33 (2 votes)

Rate: vote 1vote 2vote 3vote 4vote 5

A small utility class to ease C++ stream input with predefined and controlled format

Download source - 6.8 KB

Introduction

Processing formatted input such as comma separated lists, ISBN numbers, or phone numbers is a very common task. Unfortunately, there is no easy and consistent way to control fixed formatting characters in C++ input streams.

I present you a C++ class that makes controlled formatted input as easy as C's standard `scanf()`/`fscanf()`, and without the complexity of more powerful solutions such as for example regular expressions or parser generators.

Background

Formatted input is defined by predefined fixed characters that make the data recognizable and help to separate it into meaningful sub-parts. For example:

- a 2D point: `(12,24)` where `x` is `12` and `y` is `24`
- an ISBN number: `0-201-70073-5` where `0` is a linguistic group, `201` is a registrant, and `5` a checksum
- other custom formats, potentially with multiple control characters in sequence. For example, a set of 2 points: `{(12,24),(6,12)}`

In standard C, you could easily manage such requirements with `scanf()`:

Hide Copy Code

```
int x,y;
fscanf(file, "(%d,%d)", &x, &y);
```

In standard C++, you could also use `scanf()`. But you'd have to mix traditional C file functions with modern, elegant, object oriented C++ streams.

The other C++ alternative requires the wheel to be reinvented each time:

Hide Copy Code

```
int x,y;
char d1,d2,d3;                          // dummy data for separators
if ( (cin >> d1 >> x >> d2 >> y >> d3)     // read all data
      && d1=='(' && d3==')' && d2==',')    // verify that separators are correct
```

```
{
    ... //process data
}
else cout << "Bad input !"<<endl;
```

Of course, this works. But it's tedious, and doesn't ensure clean error processing: if it fails, you have no clue which part of the input could be read, and which failed, unless you add a lot of additional `if`s.

This is why I developed `mandatory_input`, a small, powerful but astonishingly simple utility class, with the following design objectives:

- work on any kind of input streams (cin, file streams, string streams, ...)
- show expected formatting characters directly in the input operations
- process formatting errors consistently with other input errors
- provide flexible and easy error processing

Using the Code

You have to include the header *mandatory_input.h* wherever you use this class. You also have to compile*mandatory_input.cpp* and link it with the other files of your project.

Controlling Input

To require a specific character in a stream input, you just have to extract the stream into a `mandatory_input`object. The class is designed to be used as temporary object:

Hide Copy Code

```
int x,y;
cin >> x >> mandatory_input(",")>>y;   // read two comma separated int
```

You can specify several control characters at once: each character must be read in the right order for the input to succeed:

Hide Copy Code

```
cin >> y >> mandatory_input( ")},");
```

White spaces in the control characters indicate that the input can contain additional whitespace before the next control character:

Hide Copy Code

```
cin >> y >> mandatory_input( ")},");     // No space:  "12)}," is valid but "12 ) }," fails
cin >> y >> mandatory_input( " ) } , "); // Space:  "12)}," is valid as well as "12 )   },"
```

Processing Errors

The error processing is consistent with the usual stream behaviour: if a value cannot be read because of invalid input, the failbit of the stream is set.

If the stream is configured for exceptions on failed input, `mandatory_input` will raise an exception`mandatory_input::failure` whenever control characters do not match:

Hide Copy Code

```
if ( !(cin >> x >>mandatory_input(",") ) // traditional processing of failbit is supported
{ ... }


                                    // standard stream exception process is also supported
cin.exceptions(std::istream::failbit | std::istream::badbit);
try {
    cin >> x >>mandatory_input(",") ;
} catch (mandatory_input::failure& e)    // a specific formatting exception can caught.
{ ...}
```

If you don't want to care about details of exceptions, you can choose to catch only the standard`istream::failure`. This will also take care of `mandatory_input::failure`.

The following functions are useful for advanced error processing:

Hide Copy Code

```
char mandatory_input::getlast();      // last successfully read non-white formatting character
bool mandatory_input::error();        // indicates that there was a formatting error
char mandatory_input::getexpected();  // expected formatting char that was not obtained and
                                   //   lead to the error
char mandatory_input::getread_error();// read char that didn't match the expected formatting
                                   // char
char reset();                       // returns last successful formatting char and resets
                                   //   the error state
```

Note that the input character that caused the failure will be the next character read after the stream's error state is reset.

Examples

Here is a small example to read an ISBN:

Hide Copy Code

```
istringstream isbn1("0-201-70073-5");


if (isbn1 >> lg >> mandatory_input("-") >> rg >> mandatory_input("-")
                                          >> bk >> mandatory_input("-") >> chk)
    cout << "ISBN1 ok:" << lg << "-" << rg <<
    "-" << bk << "-" << chk << endl;
else
```

```
    cout << "ISBN1 failed: "<< mandatory_input::getexpected() << " was expected, but "
          << mandatory_input::getread_error() << " was read !\n";
```

Here is a small example to read a list of pairs, such as `(1,2),(3,4),(5,6)`. The input shall end as soon as there is a missing comma behind a pair. However, other formatting errors shall not be allowed:

Hide Copy Code

```
int x = 0, y = 0;

// Reading loop trying to read (x,y),
while ((is >> mandatory_input(" ( ") >> x >>
mandatory_input(" , ") >> y >> mandatory_input(" ) , ") ))
{
    cout << "Pair " << x << " " << y << "\n";
}

// if only the comma separator between pairs is missing
if (mandatory_input::getlast() == ')')
    cout << "final pair " << x << " "
    << y << " !" << endl;  // use the successful input

// but if another error was detected, explain the problem
else if (mandatory_input::error()) {
       cout << " last succesfully controlled char:" << mandatory_input::getlast() << endl;
       cout << mandatory_input::getexpected() << " was expected, but "
            << mandatory_input::getread_error() << " was read !\n";
}
if (!is.eof())
    is.clear();  // resume reading is possible
```

Points of Interest

This class is designed for instantiating temporary objects. This sheds light on the lifetime of temporaries. The C++ standard, section 12.2 point 3 explains:

"*Temporary objects are destroyed as the last step in evaluating the full-expression that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception.*"

This means that if a temporary `mandatory_input` is instantiated in an extraction statement, the argument of the constructor will live at least up to the end of the extraction. This is why no copy of it is made, saving some overhead.

The problem with a temporary design is the error processing. For ease of use, I used static variables/functions to hold the error state. This approach has its limitations: it won't work for multithreaded use. An easy way out could be to provide an additional error processing object to the constructor.

Another point of interest is the power of object oriented design of streams. It's really impressive to see how easily the creation of a new class with the proper operator overloading can increase the level of abstraction of the code.

A last point to mention is the performance aspect. Some informal benchmarks show that extracting with`mandatory_input` is of roughly the same performance than extracting with uncontrolled placeholder characters. So this class is as consistent with stream extraction as one could dream. Surprisingly however, `scanf()` performs the same task almost 5 times faster! The regex alternative is 3 times slower than stream extraction. So regular expressions should really be kept for problems requiring more complex parsing.

Here are the details:

Hide Copy Code

```
BENCHMARKING:
getline+regex          8843 ms for 1024000 elements
stream extract 1        375 ms for 1024000 elements
stream extract 4       3250 ms for 1024000 elements
stream ext.mandat      3093 ms for 1024000 elements
C file scanf           625 ms for 1024000 elements
```

History

- 14/7/2014: First version of the code, inspired by an idea found on StackOverflow.
- 10/4/2015: First version of this tip

License

About the Author

cth027

Technical Lead
France