Compile-Time Compression and Resource Generation with C++20

ASHLEY ROLL

Cppcon
The C++ Conference

2021 October 24-29

Introduction

Explore how C++20's constexper features can:

- Generate data from code at compile-time
- Be used to construct:
    - Lookup Tables
    - Configuration Fuses
    - Compressed Strings
    - USB Descriptors

Along the way

- Introduce some libraries I created for this code
- Discuss some techniques I found building compile-time libraries

constexpr in Brief

- Specifies a variable or function CAN appear in a constant expression
- Constant expressions can be evaluated at compile-time
- eg:
    - Array size
    - non-type template parameter

constexpr Variable

- Must be a Literal Type
  - scalar (int, char, etc)
  - array of literals
  - struct/class/union (with some constraints)
  - a closure type (lambda)
- Must be immediately initialised
- Implies const

```
constexpr int Base{0b0010'1000};
constexpr std::array<int, 3> Values{1, 2, 3};
int main() {
    // Base += 1;    // ERROR
    return Base + Values[1];
}
```

constexpr Function

- Must return a Literal Type (or void)
  - must ensure all members of return value are initialised
- Can be a constructor
  - must initialise all members of object
- parameters must be a literal type (if any)

```
constexpr int make_bigger(int v)
{
    return v * 100;
}
```

consteval

- Specifies that a function MUST produce a compile-time constant
- Implies constexpr

constinit

- Specifies a variable has static initialization
  - constant initialised (from a constant expression)
- computed at compile-time, and stored in program binary
  - can be mutable or const

```
constinit int foo = 10;
constinit const int bar = 1;
int main() {
    foo += 1;
    // bar += 1;  // ERROR
}
```

Building Resources

- Lookup Tables
- Configuration Fuses
- Compressed String Tables
- USB Descriptor

Code samples available on GitHub

https://github.com/AshleyRoll/cppcon21

Why

- No external tools
- Can enforce correctness
- Pushing limits is fun 😁

Lookup Tables

Lets make a lookup table that does linear interpolation.

- Warning: constexpr <cmath> is a gcc extension [P1383]

- Table of sin(x) where x is in degrees, not radians
- Reusable type, just plug in a different function

```
constexpr float radians(float degrees) {
    return (std::numbers::pi/180.0)*degrees;
}
constexpr float sine_deg(float degrees ) {
    return std::sin(radians(degrees));
}

constexpr auto DegreeSineTable =
  LerpTable<float, 32>::make_table(0.0f, 90.0f, sine_deg);

static_assert(DegreeSineTable(30.0f) >= 0.499f
           && DegreeSineTable(30.0f) <= 0.501f);
```

```
template<typename T, std::size_t NUM_ENTRIES>
struct LerpTable
{
    constexpr T operator()(T in) const
    { ... }

    constexpr static LerpTable make_table(
        T min, T max, T (&function)(T)
    )
    { ...  }

private:
    struct Entry
    {
        T input;
        T output;
    };

    std::array<Entry, NUM_ENTRIES> entries;
}
```

```
constexpr static LerpTable make_table(
    T min, T max, T (&function)(T) // or: auto function
)
{
    LerpTable table;

    // fill min/max value explicitly
    table.entries[0] = {min, function(min)};
    table.entries[NUM_ENTRIES-1] = {max, function(max)};

    // fill the other entries
    T const step = (max - min) / NUM_ENTRIES;
    for(std::size_t i = 1; i < NUM_ENTRIES-1; ++i) {
        T const a = step * i;
        table.entries[i] = {a, function(a)};
    }

    return table;
}
```

```cpp
constexpr T operator()(T in) const
{
    auto const clamped_in = std::clamp(
        in,
        entries[0].input,
        entries[NUM_ENTRIES-1].input);

    auto entry_itr = std::lower_bound(
        entries.begin(), entries.end(), clamped_in,
        [](auto const &e, auto const &v) { return e.input < v;});

    auto const entry = *entry_itr;

    if(entry.input == clamped_in) {
        return entry.output; // handle exact match / first entry
    }

    auto const prev_entry = *std::prev(entry_itr);
    auto const t = (clamped_in - prev_entry.input)
                   / (entry.input - prev_entry.input);

    return std::lerp(prev_entry.output, entry.output, t);
}
```

Config Fuses

- Initialise hardware before the processor starts
- Fixed locations in Flash memory, filled with bit-mapped magic values
- Sets up
  - clocks, memory segments, watchdog timer
  - JTAG debug, code security, and much more

