Win an Xbox One S

Take this survey: http://aka.ms/cppcon

The Guideline Support Library (GSL).

One year later.

Neil MacIntosh
neilmac@microsoft.com

Microsoft

Agenda

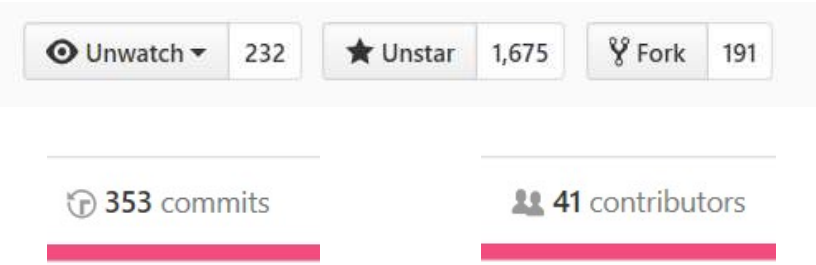
- Refresher on the GSL and Microsoft's implementation.
- What happened to each over the past year.
- What's planned for each going forward.
- How you can help.

In last year's episode of CppCon...

- We announced the Guideline Support Library (GSL)
  - Supports safety-by-construction in C++ programming.
  - Goes hand-in-hand with the C++ Core Guidelines – referenced throughout.

- We released Microsoft's implementation (Microsoft GSL)
  - Freely available on GitHub.
  - Standards-based, portable, open-source implementation.
  - Support for major compilers out-the-box (MSVC, Clang, GCC).
  - Support for major platforms out-the-box (Windows, Linux, OS X).

Now, one year later...



	Open	Resolved	Declined	Duplicate
Issues	37	122	39	10
Pull Requests	14	126	19	3

- Just over 3K lines of non-comment source code
- Just over 4K lines of non-comment source code for tests

What is the GSL all about again?

- These types and functions support simpler, better C++ programming
  - Useful abstractions
  - Memory safety, type safety are the key concerns
- These types and functions support efficient C++ programming
  - Aim for zero-overhead when compared to equivalent hand-written checks.
  - So low-overhead compared to unsafe code it replaces

What's in the GSL?

Owners/Containers

owner<T>

unique_ptr<T>

shared_ptr<T>

dyn_array<T>

stack_array<T>

Views

span<T>

string_span<T>

(cw)zstring

Utilities

not_null<T>

finally()

Contract support

Ensures()

Expects()

Conversions

narrow()

narrow_cast()

Concepts

String

Number

Sortable

Pointer

Convertible

...

Rough guide only...consult the definitive reference at:

https://github.com/isocpp/CppCoreGuidelines

But what about the standard library?

- It's awesome! And getting better all the time.
- Some elements of the GSL are already just standard library types:

```
using unique_ptr = std::unique_ptr;
using shared_ptr = std::shared_ptr;
```

- Parts of the GSL are progressively being proposed for standardization

```
span<T>
byte
```

Both on track for C++20!

Where is it, how can I use it?

- https://github.com/Microsoft/GSL
- header-only, so drop the header files into your favorite location for libraries, #include in your code and go!
- now available as a VCPkg for Windows platforms!
  - took 20 minutes in-between sessions yesterday
  - https://github.com/Microsoft/vcpkg
- also available as a NuGet package
  - https://www.nuget.org/packages/Microsoft.GSL/

~~There can be only one~~
There is not only one...

- Microsoft's implementation of the GSL
  - https://github.com/Microsoft/GSL
- Martin Moene's GSL-lite:
  - https://github.com/martinmoene/gsl-lite
- Mattia Basaglia's GSL within Melanolib:
  - https://github.com/mbasaglia/Melanolib
- Vicente Botet Escriba's fork:
  - https://github.com/viboes/GSL
- Others? Please let me know!

So where's the documentation?

- https://github.com/isocpp/CppCoreGuidelines
  - Brief overview of the main types and functions.
  - Standardization-style specifications is the ultimate aim.
    (so far, just the standardization proposals)

- We realize some standards proposals are not everyone's definition of "documentation". But they are useful, especially for other implementers.

- Some tutorial-style, usage guidance is on the way.

- This is a great place to offer contributions.

Who uses the Microsoft GSL?

- Microsoft does! (dogfooding, living the dream)
  - CppCoreCheck
  - Microsoft Visual C++ compiler
  - Office
  - Edge browser

- Bareflank Hypervisor (http://bareflank.github.io/hypervisor/)
- Boost.AFIO v2 (https://ned14.github.io/boost.afio/)

- King Entertainment (makers of Candy Crush Saga)
  - use their own (forked) version of `span`

So what happened to the GSL in a year?

- Focused on standardizing the most mature types first:

```
array_view<T>

string_view<T>

byte
```

array_view<T>...now span<T>

- We took `array_view` to the C++ standardization process
  - Now on track for inclusion in the C++20 standard library.
    (It has been recommended to the Library Working Group from the Library Evolution Working Group)

- Lots of great feedback and refinements:
  - renamed from `array_view<T>` to `span<T>.`
  - `span<T>` become single-dimension only (see `multi_span<T>` for old version).
  - `span<T>` interface refined (still some tidy-ups to go).

