`std::span` in C++

A Very Simple Vocabulary Class … But?

Non-Type Template Parms!
Ranges and Views and Iterators, Oh My!
Compile Time vs Run Time Elements!
A Turing Complete Compile Time Language!

An Exploration of C++ Principles

Cliff Green

NM C++ Programmers Meetup, Jan 31, 2024
Google Developers Group, Albuquerque, Mar 7, 2024

- A small subset of C++ principles (this is a short presentation)
- Discuss how these principles compare to other languages
- Not "fanboy" discussions, since computer languages are tools (use the right tool for the task)
- There will be gratuitous photos (as in all of my presentations)

Theme is Triangle Mosaic Template by PresentationGO, https://www.presentationgo.com

My Wife’s First Glider Ride (1/1/2024)

What is `std::span`?

- `std::span` is a C++ standard library class template that models:
  - An iterator to a sequence of elements
  - Number of elements (size)
- It is a “view type” versus a “owning type”
  - Lifetime of the element sequence is outside of `std::span` object lifetime
  - View objects are lightweight and refer to other objects

What is a primary use case for `std::span`?

- `std::span` replaces the C style function interface of a pointer to an array (buffer), and a size:

```
void func1a (double* buf, int num_elems); // C style
void func1b (std::span<double> buf); // C++ version
int func2a (const int* buf, std::size_t num_elems);
int func2b (std::span<const int> buf);
```

- `std::span` has `data()`, index `operator[]`, `size()`, etc. methods for accessing elements, size

My Son’s First Glider Ride (12/23/2018)

What is an iterator?

- In C++ (and similar to other languages) an iterator allows generic traversal of containers (and ranges)
  - Syntax is modeled on pointers (dereference, increment, etc)
  - Iterators are the “glue” between algorithms and data structures
- For example, the `std::find_if` algorithm works on (almost?) any container - `std::vector`, `std::list`, `std::deque`, `std::map`, etc.
  - Algorithms take a begin and end iterator (a “range”) and traverse the container using dereference, increment, etc operators

Why is this “iterator glue” so valuable?

- It allows algorithms and containers to be independent of each other; i.e. there is not a separate `std::find_if` algorithm for a vector versus a list versus a map
  - Internally, traversing a vector versus a list versus a map is completely different code; iterators abstract this traversal
- Generics in C++ allow this “iterator glue” design using `template` syntax
  - Other languages have generic iterators using different syntax

So a `std::span` object contains an iterator?

- Yes, but a constraint on `std::span` is the iterator must point to elements in ***contiguous memory*** - i.e. a “C style” array
- This allows the iterator to be a pointer …
- Which makes a `std::span` object very lightweight, only two (or one! … more on that coming up) elements
  - Minimal memory
  - Easy to pass in registers
  - Easy to optimize by the compiler

My Friend’s First Glider Ride (5/28/2023)

What is the `std::span` class template declaration?

```
template <
    typename T,
    std::size_t Extent =
                std::dynamic_extent >
  class span;
```

Okay, “`typename T`” might be obvious to some (basic generic programming in C++), but …

What is this “`std::size_t Extent`” syntax?

- `std::span Extent` Is a “non-type” template parameter
- E.g. the C++ `std::array` class template is declared as:

```
template<
   class T,
   std::size_t N
> struct array;
```

- The “N” is a compile time size of the array

What are the implications of `std::array` size?

- `std::array<int, 10> my_array;` means the size (10) is encoded as part of the type; everywhere `my_array` is used the compiler knows the size
- `std::array<Person, 12>` and `std::array<Person, 20>` are two different types, even though they both contain `Person` objects
- The size of a `std::array` object is not stored anywhere at runtime; no memory is allocated for it

How does `std::array` size relate to `std::span` size?

