Sizes Should Only span Unsigned

P1089
Attention: LEWG
Date: 5/6/2018
Authors:
Robert Douglas <rwdougla at gmail dot com>
Nevin Liber <nevin at cplusplusguy dot com>
Marshall Clow <mclow.lists at gmail dot com>

Introduction

span, as voted into the working draft in Jacksonville in 2018, contains the first instance of a signed size() member function.

Background

Previous Design Discussions

LEWG took a single straw poll on the subject, in Jacksonville in 2016.

From the minutes:

Happy with signed index_type returned by size()?  

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

No minutes since have shown any additional straw polls, though the topic has come up repeatedly. Each time, discussion was shut down before any new straw polls were taken.

7 to 4 is not generally a strong indication of consensus. 7 to 7 is not even a majority in favor. That no follow up discussion and debate have been allowed to happen since should cause alarm.
State of Span

`span` has a particularly unique feature, the template parameter `Extent`. This parameter is signed and given a special value of `-1`, in order to indicate that this view has a run-time provided size. Otherwise, the size of the view is that of this parameter. This is similar to `basic_string_view`'s `npos`, except that `basic_string_view::npos` is unsigned.

As `Extent` is signed, so is `span::index_type`.

User Feedback

An important part of the process, especially when skipping putting a new feature first into a TS, is to solicit for community feedback and reopen discussions based on that feedback.

From experience in integrating `span` into a production code base, it is observable that conflicts between `span::index_type` versus `vector`, `string_view`, and `sizeof(T)` are prevalent. Changing `span::index_type` to `size_t` reduces the number of `static_cast`s needed for type conversion warnings by about 90-95% in this code base. The single remaining source of most conversions is with Posix’s `read()` function, which returns a count of bytes, or a negative number as an error code.

Also, GSL’s `span` tests incorporate 33 uses of `narrow_cast`, to convert various container sizes to `ptrdiff_t` for comparisons.

Even the current C++ Working Draft (N4741) needs normative wording utilizing `static_cast` to make `as_bytes` and `as_writeable_bytes` work. This is done for conversions to `Extent`, but the problem becomes quickly obvious with a decent warning level.

We understand the desire to use a signed type, because in C++ the unsigned integer types have closed arithmetic (it wraps) while the signed integer types do not. However, both `sizeof` and the standard library long ago chose unsigned types (usually `size_t`) to represent sizes and the only thing worse than using a type with closed arithmetic is mixing types. This both breaks consistency with the rest of the standard library and is a pain point due to all the casting required to use it.

Examples

Handling Network Traffic

```cpp
class MyMessageHeader {};
void handleMessage(span<const char> message)
```
```cpp
{
    // Warning: Comparison of signed and unsigned types
    if (message.size() >= sizeof(MyMessageHeader))
    {
        MyMessageHeader const* hdr
            = reinterpret_cast<MyMessageHeader const*>(message.data());
    }
}

Bytes to ASCII text

class Key {};
Key getKey(span<char const> orig);

span<char const> getValue(span<char const> orig);
enum class ValueType { Text, Binary };
ValueType valueType(Key key) { return ValueType::Text; }

template<typename HandlerT>
void parse(span<char const> buffer, HandlerT handler)
{
    Key key = getKey(buffer);
    span<const char> value = getValue(buffer);
    switch (valueType(key))
    {
        case ValueType::Text:
            // Warning: narrowing conversion
            handler(string_view{value.data(), value.size()});
        case ValueType::Binary: // Omitted for brevity
            break;
    }
}

Design Discussion

3 options should be considered:
1) Change index_type to be unsigned. Suggest: size_t to directly match
   basic_string_view::size_type.
2) Change both index_type and Extent to be unsigned. Make dynamic_extent
   numeric_limits<index_type>::max()
3) As another option, we may consider breaking out dynamic `span` into a separate type and remove `dynamic_extent` altogether, however that wording is not provided at this time.

4) (Another option would be to take this out of C++20 and put it in Lib Fund, but I’m not sure we dare actually say that)

Option 1 is the simplest means, given the state of N4741, to get type of `size()` back in line with the rest of the standard. However, it also creates a discrepancy internal to `span<>`, via `Extent as size()`.

Option 2 gets `span` in full parity to the rest of the standard, but is simply a larger design change. From the changes to the proposed wording, though, this an overall simplification of the specification through simplified requirements, eliminated ill-formed condition, and removed `static_cast`s.

**Proposal 1**

Change span synopsis  `[span.overview]` paragraph 5

using `index_type = ptrdiff_t size_t;`

**Proposal 2**

Change `[span.syn]`

