Doc No: SC22/WG21/N2327
J16/07-0187
Date: 2007-06-22
Project: JTC1.22.32
Reply to: Cosmin Truța
IBM Canada
cosmint@ca.ibm.com

Inconsistencies in IOStreams Numeric Extraction

Abstract

The standard is inconsistent with respect to the behavior of operations in which incorrectly-formed numeric data is extracted from an istream. Moreover, inconsistencies that are inherent to `scanf` also affect the IOStreams. In this paper we outline several consistency and usability concerns, we discuss two possible resolutions, and we show the advantages brought by changing the underlying string parsing from `scanf` to the `strto*` family of functions.

1. Introduction

A C programmer who uses `scanf` can easily find out when the input items passed to `scanf` receive an output value, by checking the return code of `scanf`. This is especially important when the input items are previously uninitialized, which happens often (e.g. by calling `scanf` right after declaring the input items as automatic variables). At the same time, a C programmer who uses an input string buffer and calls a function like `strtol` will always get a meaningful output, either in case of success, or in case of failure.

On the other hand, a C++ programmer who uses IOStreams cannot reliably expect to know, at a high level, when the value of the extracted operand is the result of a successful extraction operation, or the value prior to an unsuccessful operation. This is indicated most of the times, but not always, by the status of `ios_base::failbit`.

Moreover, situations like the following constitute a frequent source of confusion to the people who are new to the C++ IOStreams:

```
int i;
cin >> i;              // enter incorrect input
cout << i << endl;     // obtain surprise output
```

It is arguably less intuitive when an overloaded operator (as opposed to a regular function) changes its operands in certain events, but leaves them intact in others. In comparison, it is usually obvious when in-out parameters in regular functions like `scanf` might be modified only partially.

This problem is further aggravated by the printed literature. Unfortunately, very few C++ books properly teach their readers to check the correctness of I/O operations. In many books, `ios_base` member functions like `good()` or `fail()` are merely listed in an

appendix; in many others, such things are not even mentioned at all. Last but not least, it is difficult to spot the failure to check `failbit` at debug time, because of the non-deterministic behavior that occurs after the extraction to an uninitialized variable fails.

Another motivation behind the changes proposed in this article is the inconsistent signaling and handling of overflows inside `scanf`. For this reason, we propose to redefine `num_get` in terms of the `strto*` family of functions, and to establish an intuitive and consistent behavior in case of overflows, as well as other kinds of mismatch.

The rest of this article is organized as follows. In Sections 2 and 3 we discuss the identified inconsistencies in detail, and in Section 4 we outline our proposed resolution. In the event of a possible rejection of our resolution, we further propose an alternate resolution in Section 5.

2. Handling the incorrect placement of thousands separators

From the ISO C++ Standard (Edition 2003), Section 22.2.2.1.2, paragraphs 11 and 12, it is implied that the value read from a stream must be stored even if the placement of thousands separators does not conform to the `grouping()` specification from the `numpunct` facet. On the other hand, incorrectly-placed thousands separators are flagged as an extraction failure, by the means of `failbit`. A consistent strategy, in which any kind of extraction failure leaves the input item intact, is conceptually cleaner, is able to avoid corner-case traps, and is also more understandable from the programmer's point of view.

Here is a quote from Stroustrup [1] (Section D.4.2.3, pg. 897): *"If a value of the desired type could not be read,* `failbit` *is set in* `r`. [...] *An input operator will use* `r` *to determine how to set the state of its stream. If no error was encountered, the value read is assigned through* `v`*; otherwise,* `v` *is left unchanged."*

This statement implies that `rdstate()` alone is sufficient to determine whether an extracted value is to be assigned to the input item `val` passed to `do_get`. However, this is in disagreement with the current C++ Standard. The above-mentioned assumption is true in all cases, except when there are mismatches in digit grouping. In the latter case, the parsed value is assigned to `val`, and, at the same time, `err` is assigned to `ios_base::failbit` (essentially "lying" about the success of the operation). Is this intentional? The current behavior raises both consistency and usability concerns.