- `std::array` is declared to always have a compile time size (with the size non-type template parameter); if dynamic / runtime size is needed `std::vector` is used
- A `std::span` is designed to be used with both fixed / compile time sizes as well as dynamic / runtime sizes
- Ahh! Now we're back to knowing what that second template parameter is for `std::span`

Can you show examples of this `std::span` size flexibility?

- You betcha! Given a `std::vector<int> vec_int; std::array<float, 10> fl_arr; double c_arr[20];` (C style array), and a `std::string str;`:

```
std::span<int>(vec_int); // dynamic extent
std::span<float, 10>(fl_arr); // fixed ext, sz 10
std::span<double, 20>(c_arr); // fixed ext, sz 20
std::span<char>(str); // dynamic extent
```

How about another example showing `std::span` compile time checks?

- Sure! Given a function `sum3` that requires three ints (e.g. 3D coords) from an array for indexing, `std::span` will verify ***at compile time*** (versus a runtime error) that there are three elements in contiguous memory

```
constexpr int sum3 (std::span<int, 3> sp) {
    return sp[0] + sp[1] + sp[2];
}
```

How can the `sum3` function be called?

- The `sum3` function can be called with a C-style array of 3 elements, a `std::array<int, 3>` or a `std::vector<int>` where a view of exactly 3 elements have been selected

```
auto a = sum3(std::span<int, 3>(vec.begin(), 3));
auto b = sum3(arr); // arr is std::array<int, 3>
auto c = sum3(c_arr); // c_arr is C-style array, 3 elems
```

My Son’s Second Glider Ride (9/12/2022)

Who cares if the size is not stored at runtime, does it really make a difference?

- Very often (most of the time?) it doesn't
- But … when it does, it can make a difference - need a million `std::span` objects? Or are you creating a `std::span` a million times a second? Is this running on an embedded system with limited memory?
- By providing compile time / static facilities, C++ (and similar facilities in other languages) can make code faster and more efficient (memory, power usage, etc)

Okay, why doesn't everyone use C++?

- C++ is large and complicated
- C++ is large and complicated (specially some of the syntax) and has a larger learning curve than a lot of other languages
- C++ is large and complicated and nobody (including Bjarne Stroustrup, the inventor of C++) knows everything about the language
- C++ is large and complicated and has it's dangerous aspects (much of it inherited from C) - "undefined behavior" can occur in many different ways

C++ generics actually provide a Turing complete facility inside the compiler?

```
template <int N>
consteval int factorial () {
    if constexpr (N < 2) // style: normally braces on ifs
        return 1;
    else
        return factorial<N - 1>() * N; // recursive call
}

int main () {
   return factorial<6>(); // 720
}
```

The factorial result is computed at compile time?

- Yes, this is the assembler output (Compiler Explorer, x86-64, gcc 13.2) - note the “720”:

```
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, 720
        pop     rbp
        ret
```

So all C++ static compile time facilities require convoluted recursive code?

- No, many of the C++ compile time features are easy and straightforward
- E.g. examine a type (using “type traits” facilities) to determine at compile time whether it can be copied with “memcopy” versus deeper object copying
- When everything - types, code, evaluation, etc - is performed at runtime, the language can be simple and flexible (e.g. interpreter evaluation of each line)

My Business Partner Wife’s First Glider Ride (1/1/2024)

Questions, comments, discussions?

- Only certain principles of the C++ language have been touched on - the most important and fundamental aspect of C++ (to me) has not been mentioned in this presentation: deterministic, controlled, and abstracted construction and destruction of objects
- Some aspects of C++ development are also irritating (to me) - complex build management (specially dependencies), extra considerations for multi-platform development, slower build times versus other languages

Thank You!

I hope you've enjoyed this presentation!

Dave Steffen (thanks!) provided additional example code which has been added to these slides

As usual, "go to" utilities include:

- CPP Reference page: https://en.cppreference.com/
- Compiler Explorer: https://godbolt.org/ - compile and run your code on multiple compilers, analyze the assembler output

Cliff Green
206-915-4382