Are you structure bindable?

A tour via concepts, SFINAE and Herb Sutter's pattern matching

Amir Kirsh

About me

Lecturer
Academic College of Tel-Aviv-Yaffo
Tel-Aviv University

Member of the Israeli ISO C++ NB

Co-Organizer of the **CoreCpp** conference and meetup group

Developer Advocate at

INCREDIBUILD

INCREDIBUILD

Suffering from slow builds?

It’s not just waste of time

It affects your dev cycles
and productivity

Structure Bindable (C++17)

https://en.cppreference.com/w/cpp/language/structured_binding

- std::tuple and tuple-like types, providing
  - member-wise get or specialized version of std::get
  - specialized version of std::tuple_size<Type>
  - specialized version of std::tuple_element<Type>
- C-style arrays
- struct with all public fields

Structure Bindable - Tuple-Like types

```
// case 1a - std::tuple
std::tuple t {1, "hello", 2.5};

auto& [a1, b1, c1] = t;
a1 = 3; // we actually change t

auto [a2, b2, c2] = t;
a2 = 3; // t is not changed
```

Structure Bindable - Tuple-Like types

```
// case 1b - std::pair
std::pair p {1, "hello"};

auto& [a1, b1] = p;
a1 = 3; // we actually change p

auto [a2, b2] = p;
a2 = 3; // p is not changed
```

Structure Bindable - Tuple-Like types

```
// case 1c - std::array
std::array arr {1, 2, 3};

auto& [a1, b1, c1] = arr;
a1 = 3; // we actually change arr

auto [a2, b2, c2] = arr;
a2 = 3; // arr is not changed
```

Structure Bindable - other Tuple-Like types

```
// case 1d - user defined tuple-like class, providing:
// - member-wise get or specialized version of std::get
// - specialized version of std::tuple_size<Type>
// - specialized version of std::tuple_element<Type>
```

See: https://godbolt.org/z/nGqPT3fEK

A *somehow related* proposal (though, not directly in the scope of this talk) –
"Compatibility between tuple, pair and tuple-like objects":
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2165r2.pdf

Structure Bindable - C-Style Array

```
// case 2 - C-style array
int arr[3][2] = {{1, 2}, {3, 4}, {}};

// yes, you can bind to a multidimensional array!
auto& [a1, b1, c1] = arr;
a1[0] = 3; // arr is changed!

auto [a2, b2, c2] = arr;
a2[0] = 3; // arr is NOT changed! (we hold a copy of the array)
```

Structure Bindable - struct

```
// case 3 - struct with public only fields
struct st {
    int a;
    std::string b;
    double c;
};
```

```
st s {1, "hello", 2.5};
auto& [a1, b1, c1] = s;
a1 = 3; // s is changed

auto [a2, b2, c2] = s;
a2 = 3; // s is NOT changed
```

Structure Bindable Concept (using C++20)

The Challenge:

```
void foo(Twople auto&& twople) {
    auto [a, b] = std::forward<T>(twople);
    …
}
```

Structure Bindable Concept (using C++20)

First Attempt

```
template<typename T>
concept Twople = requires(T t) {
    auto [a, b] = t;
};

void foo(Twople auto&& twople) { … }
```

Code: https://godbolt.org/z/5r3YoTYa5

Structure Bindable Concept (using C++20)

Second Attempt

```
template<typename T>
concept Twople = requires {
    ( [] (T&& t) consteval {
        auto [a,b] = std::forward<T>(t);
        return a;
    } ); // wrapping with curly brackets behaves the same
};
```

Code: https://godbolt.org/z/s3EjrreTx

Structure Bindable Concept (using C++20)

Covering all cases *except structs*, by requiring *T* to be either:

- std::tuple / tuple-like, providing
  - member-wise get or specialized version of std::get
  - specialized version of std::tuple_size<Type>
  - specialized version of std::tuple_element<Type>
- C-style array

Code:

https://godbolt.org/z/866z1xTj5

Structure Bindable Concept (compiler specific)

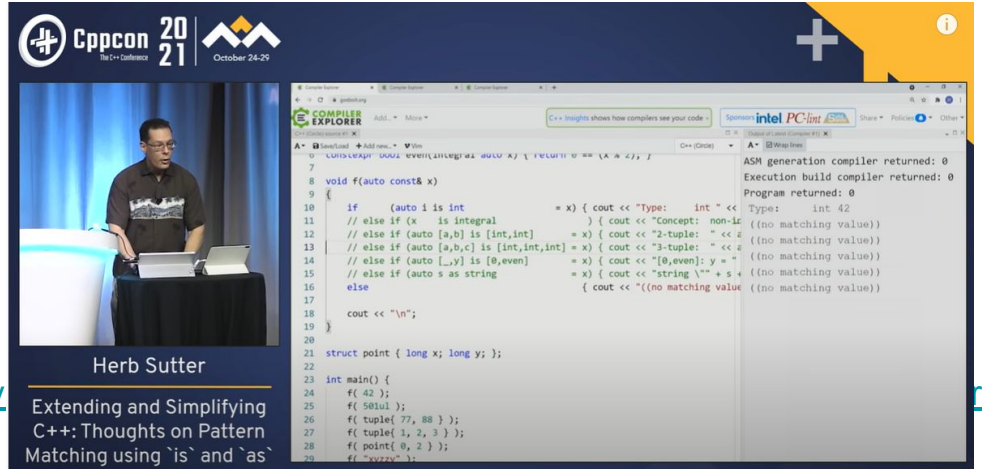
A side note: it can be achieved with compiler intrinsic features