Config Fuses

- Vendor specific compiler extensions to set fuses
    - pragmas, or similar
    - special macros
- Rely on C style #define for bit values
- No validation, potentially very complex

Lets do better

- Strongly type all configuration registers
- Provide a "builder" object
- Render a constinit object, at compile-time
- Place it in Flash using segments and linker script

```cpp
enum class WatchDogMode {/* ... */};
enum class OscillatorMode {/* ... */};

class ConfigBuilder
{
public:
    constexpr void set_watchdog(WatchDogMode wtd)
    {
        m_Wdt = wtd;
    }

    constexpr void set_oscillator(OscillatorMode osc)
    {
        m_Osc = osc;
    }

    constexpr auto build()
    {
        //...
    }

private:
    WatchDogMode m_Wdt {WatchDogMode::Disabled};
    OscillatorMode m_Osc {OscillatorMode::InternalRC};
};
```

```cpp
constexpr auto build()
{
    // Serialise all the registers in correct order and into
    // the correct bit locations without relying on mapping
    // structs and packing correctly
    // Lets pretend the registers are 32 bits, and there 2 of them
    std::array<std::uint32_t, 2> registers;

    auto wdt = static_cast<std::uint32_t>(m_Wdt);
    auto osc = static_cast<std::uint32_t>(m_Osc);

    // lets pretend we need values and their complement
    registers[0] = (wdt << 24u) | (~wdt & 0x0000'00FF);
    registers[1] = (osc << 24u) | (~osc & 0x0000'00FF);

    return registers;
}
```

Using it

```
[[gnu::section(".config_registers"), gnu::used]]
constinit auto const CONFIG_REGISTERS = []{
    ConfigBuilder cfg;
    cfg.set_watchdog(WatchDogMode::Enabled_10ms);
    cfg.set_oscillator(OscillatorMode::Crystal);
    return cfg.build();
}();
```

- capturless lambda is constexpr
- constinit const ensures it is immutable and built at compile-time
- gnu:section places value in .config_registers section
- gnu:used stops compiler discarding it if not referenced in code

Linker script

- Defines layout for final binary/image
- Maps sections into memory regions
- Normally you never need to know..
    - until you do..

- We need to ensure CONFIG_REGISTERS is placed in a very specific location for the hardware
- Lets pretend that is address 0x0100
- Define MEMORYs in the script, add one for config registers
- Map our .config_registers segment to that

```
/* World's worst linker script */
ENTRY(main)
MEMORY
{
  config    : ORIGIN = 0x0100, LENGTH = 0x8
  prog (rx) : ORIGIN = 0x1000, LENGTH = 0x100000
}

SECTIONS
{
  /* place config registers in to the right memory location  */
  .config_registers : {
    KEEP(*(.config_registers))
  } > config

  /* everyting else - don't do this */
  .text     : { *(.text) *(.text.*) } > prog
  .data     : { *(.data) } > prog
  .bss      : { *(.bss) } > prog
  /DISCARD/ : { *(.*) }
}
```

Compile and link with the linker script

`build.sh`

```
gcc -O3 --std=c++20 -c main.cpp -o main.o
ld -o config.out -T config_register.ld main.o
objdump -ds config.out
```

```
$ ./build.sh

config.out:     file format elf64-x86-64

Contents of section .config_registers:
 0100 fd000002 fc000003                    ........
Contents of section .text:
 1000 31c0c3                               1..

Disassembly of section .text:

0000000000001000 <main>:
    1000:       31 c0                   xor    %eax,%eax
    1002:       c3                      ret
```

Where to next?

- We've created simple resources of known size/type
- What about varying the output size/type based on the input?
  - A USB configuration descriptor can have many interfaces,
  - and each interface can have many endpoints
  - Compressed string data depends on the content of the string, not its length
- How do we build easy to use libraries?

Return Size & Type

- The size of an array or template parameters of a type must be compile-time constants.
- We must be able to calculate these values using constexpr functions
- This means the compiler has to use type information, it can't use parameters to functions.
- How can we pass user-supplied data into a constexpr function?
  - Lambdas!

Lambdas for the win!

