Add tuple protocol to complex

Abstract
This paper proposes amending complex with the tuple protocol, enabling structured binding and easy referential access.

Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex&lt;double&gt; c{...};</td>
<td>complex&lt;double&gt; c{...};</td>
</tr>
<tr>
<td>auto &amp; [r, i]{reinterpret_cast&lt;double(&amp;)[2]&gt;(c);}</td>
<td>auto &amp; [r, i]{c};</td>
</tr>
<tr>
<td>template&lt;typename T&gt; constexpr auto swap_parts(complex&lt;T&gt; c) -&gt; complex&lt;T&gt; {</td>
<td>template&lt;typename T&gt; constexpr auto swap_parts(complex&lt;T&gt; c) -&gt; complex&lt;T&gt; {</td>
</tr>
<tr>
<td>if not consteval {</td>
<td>auto &amp; [r, i]{};</td>
</tr>
<tr>
<td>auto &amp; [r, i] {reinterpret_cast&lt;double(&amp;)[2]&gt;(c);};</td>
<td>swap(r, i);</td>
</tr>
<tr>
<td>} else {</td>
<td>return c;</td>
</tr>
</tbody>
</table>
| //reinterpret_cast is ill-formed in constexpr... const auto tmp{c.real()}; | }
| c.real(c.imag()); | c.imag(tmp); |
| return c; } | return c; } |

vector<complex<double>> v{ ... }; vector<complex<double>> v{ ... };
| auto reals(v | views::transform([](auto c) { return c.real(); }) | auto reals(v | views::elements<>& | ranges::to<vector>(); |
| } | ranges::to<vector>(); |
| auto imgs(v | views::transform([](auto c) { return c.imag(); }) | auto imgs(v | views::elements<1> | ranges::to<vector>(); |
| } | ranges::to<vector>(); |

complex<double> c{...}; complex<double> c{...};

//interaction with pattern matching proposal P1371R3
//interaction with pattern matching proposal P1371R3
inspect(reinterpret_cast<double(&)[2]>(c)) {
| [0, 0] => { cout << "on origin"; } |
| [0, i] => { cout << "on imaginary axis"; } |
| [r, 0] => { cout << "on real axis"; } |
| [r, i] => { cout << r << ", " << i; } |
| };

//interaction with pattern matching proposal P2392R2
//interaction with pattern matching proposal P2392R2
inspect(reinterpret_cast<double(&)[2]>(c)) {
| is [0, 0] => cout << "on origin"; |
| is [0, _] => cout << "on imaginary axis"; |
| is [_, 0] => cout << "on real axis"; |
| [r, i] is _ => cout << r << ", " << i; |
| };

---

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

2 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, christoph.hofer@risc-software.at
Revisions
R0: Initial version

R1: Changes after LEWG review on 2023-06-12:
- Made `get` overloads hidden friends.
- Extending `tuple-like` concept to support tuple-based range algorithms.
- Amended proposed wording with entry to Annex C.

R2: Changes after LWG review on 2023-11-08:
- Removed the dedicated feature test macro.
- Made `get` free functions again as hidden friends do not work for `tuple-like`.
- Removed wording for Annex C.

Motivation
Mathematically the set of complex numbers \( \mathbb{C} \) is isomorphic to \( \mathbb{R}^2 \) as a vector space with the isomorphism \( \Phi: \mathbb{C} \rightarrow \mathbb{R}^2 \) such that \( \Phi(a+bi) = (a,b) \). Therefore, complex numbers can be identified with tuples and should possess the same characteristics, which is covered by the tuple protocol.

Complex numbers can equivalently be represented in cartesian coordinates \((a,b)\) as well as in polar coordinates \((r,\theta)\) using radius \(r\) and angle \(\theta\). However, alternative representations of complex numbers such as polar coordinates \((r,\theta)\) are prohibited by the requirement of matching C’s `_Complex` floating-point feature.

As the respective getters do not expose referential access (changing them to do so would result in an ABI-break), the only way to get a reference to the real and imaginary parts of a complex is by performing a `reinterpret_cast` (mandated to be valid, see [complex.numbers.general]), which is not valid in a `constexpr` context. Supporting the tuple protocol enables structured binding and referential access to the components of a complex number in a `constexpr` compatible way.

Lastly, the current pattern matching proposals ([P1371R3] and [P2392R2]) allow inspection of `tuple-like` objects, the proposed changes make complex `tuple-like`.

Design Space
The tuple protocol traits (`tuple_size<T>` and `tuple_element<I, T>`) are partially specialized for `complex<U>` and four function overloads of `get` are provided. Additionally, the exposition-only `tuple-like` concept is amended, enabling support for range algorithms like `views::elements`.

Impact on the Standard
This proposal is a library extension, that changes the meaning of `tuple-like<complex<T>>`.

Implementation Experience
The proposed design has been implemented at [https://github.com/MFHava/STL/tree/P2819](https://github.com/MFHava/STL/tree/P2819).

Proposed Wording
Wording is relative to [N4950]. Additions are presented like this, removals like this and drafting notes like this.

[version.syn]
```cpp
#define __cpp_lib_tuple_like 202207L //also in <utility>, <tuple>, <map>, <unordered_map>
```
**[tuple.like]**

```cpp
???.?? Concept tuple-like

template<class T>
    concept tuple-like = see below; // exposition only

1 A type T models and satisfies the exposition-only concept tuple-like if remove_cvref_t<T> is a specialization of array, complex, pair, tuple, or ranges::subrange.
```

**[complex.numbers]**

```cpp
???.?? Header <complex> synopsis

namespace std {
    
    // [complex.transcendentals], transcendental
    
    template<class T> complex<T> tanh (const complex<T>& x);

    // [complex.tuple], tuple interface
    
    template<class T> struct tuple_size,
        struct_tuple_size;
    template<size_t I, class T> struct tuple_element;
    template<class T> struct tuple_size<complex<T>> : integral_constant<size_t, 2> {};
    template<size_t I, class T> struct tuple_element<I, complex<T>> {
        using type = T;
    };

    template<class T> complex<T> tanh (const complex<T>& x);

27 Returns: The complex hyperbolic tangent of x.

???.?? Tuple interface

```cpp
template<class T>
    struct tuple_size<complex<T>> : integral_constant<size_t, 2> {};

```cpp
template<size_t I, class T>
    struct tuple_element<I, complex<T>> {
        using type = T;
    };

1 Mandates: I < 2 is true.

```cpp
template<size_t I, class T>
    constexpr T& get(complex<T>& z) noexcept;
    constexpr T&& get(complex<T>&& z) noexcept;
    constexpr const T& get(const complex<T>& z) noexcept;
    constexpr const T&& get(const complex<T>&& z) noexcept;
```

2 Mandates: I < 2 is true.

3 Returns: A reference to the real part of z if I == 0 is true, otherwise a reference to the imaginary part of z.

???.?? Additional overloads

```cpp
```
Acknowledgements
Thanks to RISC Software GmbH for supporting this work.