Doc. No. P1428R0
Date: 2018-01-18
Programming Language C++
Audience LEWG and EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

Subscripts and sizes should be signed

Bjarne Stroustrup

The problem

I write this in support of status quo in the WP: **std::span**'s index and size types are **signed**.

Usually, there is no need to write papers in favor of status quo, but in San Diego, the LEWG decided

> Option 4: add std::ssize() alongside std::size(), change span::size() to unsigned, no additional members. (P1227, reduced)
>
> 7/18/12/5/6
>
> …
>
> VV: Does option 4 have consensus?
> TW: 25 to 11--yes.

As stated, changes the type of sizes but not subscripts, but apparently some believed that this also changed the type of subscripts. A vote to just change the type of **span::size()** failed, but not by much:

> Option 2: span::size() is unsigned (P1089)
> 14/11/6/6/8

At an evening session, many prominent committee members spoke against that change (see minutes); I was among them and I (still) think that the decision to move to **unsigned** sizes for **span** is wrong and shortsighted. Eight "strongly against" votes almost guarantee that the issue will re-emerge, so I am putting my thoughts on record.

Please note that my reasons are in support of status quo in the current WP, not a suggestion to change anything in the C++20 time-frame.

I see two primary arguments for using **unsigned** subscripts and sizes:

- That's what we have been doing since the adoption of the STL in 1996.
- Sizes cannot be negative

and two primary reasons for using signed subscripts and sizes:

- Signed values are the result of most integer calculations
- In C and C++, **unsigned** does not model natural numbers

I will dig into the arguments and consider alternatives, but my conclusion stands:

> *Use signed subscripts and sizes for **span** as it was deliberately designed to do.*
>
> *The original use of **unsigned** for the STL was a bad mistake and should be corrected (eventually).*

Why we have unsigned subscripts in the STL

As far as I remember (the STL is 25 years old so my memory may not be completely accurate) three reasons were given for the STL using **unsigned** types for subscripts

- (As opposed to pointer subscripts) **vector** subscripts can't be negative, so **unsigned** is obviously the right type.
- We get one more bit to play with so we can get larger **vector**s; this is important on machines with 16-bit address spaces.
- Range checking needs only one check (no need to check for less than 0).

I have heard such rationales many times over the years, but

- C/C++'s **unsigned** is a very odd set of types. They do not model natural numbers. In particular, they have modular arithmetic and conversions to/from signed **int**s that can be very surprising. Beware of any argument using the word "obvious".
- Even the first version of the STL that I have tracked down specified that even though **vector**'s **max_size()** was of the unsigned **X::size_type** it was specified as the largest positive value of the **vector**'s signed **difference_type** or we could construct **vector**s with elements beyond the accessible range. So much for that extra bit (extra range).
- Checking signed **0<=x && x<max** (often evaluated as **(0<=x) & (x<max)** to avoid branching) is not slower than unsigned **x<max** on a modern computer. The number of loads in the two cases are identical and unless the instruction pipeline and the arithmetic units are saturated, the speed will be identical. Please check; I couldn't check all hardware/compiler combinations.

Basically, we were wrong on all three counts. The questions then become:

- Does it matter?
- Can we do anything about it?
- Is **span** a good place to start?

My suggested answers are

- Yes
- Yes, but only very carefully
- Yes

Problems with unsigned

Mixing **signed** and **unsigned** numbers is a common source of confusion and bugs. Most coding guidelines recommend against mixing them. However, there are two main sources of **unsigned** values getting mixed with **signed** ones:

- People trying to ensure that only nonnegative numbers are passed where negative values don't make sense
- Loop variables and sizes

The use of unsigned for subscripts and sizes is creates a need for mixed **signed** and **unsigned** arithmetic and comparisons. Thus, it forces violations of sane coding guidelines (or amazing workarounds) and sets a bad example, that are enthusiastically followed by programmers unacquainted with the subtleties of **unsigned**.