- A lambda's call operator can be constexpr
- Each lambda is a unique type and its return type is known at compile-time
- Therefore we can write helper methods that take a user-suppiled lambda to generate the data needed to render our desired compile-time resource!
- These are effectively templated functions, but we will use the cleaner auto parameter syntax for our helper functions

Library sketch

```
// library
constexpr auto every_second_item(auto lambda)
{
    constexpr auto data = lambda();
    constexpr auto length = data.size() / 2;
    std::array<int, length> output;

    for(std::size_t i = 0; i < output.size(); ++i)
        output[i] = data[i*2];

    return output;
}

// invoke
constexpr auto test = every_second_item([] {
    return std::to_array<int>({1, 2, 3, 4, 5, 6});
});

int main() { return test.size(); } // == 3
```

String Compression

Lets make a compressed string table

- https://github.com/AshleyRoll/squeeze
- map from enum Key to Compressed String
- Huffman Coding for compression
- Output struct:
    - Mapping of Key -> bit stream location/length
    - Compressed bitstream
    - Huffman tree encoded into array
    - Lookup -> iterator over compressed data

```
enum class Key { String_1, String_2, String_3 };

static constexpr auto buildMapStrings = [] {
    return std::to_array<squeeze::KeyedStringView<Key>>({
      // out of order and missing a key
      { Key::String_3, "There is little point to using short strings in a
      { Key::String_1, "We will include some long strings in the table to
    });
};

constexpr auto map = squeeze::StringMap<Key>(buildMapStrings);

int main()
{
    static constexpr auto str1 = map.get(Key::String_1);
    for (auto const &c : str1) {
        std::putchar(c);
    }

    std::putchar('\n');
    return 0;
}
```

Overview

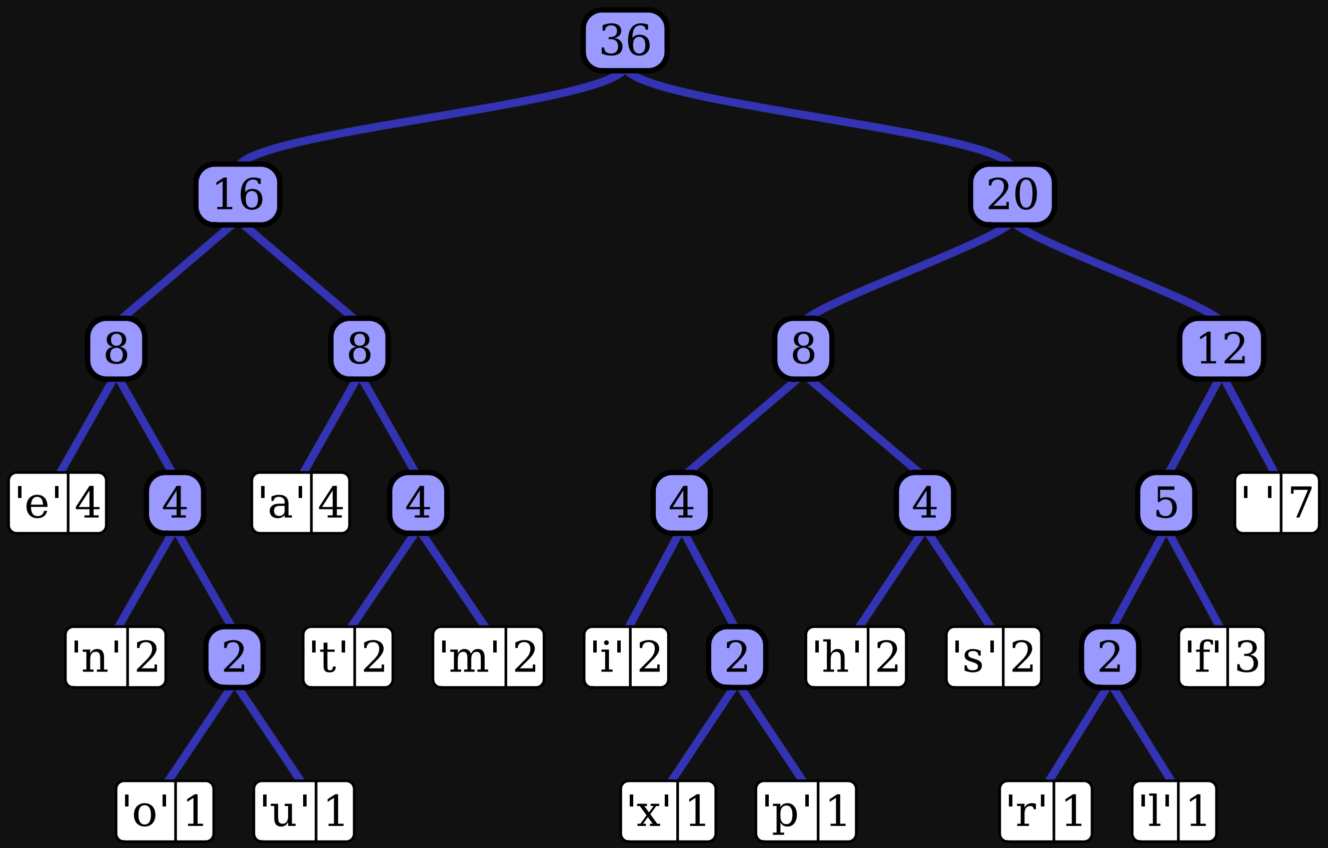
- Under the StringMap is a StringTable - simple indexed access to compressed string
- Wrap that with mapping Key -> index
- Both StringMap and StringTable are usable
- Strings (and keys) are passed to library using lambda providing std::string_view
- Iterator interface to avoid need to decompress full string
- Useful for long strings - minimum metadata size

```
$ objdump -s --section=.compressedmap example

example:     file format elf64-x86-64

Contents of section .compressedmap:
 20c0 00000000 00000000 01000000 00000000  ................
 20d0 02000000 00000000 00000000 00000000  ................
 20e0 00000000 00000000 4a000000 00000000  ........J.......
 20f0 24010000 00000000 3a000000 00000000  $.......:.......
 2100 85fbe507 2e60c7ef 50b4675d 4e741d6f  .....`..P.g]Nt.o
 2110 158f1674 e5d03b97 687ac893 5f8e1674  ...t..;.hz.._..t
 2120 9dcf2cfe 5afc5d10 31741279 f9a3a6bf  ..,.Z.].1t.y....
 2130 d0aea305 5d39f47c ff7c66f1 b3ce4f0f  ....]9.|.|f...O.
 2140 be020100 02000100 03000400 01000500  ................
 2150 06000100 07000800 01000900 0a000100  ................
 2160 0b000c00 01000d00 0e000100 69000000  ............i...
 2170 00007300 00000000 0f001000 01007400  ..s...........t.
 2180 00000000 11001200 01001300 14000100  ................
 2190 20000000 00001500 16000100 72000000   ...........r...
 21a0 00006f00 00000000 6c000000 00001700  ..o.....l.......
 21b0 18000100 19001a00 01006e00 00000000  ..........n.....
 21c0 1b001c00 01006500 00000000 1d001e00  ......e.........
 21d0 01001f00 20000100 21002200 01006700  .... ...!."...g.
 21e0 00000000 23002400 01002500 26000100  ....#.$...%.&...
 21f0 63000000 00006d00 00000000 62000000  c.....m.....b...
```

