Proposal for C++ replacing unsigned integer types with signed integers in the Standard Library

Christopher Di Bella

2017-03-24

Abstract

This document proposes to replace all unsigned integer types with signed integers within the confines of the Standard Library containers, particularly for their inclusion in STL2, and any associated Technical Specifications. The proposal discusses both sides of the argument, but firmly favours the replacement.

Motivation

Only `vector` is mentioned for brevity. This motivation should be generalised to all container requirements, and should open a discussion point for other areas of the Standard Library.

The Standard Library requires that containers return `size_type` for many operations, including `size`, `max_size`, and `capacity`. Similarly, `operator[]`, `at`, `assign`, `reserve`, and `resize` all require a `size_type` parameter. `size_type` is required to be an unsigned integer, as per **??**.

CppCoreGuidelines[4] §ES.101, §ES.102 recommend that programmers reserve unsigned types for bit manipulation and signed types for arithmeitc. §ES.102 states that "most arithmetic is assumed to be signed; `x-y` yields a negative number when `y>x` except in the rare cases where you really want modulo arithmetic. Unsigned arithmetic can yield surprising results if you are not expecting it. This is even more true for mixed signed and unsigned arithmetic." Further arguments for preferring signed types have been made by Stroustrup, Meyers, Sutter, Carruth, and Lavavej at a panel in 2013[3].

[4]§ES.100 advises that signed and unsigned integers are not mixed in operations to avoid incorrect results. A programmer following §ES.101 and §ES.102 reaches a contradiction, since they are encouraged to use signed integers for arithmetic operations, but must now mix signed and unsigned integers for the following code:

```
auto v = vector<int>{};
for (auto i = 0; i < v.size(); ++i)
   cout << i << ": " << v[i] << '\n';
```

Here, we have a comparison between a signed integer and an unsigned integer. Although the comparison is well-formed (albeit ill-advised), the increment operation is undefined when `i == numeric_limits<int>::max()`. Such an evaluation is highly undesirable.

Proposal

This proposal suggests:

— All container concepts shall require `difference_type` return types where they currently require `size_type`.

— Similarly for function parameters.

— Deprecate all functions that either return or use `size_type` in the current Standard Library. As desirable as erasing these functions is, the author acknowledges that there may be significant bodies of code that jump through hoops to accommodate for this.

Discussion

The following subsections detail numerous arguments against the change that the author has heard or read in the past twelve months.

Unsigned types assert that an integer is non-negative.

Consider this code:

```
template <typename T, typename A>
vector<T, A>::reserve(unsigned int i);

int main()
{
   auto v = vector<int>{};
   v.reserve(-1); // logic error: allocates 2^32 − 1
}
```

Due to an implicit narrowing conversion, the sign is lost, and we allocate $2^{32} - 1$ times as many objects.

No formal assertion is made to stop this, in part because it is impossible for us to detect if we're allocating -1 bytes of memory or $2^{2^{sizeof(int)}} - 1$ bytes of memory without extra information.

Let's add that information

Sure, we could introduce

```
enum class intentionally_big { yes, no };
v.reserve(-1, intentionally_big::no);
```

This looks error prone and is strongly recommended against.

We're not going to run out of memory any time soon. Let's just leave it be.

See Y2K bug. We don't want to be responsible for Y2K++.

This is an issue to do with implicit narrowing conversion. We should outlaw that and leave the Standard Library as it is.

Consider this code:

```
// insert better example here when possible
for (auto i = 0; i < v.size(); ++i) {
   cout << (i - v.size()) << " positions to go\n";
}
```

Under the current model, which expression should be explicitly cast? We'll also need to do it twice, and guarantee that we are casting the correct way each time. We've also lost some information: if `v.size() >=` $2^{2^{sizeof(int)}}$, then converting the size will yield a negative and meaningless number for our computation. Similarly, if we evaluate `static_cast<unsigned int>(i)`, we risk losing information in `i`. We will also then need to perform a conversion to an `int` over the whole subtraction to output the information we are truly after.

Preventing non-promoting implicit conversions doesn't fix the issue here, and has the potential to create issues for readability and design.

This is a contract design error that the user should handle.

No, it isn't. We are calling a function. It is the callee's responsibility to make sure that its input is correct, not the caller, who may not know. If a bug is identified in a library function, it is the responsibility of the author to fix the bug; the user is only responsible for identifying and reporting the bug.

People should write better code

Absolutely correct! We should be encouraging people to write better code, and one way that we can do this is to provide them with a toolset that isn't dangerous.

But this is C++. People that aren't serious about the low-level stuff should go back to Language X.

Recall the original evolution rules for designing C++, taken from [2]:

— C++'s evolution must be driven by real problems

— Don't get involved in a sterile quest for perfection

— C++ must be useful *now*

— Every feature must have a reasonably obvious implementation

— Always provide a transition path

— C++ is a language, not a complete system

— Provide comprehensive support for each supported style

— Don't try to force people

This proposal is driven by a real problem, and has a reasonably obvious implementation. C++ is not a low-level language for the wizards, and programmers should be able to write clear, readable code that doesn't involve jumping through hoops to do simple arithmetic with Standard containers.

It really comes down to how you use it. Just because someone owns a `deadly_tool` doesn't mean that they go around killing people.

This is a strawman argument. Common decency and laws prevent exactly this, and in our case, the International Standard is the Law. This proposal offers a way to restrict the misuse of types, by preventing the most commonly used concept, `SignedInteger`, with a type that has no business in arithmetic.

But I really need to reserve those extra objects! What can I do?

You can write your own non-`Container`-compliant data structure. If you desparately need $2^{2^{sizeof(std::size\_t)}}$ objects, you might very soon need $2^{2^{sizeof(std::size\_t)+1}}$ objects, which you can't get on a $2^{sizeof(std::size\_t)}$-bit CPU anyway (by conventional means)[1].

Acknowledgements

This document was inspired by [1], and the author would like to thank Eric Niebler, Casey Carter, Sascha Atrops, GitHub user dhaumann, and GitHub user poelmanc for their participation in the discussion online.

Andrei Alexandrescu, Chandler Carruth, Stephan T. Lavavej, Scott Meyers, Bjarne Stroustrup, and Herb Sutter all provide further motivation for replacing unsigned types with their signed counterparts in [3].

Both Margret Steele and Arien Judge have provided talking points in the discussion section of the proposal.

The LaTeXsource for this document was taken from an older revision of the Ragnes TS.

Bibliography

[1] Eric Niebler, Casey Carter, Sascha Atrops, Christopher Di Bella, dhaumann, and poelmanc. Kill unsigned integers throughout stl2. `https://github.com/ericniebler/stl2/issues/182`. Accessed: 2017-03-23.

[2] Bjarne Stroustrup. *The Design and Evolution of C++*. Pearson Education, Indianapolis, 1994.

[3] Bjarne Stroustrup, Andrei Alexandrescu, Herb Sutter, Scott Meyers, Chandler Carruth, Sean Parent, Michael Wong, and Stephan T Lavavej. Interactive panel: Ask us anything. `https://www.youtube.com/watch?v=Puio5dly9N8#t=42m40s`. Accessed: 2017-03-23. Hosted by: Paulo Portela.

[4] Bjarne Stroustrup and Herb Sutter. Cppcoreguidelines. `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`. Accessed: 2017-03-23.