Consider

```
unsigned area(unsigned x, unsigned y)  // calculate area
{
        return x*y;
}
```

This appears to makes sense; after all lengths and areas can't be negative. However, this definition doesn't prevent **area(-2,3)**. Naturally, in real code the negative value is unlikely to be a literal (that a compiler could warn against. Instead

```
auto a = area(height1-height2, length1-length2);
```

If **height1<height2**, we have a reasonably realistic example.

The problems with **mixing** signed and **unsigned** values in loops is the focus of the rest of this paper. Fundamentally, an **unsigned** is not just a natural number (a nonnegative integer); it is a nonnegative number with modular arithmetic, and that can bite.

Please note that EWG has voted to review and possibly fix mixed **signed**/**unsigned** comparisons to give mathematically correct answers (e.g., **-1<2u** really should be **true**). This is another example of work to escape the mistakes of the late 20th century.

Problems with unsigned sizes

Consider

```
vector<int> v(-2);
```

This is (of course a range error), but why? The reason is that the **vector** constructor takes an **unsigned** so **-2** is interpreted as a very large (positive) number. That error is caught by a run-time check. The use of **unsigned** did not eliminate the need for that check.

We are not always this lucky:

```
unsigned char x = -200;
vector<int> v (x);
```

This executes correctly because **x** is the valid subscript **56**. Somebody might even have wanted that to work; who knows? However, I suspect that negative sizes are almost invariable bugs, often subtle bugs.

The major problem is that **unsigned** sizes yield mixed **signed**/**unsigned** expressions. Consider a simple naive loop:

```
for (int i = 0; i<v.size(); ++i) v[i]=7;
```

Some (but not all) compilers warn that the **i<v.size()** comparison mixes signed and unsigned and is therefore suspect. There is hardly ever a real problem, so those warnings are annoying and confuse novices. "Obviously," I should have written:

```
for (vector<int>::size_type i = 0; i<v.size(); ++i) v[i]=7;
```

Had **v.size()** been **signed**, the loop as written would have been perfectly fine.

More about loops below.

We sometimes (often?) do arithmetic with sizes; notably we subtract sizes to find differences. For example:

```
unsigned u1 = -2;
unsigned u2 = -4;

cout << is_signed<decltype(u1-u2)>::value << " " << u1-u2 << "\n";
cout << is_signed<decltype(u2-u1)>::value << " " << u2-u1 << "\n";
```

Now, we are all experts here and immediately spot the problem (right?), but the output of the second line could cause confusion.

Problems with unsigned subscripts

Consider

```
vector<int> v(100);
auto x = v[-2];
```

This is (of course a range error), but why? It is not because **v** is subscripted by the negative integer **-2**. The subscript to **vector::operator[]** is an unsigned value so that's not possible. Instead, **-2** is the valid subscript **4294967294** which just happens to be too large for that **vector**. It's a run-time error (subscript too large). Compilers should warn, but since there is no type error, not every compiler does. We are not always this lucky:

```
unsigned char x = -200;
auto c = v[x];
```

This executes correctly because **x** is the valid subscript **56**. In a real program were the value of **x** was obtained in a slightly more indirect manner I suspect this would be a surprising result – a logic error.

Consider a simple naive loop:

```
for (int i = 0; i<v.size(); ++i) v[i]=7;
```

Some (but not all) compilers warn that the **i<v.size()** comparison mixes **signed** and **unsigned** and is therefore suspect. There is hardly ever a problem, so those warnings are annoying and confuse novices. "Obviously," I should have written

```
for (vector<int>::size_type i = 0; i<v.size(); ++i) v[i]=7;
```

but that's verbose and non-obvious to anyone but an STL expert. It is also a maintenance hazard: why should I have to mention the element type of the vector to write a loop?

```
for (vector<decltype(v[0])>::size_type i = 0; i<v.size(); ++i) v[i]=7;
```

Anyone?

Yes, we have algorithms and range-**for**, but people still write lots of loops.