Huffman coding (overview)

- Optimal encoding given the list of symbols and their frequency
- symbols are all characters in all strings
- Uses a min-heap priority_queue to build a tree bottom up from least used to most used
- Bit pattern assigned by walking down the tree

  Most common symbols have shortest bit pattern



"this is an example of a huffman tree"

https://en.wikipedia.org/wiki/Huffman_coding

Library Process

- Build character frequency table
- Build Huffman tree
- Build char -> bitstream cache
- Build encoded bit stream
- Build index -> bit position / string length lookup
- Package Huffman tree to array
- Combine Huffman tree, bit stream and index lookup into result

Constrain Lambdas

```cpp
template<typename TKey>
struct KeyedStringView
{
    TKey Key;
    std::string_view Value;
};

template<typename T>
concept CallableGivesIterableStringViews = requires(T t) {
    t();
    std::input_iterator<decltype(t().begin())>;
    std::is_same_v<decltype(t().begin()), std::string_view>;
};

template<typename T, typename K>
concept CallableGivesIterableKeyedStringViews = requires(T t, K k)
{
    t();
    std::input_iterator<decltype(t().begin())>;
    std::is_same_v<decltype(t().begin()), KeyedStringView<K>>;
};
```

Map to Table

```
template<typename TKey>
static constexpr auto MapToStrings(
    CallableGivesIterableKeyedStringViews<TKey> auto f
    ) -> CallableGivesIterableStringViews auto
{
    return [=]() {
        constexpr auto stringmap = f();
        constexpr auto NumStrings =
            std::distance(stringmap.begin(), stringmap.end());

        std::array<std::string_view, NumStrings> result;
        std::size_t idx{0};
        for(auto const &v : stringmap) {
            result.at(idx++) = v.Value;
        }
        return result;
    };
}
```