- http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0122r3.pdf

string_view<T>...now string_span<T>

- We took `string_view` to the C++ standardization process
  - Renamed to `string_span.`
  - Withdrew the proposal in Oulu meeting.
  - Clear that existing basic_string_view was a better fit for the standard library.
- Still evaluating how useful `gsl::string_span` will be in a world of `std::string_view`.
- `gsl::zstring_span` and friends definitely still useful
  - to identify `CharT*` that are null-terminated strings (legacy APIs).

byte

- We took `byte` to the C++ standardization process
  - recognized as addressing a long-standing and painful hole in the language.
  - needed to update core language rules to allow accessing object storage through `byte*`.
  - added basic bitwise operations and comparisons.
  - just missed the cutoff for C++17 (ran out of time in Oulu).
  - scheduled for inclusion in C++20.

  - http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0298r1.pdf

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

```
bool VerifyHeader(gsl::span<const byte, HeaderSize> s); // fixed-size span makes implementation safer
PacketResults ProcessBody(gsl::span<const byte> s); // dynamic-size span useful too

bool HandlePacket(const byte* p, int length) // APIs can still use pointer+length for back compat
{
    gsl::span<const byte> s {p, length} ; // trust-me point!
    if (s.size() <= HeaderSize) // handles nullptr cases safely!!!
         return false;

    if (!VerifyHeader(s)) // safe conversion to fixed-size span
        return false;

    // cheaply get the bytes after skipping the header
    auto results = ProcessBody(s.subspan<HeaderSize>());
    // ...
}
```

What happened to Microsoft GSL in a year?

- Mainly busy tracking changes to `span`, `string_span` and `byte`.

- Discussions around and implementation improvements to:
  `final_act/finally()`
  `narrow()/narrow_cast()`
  `not_null`
  `Expects/Ensures`

So what's next for the GSL?

- Finalize standardization of `span` and `byte`
- More documentation
  - usage tutorials and specifications for all components.
- `not_null<T>`
  - need to think about design around owner and smart-pointer cases a little more.
  - consider for standardization.
- `multi_span<T,...>`
  - what are the right target scenarios for this type?
  - how does it relate to the standardization proposal for a multidimensional view? http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0009r2.html

So what's next for Microsoft GSL?

- Look at some new container types
  `stack_array`
  `dyn_array`

- Better release management
  - designate and label releases when significant checkpoints are reached
  - releases likely to roughly follow MSVC update and ship cycle
  - deprecate MSVC 2013 when MSVC Next release

- Performance tuning
  - optimizations for `span`

Performance and the GSL

- Performance target: zero overhead
  - When compared to handwritten code **that has equivalent checks** to ensure safety.
  - Compared to unsafe code – some overhead, but as low as possible.
  - We want the code to be correct, simple, and fast.

Performance tuning in the Microsoft GSL

- All runtime contract checks are performed using Expects/Ensures
- Can compile checks to fail-fast (std::terminate) using GSL_TERMINATE_ON_CONTRACT_VIOLATION
  - (recommended path for safety, offers optimization opportunities)
- Can compile checks to throw exceptions using GSL_THROW_ON_CONTRACT_VIOLATION
  - not recommended except for testing, but may be useful in some environments)
- Can compile without runtime checks using GSL_UNENFORCED_ON_CONTRACT_VIOLATION
  - (not recommended unless you are very confident)

Performance tuning in the Microsoft GSL

- For span<T>, we want to run with runtime checks but still be fast.
  - Worked on optimizing span<T> with MSVC compiler
- Optimizer support shipping since MSVC 2015 Update 2.
  - Currently, only available for x64 architecture target, other architectures later
- Source changes to span<T> implementation required
  - will become available on GitHub post-CppCon
  - Watch the VC Blog for more details over the next few months
  - https://blogs.msdn.microsoft.com/vcblog/

Performance tuning: sneak peek

```
void elide(int i, gsl::span<int, 17> a) {
    a[4] = 2; // elide check completely as it is redundant
}
void eliminate_redundant(int i, span<int> a) {
    int j = // ... ;
    if ((a[i+j] == 1))  {
        a[i+j] = 0; // elide redundant check and rely on branch-entry check
    }
}
void hoist_and_rewrite(int i, int u, gsl::span<int> a) {
    for (int i=0; i<u; i++)
        a[i] = 2; // check u <= a.size() outside loop instead.
}
```

Join the party!

- This is an open source project....
  - Improve it – log issues for bugs found (better yet, fix them in a PR!!!).
  - Port it. Let us know if you have success with a new platform or compiler.
  - Use it. Improve your code! Tell us, where are you using the GSL?
  - Give feedback and suggestions. They are always appreciated!

- Resources:
  - https://github.com/isocpp/CppCoreGuidelines
  - https://github.com/Microsoft/GSL
  - http://isocpp.org/
  - https://blogs.msdn.microsoft.com/vcblog/