```cpp
inline constexpr ptrdiff_tsize_t dynamic_extent = -numeric_limits<size_t>::max();
template<class ElementType, ptrdiff_t Extent = dynamic_extent>
class span;
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator==(span<T, X> l, span<U, Y> r);
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator!=(span<T, X> l, span<U, Y> r);
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator<(span<T, X> l, span<U, Y> r);
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator<=(span<T, X> l, span<U, Y> r);
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator>(span<T, X> l, span<U, Y> r);
template<class T, ptrdiff_tsize_t X, class U, ptrdiff_tsize_t Y>
constexpr bool operator>=(span<T, X> l, span<U, Y> r);
```
template<class ElementType, ptrdiff_t Extent>
span<const byte, Extent == dynamic_extent ? dynamic_extent : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
as_bytes(span<ElementType, Extent> s) noexcept;
template<class ElementType, ptrdiff_t Extent>
span<byte, Extent == dynamic_extent ? dynamic_extent : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent>
as_writable_bytes(span<ElementType, Extent> s) noexcept;

Change span synopsis [span.overview]

3 If Extent is negative and not equal to dynamic_extent, the program is ill-formed.

template<class ElementType, ptrdiff_t Extent = dynamic_extent>
class span {
    using index_type = ptrdiff_t;

template<class OtherElementType, ptrdiff_t OtherExtent> constexpr
    span(const span<OtherElementType, OtherExtent> & s) noexcept;

template<ptrdiff_t Count> constexpr
    span<element_type, Count> first() const;
template<ptrdiff_t Count> constexpr
    span<element_type, Count> last() const;
template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
    span<element_type, see below > subspan() const;

Change [span.sub]

template<ptrdiff_t Count> constexpr
    span<element_type, Count> first() const;
  1. Requires: 0 <= Count && Count <= size().

template<ptrdiff_t Count> constexpr
    span<element_type, Count> last() const;
  3. Requires: 0 <= Count && Count <= size().
template<ptrdiff_t Offset, ptrdiff_t Count = dynamic_extent>
constexpr span<element_type, see below> subspan() const;

5. Requires:
   (0 <= Offset && Offset <= size())
   && (Count == dynamic_extent || Count >= 0 && Offset + Count <= size())

8. Requires: 0 <= count && count <= size().

10. Requires: 0 <= count && count <= size().

12. Requires:
    (0 <= offset && offset <= size())
    && (count == dynamic_extent || count >= 0 && offset + count <= size())

Change [span.elem]
1. Requires: 0 <= idx && idx < size().

Change [span.comparison]

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator==(span<T, X> l, span<U, Y> r);

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator!=(span<T, X> l, span<U, Y> r);

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator<(span<T, X> l, span<U, Y> r);

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator<=(span<T, X> l, span<U, Y> r);

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator>(span<T, X> l, span<U, Y> r);

template<class T, ptrdiff_t X, class U, ptrdiff_t Y>
constexpr bool operator>=(span<T, X> l, span<U, Y> r);

Change [span.objectrep]

template <class ElementType, ptrdiff_t Extent>
span<const byte,
Extent == dynamic_extent ? dynamic_extent
  : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent
as_bytes(span<ElementType, Extent> s) noexcept;

template<class ElementType, ptrdiff_t Extent>
span<byte,
Extent == dynamic_extent ? dynamic_extent
  : static_cast<ptrdiff_t>(sizeof(ElementType)) * Extent
as_writable_bytes(span<ElementType, Extent> s) noexcept;