Modularizing Code

- Need to calculate compile-time constants from user data
- Can't pass from one function to another as parameters - not constant
- The user supplied lambda needs re-evaluation in each context
- Can define local lambdas to help but results in long methods
- Still results in large chunks of code, can't see a better way

```cpp
static constexpr auto BuildHuffmanTree(
    CallableGivesIterableStringViews auto makeStringsLambda)
{
    constexpr auto st = makeStringsLambda();
    // ...
    return tree;
}

static constexpr auto MakeEncodedBitStream(
    CallableGivesIterableStringViews auto makeStringsLambda)
{
    constexpr auto st = makeStringsLambda();
    constexpr auto NumStrings = std::distance(st.begin(), st.end());

    constexpr auto tree = BuildHuffmanTree(makeStringsLambda);

    ResultType<NumStrings> stream;
    // fill output stream...
    return stream
}
```

```
constexpr auto charLookup = MakeCharacterLookupTable();

constexpr auto CalculateStringLength =
    [=](std::string_view s) -> std::size_t {
    std::size_t len{0};
    for(char const c : s) {
        len += charLookup.at(static_cast<std::size_t>(c)).BitLength;
    }
    return len;
};

constexpr auto CalculateEncodedStringBitLengths = [=]() {
    std::array<std::size_t, NumStrings> result;
    std::size_t i{0};
    for(auto const &s : st) {
        result.at(i) = CalculateStringLength(s);
        ++i;
    }
    return result;
};

constexpr auto stringLengths = CalculateEncodedStringBitLengths();
constexpr auto totalEncodedLength =
    std::accumulate(stringLengths.begin(), stringLengths.end(), 0);
```

Using <algorithm>s

- Able to use most algorithms, eg:
  - std::sort
  - std::push_heap / std::pop_heap
  - std::count_if
  - std::difference
  - std::accumulate

But not containers

- except std::array
- std::vector waiting for compiler support
- had to build my own
  - priority_queue
  - list
  - bit_stream (std::bitset?)
- More constexpr containers please!

Heap Allocations

- Can allocate heap as long as it is freed before leaving context
- Means you can't return it, even to another constexpr context
- Used list implementation as a queue for breadth first tree traversal when laying out nodes in output array
- Was easier to count required nodes and allocate std::array to hold all nodes making the tree (std::vector please!)
- More container support would make this much easier

Limits

- The compilers choose an arbitrary amount of work they will allow in constexpr context
- Complex processing like compression will hit the limits
- Had to make more complex implementation to cache bit streams rather than walk tree for each character
- Still possible to hit limits on large strings
- -fconstexpr-ops-limit=VERY_BIG_NUMBER

USB descriptors

Lets make a USB Configuration descriptor

- https://github.com/AshleyRoll/cpp_usbdescriptor
- Partial implementation so far
- Simple Reference for USB protocol https://www.beyondlogic.org/usbnutshell/usb1.sht
- This is a brief overview, check out the code for implementation details
- Most complex example - uses variadic templates, std::tuple and constexpr functions to deal with different sized interfaces

Configuration Descriptor

- Tells the host about interfaces and endpoints
- Interfaces make "functionality"
- Endpoints are data transfers addresses
- Variable number of interfaces
- Each interface has variable number of endpoints

```
constinit auto const Descriptor1
 = usb::descriptor::MakeConfigurationDescriptor([]() {
    using namespace usb::descriptor;

    return Configuration{
            1,      // config number
            3,      // string identifier
            false,  // selfPowered
            false,  // remoteWakeup
            100,    // 200mA (units of 2mA)
            define_vendor_specific_interface(
                1,
                BulkEndpoint{EndpointDirection::Out, 1, 512},
                InterruptEndpoint{EndpointDirection::In, 1, 512, 1}
            ),
            define_vendor_specific_interface(
                2,
                BulkEndpoint{EndpointDirection::Out, 1, 512},
                BulkEndpoint{EndpointDirection::In, 1, 512},
                BulkEndpoint{EndpointDirection::In, 2, 512},
                BulkEndpoint{EndpointDirection::In, 3, 512}
            )
    };
});
```

```
$ objdump -s --section=.descriptor usbdescriptors

usbdescriptors:     file format elf64-x86-64

Contents of section .descriptor:
 2020 09024500 02010380 64090400 0002ffff  ..E.....d.......
 2030 ff010705 01020002 00070581 03000201  ................
 2040 09040000 04ffffff 02070501 02000200  ................
 2050 07058102 00020007 05820200 02000705  ................
 2060 83020002 00
```