Although digit grouping is outside the scope of `scanf` (on which the virtual methods of `num_get` are based), handling of grouping should be consistent with the overall behavior of `scanf`. The specification of `scanf` makes a distinction between input failures and matching failures, and yet both kinds of failures have no effect on the input items passed to `scanf`. A mismatch in digit grouping logically falls in the category of matching failures, and it would be more consistent, and less surprising to the programmer, to leave the input item intact whenever a failure is being signaled. Moreover, a counter-intuitive behavior is derived from the fact that, when `failbit` is set, subsequent extraction operations are no-ops until `failbit` is explicitly cleared. Assuming that there is no explicit handling of `rdstate()` (as in `cin>>i>>j`), it is counter-intuitive to be able to extract an integer with mismatched digit grouping, but to be unable to extract another, properly-formatted integer that immediately follows.

Last but not least, the current behavior is not only confusing to the casual reader, but it has also been confusing to some book authors. Besides Stroustrup's book [1], other books (e.g. Langer and Kreft [2]) are describing the same mistaken assumption. The vast majority of C++ books do not even mention digit grouping at all. (To the author's best knowledge, no book actually describes this behavior correctly.) Although books are not to be used instead of the standard reference, the people who learn about C++ and iostreams by reading books, as well as the people who are generally familiar to `scanf`, are likely to misinterpret the standard.

3. Handling overflows

In 1998, Nathan Myers [3] opened a Standard Library Issue concerning the lack of an overflow indicator in `num_get`. Programmers have to rely on the value of `errno`, which is an implied side effect of `scanf`. Besides the well-known disadvantages brought by the use a global state such as `errno`, there are a few additional concerns to be raised.

3.1. Overflows in extraction of narrow integers

The behavior of `scanf` in case of overflow is inconsistent across integer types. The conversion to integer types that are narrower than `long` is defined in terms of `strtol` and `strtoul` in the ISO C Standard, but no additional processing occurs when narrowing the result of `strtol` to `int` or `short`. (The comment also applies to `strtoul` and the unsigned integer types, but they are not discussed here, for brevity.) For this reason, while the conversion to `long` always results in `LONG_MIN` or `LONG_MAX` in case of overflow, the conversion to short or int is implementation-defined. In the following example, we assume a machine having 16-bit `short`, 32-bit `int` and 64-bit `long`:

Input string	Integer conversion in `scanf`			Overflow signaled?
	`"%hd"` (`short`)	`"%d"` (`int`)	`"%ld"` (`long`)	
"100000" (= 10^5)	-31072	100000	100000	No
"10000000000" (= 10^{10})	-7168	1410065408	10000000000	No
"1000…000" (≥ 10^{20})	`SHRT_MAX`	`INT_MAX`	`LONG_MAX`	Yes

Another event surprising to unaware programmers may also occur when they switch between compilation modes that use 32-bit `long` and 64-bit `long` (which is typical nowadays): the result of conversion to `int` changes when the input string falls between the 32-bit bracket and the 64-bit bracket, even though `int` is 32-bit wide in both cases.

3.2. Overflows in extraction of unsigned integers

The second concern is given by the inability of `strtoul/strtoull` (and, implicitly, the inability of `scanf`) to signal a negative overflow while converting to unsigned integers.

Negative numbers cannot be directly represented using unsigned integer types, and, for this reason, they should conceptually be outside the normal range of unsigned integers. However, there is no direct way of diagnosing negative overflows, and the programmer has to search for the minus sign in the input string when he or she needs such a diagnostic. It is highly useful to automatically convert a negative signed integer to its unsigned counterpart, but, at the same time, it is desirable to know when a negative overflow occurs.

The converse of this problem does not exist in the signed case: `strtol` and `strtoll` correctly signal the situation when the input string contains a positive number that is larger than `L[L]ONG_MAX` but smaller than `UL[L]ONG_MAX`, even though this number fits in the same machine word, in unsigned form.

