A Few Good Types

Evolving array_view and string_view
for safer C++ code.

Neil MacIntosh
neilmac@microsoft.com

Microsoft

array_view in action: (credit: Herb Sutter)

Before

```
void f(_In_reads_(num) Thing* things, unsigned count) {
   unsigned totalSize = 0;
   for (unsigned i = 0; i <= count; ++i)
      totalSize += things[i].GetSize();
         // SA can catch this error today

    memcpy(dest, things, count);
         // SA can catch this error today


}

void caller() {
   Thing things[10]; // uninitialized data

   f(things, 5);
   f(things, 30);    // wrong size
}        // SA can catch these today
```

After

(ptr,len) → **array_view**

```
void f(array_view<const Thing> things) {
   unsigned totalSize = 0;
   for (auto& thing : things)
      totalSize += thing.GetSize();


   copy(things, dest);  // better; use std::copy (range)


}

void caller() {
   Thing things[10];     // uninitialized data
   f(things);            // length ok & uninit compile err
   f({things, 5});       // ok, and convenient: add {}
   f({things, 30});      // compile-time error
}
```

deduce array length

call site **compat**: add {}

compile-time error (no SA required)

Some goals of our effort

- Encourage safety-by-construction in systems programming
  - Prevent defective code from being compiled
  - Catch defects at runtime and fail fast
  - Maintain C++ advantages of efficiency and control (“close to the metal”)

- Enable modern static analysis techniques
  - Experience: key is to have memory-access semantics clearly described for types
  - Enable deep insights into program behavior and early defect detection

- Standards-based, portable and open source library implementation
  - Open source on GitHub now: http://github.com/Microsoft/GSL
  - Support some major compilers out-the-box (MSVC 2013, MSVC 2015, Clang, GCC)
  - Support three platforms out-the-box (Linux, Windows, OS X)

Specific goals for these types

- Memory safety
  - These types enforce bounds-safety, other features prevent dangling.
  - Replace nonstandard annotations (e.g. SAL)
- Type safety
  - Ensure that unsafe type conversions are prevented
- Efficiency
  - Zero-overhead when compared to equivalent hand-written checks.
  - Low-overhead compared to unsafe code it replaces
- Abstraction
  - Separate concerns: data access (view) from storage (container)

array_view<ValueType,Extents...>

- A view over a contiguous range of elements with a known length
  - “Pointer & Length”
  - Cheap to copy and move: this is a value type
  - Designed to replace any observing pointers that point to more than one object

- Length can be fixed at compile time or specified at runtime
  - defaults to runtime (dynamic_range)

```
array_view<int> av1 = ...;     // seq. of ints with dynamic length
array_view<int, dynamic_range> av2 = ...; // same as above
array_view<int, 10> av3 = ...; // sequence of exactly 10 ints
```

array_view

- Small storage overhead: conceptually { T*, size_type }
    - When fixed-length: storage requirements become just { T* }: zero overhead!

- No allocation (ever) – provides a view on existing storage
    - size immutable after construction

- All accesses are bounds-checked. Always.
    - Violations result in fail-fast.

```
int read(char* packet, size_t length, /*...*/) {

    // ensure the packet we received is large enough
    size_t needed = sizeof(Foo) + sizeof(Bar);
    if (length < needed)
        return -1;

    // write foo
    Foo* foo = (Foo*)packet;
    foo->someEntry = ...; // write the fields of foo

    // write bar
    packet += sizeof(Foo);
    Bar* bar = (Bar*)packet;
    bar->someField = ...; // write the fields of bar

    // write a fuzzbuzz
    packet += sizeof(Bar);
    FuzzBuzz* fuzzbuzz = (FuzzBuzz*)packet;
    fuzzbuzz->anotherField = ...; // write the fields of fuzzbuzz
}
```

Wait, WHAT?

```
int read(array_view<byte> packet, /*other stuff*/) {

    // ensure the packet we received is large enough
    constexpr size_t needed = sizeof(Foo) + sizeof(Bar);
    if (packet.length() < needed)
        return -1;

    // write foo
    auto foo = p.as_array_view<Foo, 1>();
    foo[0].someEntry = ...; // write the fields of foo

    // write bar
    p = p.sub(sizeof(Foo));
    auto bar = p.as_array_view<Bar, 1>();
    bar[0].someField = ...; // write the fields of bar

    // write a fuzzbuzz
    p = p.sub(sizeof(Bar));
    auto fuzzbuzz = p.as_array_view<FuzzBuzz,1>();
    fuzzbuzz[0].anotherField = ...; // write the fields of fuzzbuzz
}
```

Safe: will fail-fast.