- Placed `Descriptor1` into `.descritor` section to help dump it

```cpp
template<std::size_t ... Sizes>
class Configuration
{
    constexpr Configuration(
        std::uint8_t  configurationNumber,
        std::uint8_t stringIdentifier,
        bool selfPowered,
        bool remoteWakeup,
        std::uint8_t maxPower_2mAUnits,
        Interface<Sizes>... interfaces
    )
    { /* ... */ }
};
```

```
template<std::size_t NumEndpoints>
class Interface
{
    constexpr Interface(
        std::uint8_t interfaceClass, std::uint8_t interfaceSubClass,
        std::uint8_t interfaceProtocol, std::uint8_t stringIdentifier,
        std::array<Endpoint, NumEndpoints> endpoints
    )
    { /* ... */ }
};

class Endpoint
{
    // Endpoint is a literal type, no need to have a builder
    constexpr Endpoint(
        EndpointDirection direction, std::uint8_t address,
        EndpointTransfer transfer,
        EndpointSynchronisation synchronisation,
        EndpointUsage usage, std::uint16_t maxPacketSize,
        std::uint8_t interval
    ) { /* ... */ }
};

// derived class for BulkEndpoint, InterruptEndpoint etc...
```

```cpp
// Specific interface type helper function
template<typename ... TEPs>
constexpr auto define_vendor_specific_interface(
    std::uint8_t stringIdentifier,
    TEPs ... endpoints)
{
    return define_interface(
        0xFF, 0xFF, 0xFF, stringIdentifier,
        endpoints...);
}

// General Helper functions to create interface
template<typename ... TEPs>
constexpr auto define_interface(
    std::uint8_t interfaceClass, std::uint8_t interfaceSubClass,
    std::uint8_t interfaceProtocol, std::uint8_t stringIdentifier,
    TEPs ... endpoints)
{
   return Interface<sizeof...(TEPs)>{
       interfaceClass, interfaceSubClass,
       interfaceProtocol, stringIdentifier,
       std::array<Endpoint, sizeof...(TEPs)>{ endpoints... }
   };
}
```

Rendering the descriptor

- Each class has a length() method
- Each class has a render() method taking std::span
- the MakeConfigurationDescriptor() function then:
  - Calculates the required buffer size
  - calls the render method to fill an array
- Use std::tuple of interfaces of varing number of endpoints
- Use std::apply to fold over tuple for length/render

```cpp
// Concept to enforce a lambda
template<typename T, std::size_t ... Sizes>
concept CallableGivesConfiguration = requires(T t)
{
    t();            // is callable
    // result is a Configuration<...>
    std::is_same_v<decltype(t()), Configuration<Sizes...>>;
};

// Rendering helper, pass in the lambda to generate the configuration
template<std::size_t ... Sizes>
consteval static auto MakeConfigurationDescriptor(
    CallableGivesConfiguration<Sizes...> auto makeConfigLambda)
{
    // build the configuration using the supplied lambda
    constexpr auto cfg = makeConfigLambda();
    constexpr auto len = cfg.length();

    std::array<std::uint8_t, len> data;

    cfg.Render(data);

    return data;
}
```

```cpp
template<std::size_t ... Sizes>
class Configuration
{
    constexpr std::size_t length() const
    {
        return ConfigurationDescriptorSize + std::apply(
            [](auto && ... interfaces)
            {
                return (0 + ... + interfaces.length());
            },
            m_Interfaces
        );
    }

    std::tuple<Interface<Sizes>...> m_Interfaces;
};
```

```
template<std::size_t ... Sizes>
class Configuration
{
    constexpr void Render(std::span<std::uint8_t> buffer) const
    {
        // ... Render fixed Configuration data ...
        std::size_t location{ConfigurationDescriptorSize};
        std::apply(
            [&](auto && ... interfaces) {
                std::size_t index{0};
                auto render = [&](auto i) {
                    auto len = i.length();
                    i.Render(buffer.subspan(location, len), index);
                    location += len;
                };

                ((render(interfaces)), ...);
            }, m_Interfaces);
        // store lenght data into fixed Configuration section
        impl::write_le(buffer.subspan<2, 2>(), location);
    }
};
```

Acknowledgements

- Members of the #include<C++> discord community for being so welcoming and helpful!
- Jason Turner for so much help and advice
    - and encouraging me to submit this talk

Questions?

https://github.com/AshleyRoll

#include<C++> @AshleyRoll

Cpplang @AshleyRoll

@AshleyJRoll