3.3. Overflows in extraction of non-alphabetic booleans

Although the extraction of booleans is outside the scope of `scanf`, the numeric (i.e. non-alphabetic) extraction might be seen as the extraction of an integer within the bounds 0 and 1. Let `val` be a variable that stores a `long` value. Conversion of `val` to `bool` satisfies the following constraint: `(bool)val == (val != 0)`.

However, `num_get` extraction of `bool` is explicitly required to leave the input item intact and indicate a failure. This is contrary to the usual expectation, in which any value different than 0 and 1 would be converted (narrowed) to true and accompanied by an overflow indication.

4. Resolution

The proposed resolution consists in introducing a uniform behavior: there will always be an extracted value which, in case of failure, will have a meaning as close as possible to the meaning of the input string.

Specifically, this resolution consists of completely replacing `scanf` with the `strto*` family, defining new strategies to be applied when narrowing takes place (from `long` to `int` or `short`, and from `double` to `float`), and defining similar strategies for the cases that are currently outside the scope of `scanf`:

- When narrowing the result of `strtol` to a signed integer `SInt`, conversion overflows will be extracted as `numeric_limits<SInt>.max()` or `numeric_limits<SInt>.min()` (whichever is more appropriate). These overflows will be flagged as such via `failbit`.
- When narrowing the result of `strtoul` to an unsigned integer `UInt`, conversion overflows will be extracted as `numeric_limits<UInt>.max()`, and flagged as such via `failbit`. Negative inputs will also be flagged (even though they are not originally flagged by `strtoul`), but the result of `strtoul` will not be altered (e.g. `"-2"` will still be extracted as `static_cast<UInt>(-2)`).

- When narrowing the result of `strtol` to non-alphabetic `bool`, overflows (i.e. all values except 0 and 1) will be extracted as `true`, and will be flagged as such via `failbit`.
- When the input string cannot be parsed to alphabetic `bool`, it will be extracted as `false`, and flagged as such via `failbit`. (This strategy is in sync with the behavior of the `strto*` functions, when string parsing fails.)
- When narrowing the result of `strtod` to `float`, overflows will be extracted as `±FLT_MAX` and underflows as `±0.0` (whichever is more appropriate), and will be flagged as such via `failbit`.
- When thousand separators are incorrectly placed, the extracted value will remain intact, but the mismatch will be flagged via `failbit`. (This strategy is the same as in the current Standard.)

This resolution is, admittedly, a conceptual departure from the current Standard. In case this is deemed too radical, an alternate resolution is also proposed.

5. Alternate resolution

Another possibility is to introduce a well-determined relationship between the success of extraction and the status of `failbit`. This incurs a lesser amount of change in the existing Standard, although it still leaves the inability to extract potentially-useful values (e.g. a boolean out of a numeric string different than `"0"` or `"1"`, or an integer out of a string containing mismatched thousands separators). The overflow problems inherent to `scanf` still remain in place, as well as the confusion that is typical among novice C++ programmers, regarding the unexpected "extracted" values in case of failure.

Specifically, this consists in discarding the result of Stage 2 if digit group checking fails. A complete description of this resolution is provided in the C++ Standard Library Issue no. 662 [4].

Bibliography

[1] Bjarne Stroustrup. *"The C++ Programming Language (Special Edition)."* Addison-Wesley, 2000.

[2] Angelika Langer, Klaus Kreft. *"Standard C++ IOStreams and Locales."* Addison-Wesley, 1999.

[3] Nathan Myers. Num_get overflow result.
C++ Standard Library Active Issues List, Item 23, submitted on 1998-Aug-06.
http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#23

[4] Cosmin Truța. Inconsistent handling of incorrectly-placed thousands separators.
C++ Standard Library Active Issues List, Item 662, submitted on 2007-Apr-05.
http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#662