For example - for clang (courtesy of Avi Lachmish):

https://godbolt.org/z/vEhbres3G

Pattern matching using `is` and `as`

A proposal by Herb Sutter: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2392r1.pdf



Extending and Simplify … rb Sutter CppCon 2021

Pattern matching using `is` and `as`

```
void foo(const auto& x) {
    if(auto s is std::string = x) {
        cout << s[0] << endl;
    }
    else if(auto[a, b] is [int, int] = x) {
        cout << a << ' ' << b << endl;
    }
}
```

```
int main() {
    foo("hello");
    foo("world"s);
    foo(std::tuple{1, 2});
}
```

Code:
https://circle.godbolt.org/z/c35dMKbb3

Structure Bindable Concept (C++2x=> 26?)

Using pattern matching – `as`

```
template <typename T>
concept structured_bindable = requires (T t) {
    t as [...];
};
```

Structure Bindable Concept (C++2x=> 26?)

```
template <typename T> struct single_element_structured_bindable_helper {
    auto first() {
        auto[a, ...] = std::declval<T>(); return a;
    }
    using first_type = decltype(first());
};

template <typename T>
concept single_element_structured_bindable = structured_bindable<T>
    && requires (T t) {
    {t as [single_element_structured_bindable_helper<T>::first_type]};
};
```

Structure Bindable Concept (C++2x=> 26?)

```
template <typename T> struct two_elements_structured_bindable_helper {
    auto first() {
        auto[a, ...] = std::declval<T>(); return a;
    }
    auto second() {
        auto[a, b, ...] = std::declval<T>(); return b;
    }
    using first_type = decltype(first());
    using second_type = decltype(second());
};
```

Structure Bindable Concept (C++2x=> 26?)

```
template <typename T>
concept two_elements_structured_bindable = requires (T t) {
    {t as [
            two_elements_structured_bindable_helper<T>::first_type,
            two_elements_structured_bindable_helper<T>::second_type
          ]};
};
```

Structure Bindable Concept (C++2x=> 26?)

```
template <typename T>
concept Twople =
    structured_bindable<T>
    && !single_element_structured_bindable<T>
    && two_elements_structured_bindable<T>;
```

Code: https://circle.godbolt.org/z/931TP8q8r

A note, to save you redundant trials…

auto would not come to your rescue…

This doesn't work:

```
template <typename T>
concept Twople = requires (T t) {
    t as [auto, auto];
};

void foo(Twople auto&& t) {}

foo(std::tuple{1, 2}); // fails
```

Code: https://circle.godbolt.org/z/nK4PMqhGf

Still, it can be made much easier…

indeed, *auto* doesn't work

But the below does! (proposed by Dvir Yitzchaki):

```
struct anything { // note, std::any doesn't do the trick
    template<typename T>
    anything(T&&) {}
};

template<typename T>
concept twople = requires(T t) {
    t as [anything, anything];
};
```

Code: https://circle.godbolt.org/z/jh87vYejj

Wait… we are not done yet!

Can we have a generic:

```
structured_bindable_with<2>
```

Structure Bindable Concept (C++2x=> 26?)

Based on p1858 or a related proposal:

```
template <typename T, size_t SIZE>
concept structured_bindable_with = (sizeof...(T) == SIZE);
```

Code: https://circle.godbolt.org/z/nh7Wqh5bo

Reference

Main resource:

Stack Overflow: How to define a concept of a object that is can be structured binding?

Thanks:

Dvir Yitzchaki, for contributing many of the ideas behind this talk.

Additional related resources:

C++20 Concept to check tuple-like types

Is the body of requires block unevaluated context?

Using concepts in an unevaluated context

Why are lambda expressions not allowed in an unevaluated operands

C++ primary expressions

3 final things before we depart…

Core C++ 2022: https://corecpp.org

Tel-Aviv, 5th to 7th of September

Registration is Open


CPP Game Show
from Incredibuild
25:00
C++
CPP GAME SHOW
Amir
Daisy
Dima
Vittorio
Kilian

In case you still don’t have our T-Shirt 😯

(Are you a Ruster in disguise?)

Go get it now!

Thank you!

```
void conclude(auto greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });
```