Safety Features

- Only allow safe conversions

```
array_view<int> ==> array_view<const int> // ok!
array_view<int> ==> array_view<short> // won’t compile!

// only compiles when is_simple_layout_type<T>...
array_view<byte> ==> array_view<T> // ok!
```

- Constructs readily and sensibly from existing containers
  - arrays, std::array, vector... deduces size automatically.

Safety Features

- When array_view is fixed-size, we can use the type system to enforce bounds-safety

```
int arr[] = { 1, 2, 3, 4 };
array_view<int, 4> av4 = arr; // safe, fixed size view of 4

array_view<int, 2> av2 = arr; // ok, 2 < 4 so conversion allowed
av2 = av4; // ok, 2 < 4 so conversion allowed
av4 = av2; // error – fails to compile as types are not compatible

array_view<int> av_dyn = av2; // ok, going from fixed to dynamic
av4 = av_dyn; // dynamic to fixed will fail-fast on bounds-check
```

```
void Write(_In_reads_(count) const char* s, size_t count);

void WriteXml(_In_reads_(cchText) PCSTR szText, size_t cchText)
{
    if ((size_t)-1 == cchText) // invisible to callers
        cchText = strlen(szText);

    while (cchText)
    {
        if (*szText == '&')
            Write(“&amp;”, sizeof(“&amp;”));
        else
            Write(szText, 1);

        cchText--;
        szText++;
    }
}
```

Whoops!

```
void Write(cstring_view s);

void WriteXml(cstring_view text)
{
    // no longer need strlen-on-special-case-length

    auto it = text.begin();
    while (it != text.end())
    {
        if (*it == '&') // bounds-checked
            Write(ensure_z(“&amp;”)); // safe and explicit
        else
            Write({*it, 1});

        ++it; // bounds-checked
    }
}
```

```
void Write(cstring_view s);

void WriteXml(cstring_view text)
{
    // no longer need strlen-on-special-case-length

    for (auto c : text)) // cannot overrun
    {
        if (c == '&')
            Write(ensure_z(“&amp;”));
        else
            Write({c,1});
    }
}
```

string_view<CharType, Extent>

- A view over a contiguous range of elements with a known length
  - “Pointer & Length”
  - Cheap to copy and move: this is a value type

- Length can be fixed at compile time or specified at runtime

- No allocation (ever) – they provide a view on existing storage
  - size immutable after construction

Sound familiar?

- All accesses are bounds-checked. Always.
  - Violations result in fail-fast.

string_view<CharType, Extent>

- It is just an alias for array_view

```
template <class CharT, size_t Extent = dynamic_range>
using basic_string_view =
      array_view<array_view_options<CharT, unsigned short>, Extent>;
```

- Convenient aliases for common cases: “w”ide chars and “c”onst views
  - string_view
  - cstring_view
  - wstring_view
  - cwstring_view

string_view<CharType, Extent>

- “string” operations become free functions (find, compare, trim....)
  - need to add these to our current GSL implementation

- Agnostic regarding zero-termination
  - require you to be explicit when initializing from zero-terminated strings
  - puts more information into the source code

```
void f(const char* s) {
    string_view sv = ensure_z(s); // initializes correctly
    ...
}
```

Extracting sub-views

- "Trim" operations for sub-views are convenient for one-dimension cases

```
array_view<T, Count> first<Count>() const;
array_view<T> first(size_type count) const;

array_view<T, Count> last<Count>() const;
array_view<T> last(size_type count) const;

array_view<T, Count> sub<Offset, Count>() const;
array_view<T> sub(size_type offset, size_type count) const;
```