The warnings are not completely misguided; here is an infinite loop:

```
for (unsigned char i = 0; i!=v.size(); ++i) v[i] = 7;
```

We don't often see **char** or **short** loop variables, though.

Here is an example that is occasionally seen in the wild:

```
for (size_t i = n-1; i >= 0; --i) { /*  ...  */ }
```

Obviously an **unsigned** is always larger than zero.

The problem is that **unsigned** is not just a natural number (a nonnegative integer); it is a nonnegative number with modular arithmetic, and that can bite.

Consider calculating a starting point from other subscripts

```
for (size_t pos = max(start,pos-length); i<last; ++i) …
```

(if the **unsigned length** is larger than the **unsigned pos**, **pos** goes HUGE) or calculating an end point from other subscripts

```
for (size_t i = 0; i<last; ++i)
        for (size_t j = 0; i<i-j; ++i) …       // near infinite loop
```

Using subscripts in the loop body is common and error-prone for **unsigned** subscripts

```
for (size_t i = 0; i<last; ++i)
        for (size_t j = 0; i<last; ++i) v[i-j] = 7;      // huge subscript
```

I have heard claims that using **unsigned** loop variables leads to less good code than **signed** ones, but I have not been able to find evidence for that in modern compilers.

Essentially all of these examples, could happen with **unsigned** subscripts and **signed** sizes instead: the root problem is mixing **signed** and **unsigned** values.

Span

Why does **std::span** have **signed** subscripts and sizes? The designers of **span** (originally **gsl::span**) had several aims:

- The primary intended use for **span** was as a replacement for {pointer,offset} pairs and pointer arithmetic (incl. subscripting) is signed. It seemed unwise to introduce potential conversion problems related to **signed**/**unsigned** differences.
- Modular arithmetic can cause surprising behavior.
- Modular arithmetic makes "overflow" well-defined behavior and thus inhibits error detection and handling.
- **span** is not a container, so the analogy to **vector** is not compelling; on the other hand, **span** is closely related to pointers (some versions of the idea have been called "fat pointers"). In other words, a span is at a different (lower) level of abstraction that vector and other containers.
- This was an opportunity to "do it right" as opposed to adding another example of the problem.
- This was an opportunity to start an effort to convert the STL away from its mistaken use of **unsigned** for subscripts.

Unsigned sizes

Unfortunately, **sizeof** yields an **unsigned** (and it would be hard to change that), but we don't have to follow that for all types with something to do with sizes.

Compatibility

We have had **unsigned** container sizes for 25+ years. We have had people use **unsigned** to represent non-negative values for longer than that. Changing that is going to be hard. Having a type that differs in its use of **signed** sizes and subscripts is going to be a bother for people who tries to use **span** exactly as a container (but if they do, they are setting themselves up for other conceptional problems) or who are trying to write generic code that (somehow) is sensitive to the **signed**/**unsigned** distinction. On the other hand, the current span saves them from traditional **signed**/**unsigned** problems. So, there is a tradeoff; I think the tradeoff favors the correct (**signed**) solution.

A "unicorn type"

At the evening session, Chandler Carruth suggested we could define a "unicorn type" that simply did the right thing in combination with both **signed** and **unsigned** types. I like that idea. I wrote a primitive (**uni**) type to try out the idea, but I'm not proposing it because I consider my solution ugly and incomplete, I cannot be sure that it really is a drop-in alternative to the current uses of **unsigned**, and I would prefer not to use subtle types for really basic and frequent operations: **int** is good enough. Even if we had a really good **uni**, we shouldn't start its introduction by breaking **span** so that we could fix it later.

Conclusion

std::span is doing "the right thing"(tm). We should not "fix it" to do the wrong thing"(tm) just like the STL containers do. Instead, we should use **span** as the vanguard of a change to do the mathematically right thing throughout. This is an opportunity, possibly the only opportunity we'll get. If we lead, tools and teaching materials will follow because we would be moving to a simpler world.

Acknowledgements

Thanks to Herb Sutter and Neil MacIntosh for constructive comments on early drafts of this paper.