- Arbitrary creation of new sub-view (also works for dimensions > 1)

```
array_view<T> section(size_type offset, size_type count) const;
```

Interoperability with legacy code

- Constructor from (T*, size_type)
  - Allows construction from parameters that cannot change (ABI compat)

- Direct access to raw pointer

  ```
  T* data();
  ```

  - Allows access to the underlying data for passing to legacy functions

- Using these would require you to [[suppress(bounds)]] as you are performing “trust-me” operations

Diffs from N3851 proposed array_view

- Adds the possibility of fixing extents for each dimension
- Adds conversions to/from byte-representation
  - as_bytes(), as_array_view()
- Adds more ctors to support drop-in use
- Describes length in both elements and bytes
  - length()/bytes()
- Allows specification of a size type for measuring/indexing
- More slice-n-dice operations: first(), last(), sub()

Diffs from Lib. Fundamentals TS: string_view

- Is a type alias for array_view<CharType...>
  - Adds the possibility of fixing length statically
  - Adds conversions to/from byte-representation
  - Describes length in both elements and bytes
  - Allows specification of a size_type for measuring/indexing

- Allows string views of mutable or immutable characters
- Requires explicit construction from zero-terminated strings
- Has string-specific functions as free functions

Early lessons from usage

- Easy replacement at callsites – nearly always ”just add braces”

```
foo(p, len); ==> foo({p, len});
```

- Required little change inside callees besides length calculations

```
for(UINT i = 0; i < len; i++) ==> for(UINT i = 0; i < p.length(); ++i)
```

- bytes/elements difference makes code clearer to read
- Need to wrap standard and common libs to understand array_view<byte> (at the least)

```
memcpy(), memset(), ZeroMemory(), CopyMemory(), ...
```

Performance

- Performance target: zero overhead
  - When compared to pointer+length code **that has equivalent checks** and ensures safety
  - Compared to unsafe code – some overhead, but as low as possible

- Have begun work to optimize array_view in MSVC compiler
  - Will show up in future VS updates.
  - More detailed deep dives as we make progress.

Performance: Key Insights

- Optimization can leverage guarantees provided by type system
  - e.g. semantics of default copy constructors, const on globals
  - make it clear to the optimizer you are a simple, safe type

- Range-check optimizations are important and do-able
  - hoisting, elimination...
  - considerable body of theory there (and growing)
  - MSVC already knows how to do efficient range-checking for .NET Native compilation – we get to use RNGCHK() without any of the overheads of GC, framework or runtime.dll.

```
typedef int my_array[9];
my_array glob;

void f(my_array a) {
    // a is effectively a pointer, and the compiler
    // knows that from here, the value of that pointer cannot change

    for (int i = 0; i < len; ++i)
        a[i] = glob[i]; // the compiler knows the address of glob cannot change

    // loop is monotonically increasing over an induction variable
}
```

- Address of ‘a’ can be passed in a register and loaded once
- Address of ‘glob’ can be loaded once
- Strength reduction can be performed on the loop
- Basically…this simple C code becomes few instructions and **fast**

```
int arr[9] = { ... };
const array_view<int,9> glob = arr; //const means glob’s int* member won’t change

void f(array_view<int,9> p) {
    // p contains a single int*. The compiler knows that from here on
    // the value of that int* cannot change.

    for (int i = 0; i < p.length(); ++i)
        a[i] = glob[i]; // this line causes a bounds-check

    // loop still has the same properties as previous form
}
```

- ‘p’s pointer member can be passed in a register and loaded once
- Strength reduction can be performed on the loop
- This becomes few instructions and **fast**
- **But what about the range check? It can be eliminated (proved away)!**
  - Compiler recognizes the bounds-check instruction inside op[]
  - In cases it can’t be eliminated, it can often be hoisted above the loop instead

Join the fun!

- There is a reference open source implementation....
  - Improve it
  - Port it
  - Use it
  - Give feedback and suggestions
  - Write your own that's faster/smaller/...

- Resources:
  - https://github.com/Microsoft/GSL (reference implementation)
  - http://isocpp.org/ (for links to the array_view and string_view proposals)