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>
</tbody>
</table>
| template<typename T>
constexpr auto swap_parts(complex<T> c) -> complex<T> |
if not consteval {
  auto & [r, i]{reinterpret_cast<double(&)[2]>(c);};
  swap(r, i);
} else {
  //reinterpret_cast is ill-formed in constexpr...
  const auto tmp{c.real()};
  c.imag(tmp);
} |
| |
| return c; |
| |
| vector<complex<double>> v( _ ); | vector<complex<double>> v( _ ); |
| auto reals(v | views::transform([](auto c) { return c.real(); }) | ranges::to<vector>()) | auto reals(v | views::elements<0> | ranges::to<vector>()) |
| auto imgs(v | views::transform([](auto c) { return c.imag(); }) | ranges::to<vector>()) | auto imgs(v | views::elements<1> | ranges::to<vector>()) |
| |
| complex<double> c(...); |
| //interaction with pattern matching proposal P1371R3
inspect(reinterpret_cast<double(&)[2]>(c)) |
| is [0, 0] => { cout << "on origin"; }
| is [0, 1] => { cout << "on imaginary axis"; }
| [r, 0] => { cout << "on real axis"; }
| [r, 1] is _ => cout << r << ", " << i; |
| //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, 1] is _ => cout << r << ", " << i; |
| //interaction with pattern matching proposal P1371R3
inspect(c) |
| is [0, 0] => { cout << "on origin"; }
| is [0, 1] => { cout << "on imaginary axis"; }
| [r, 0] => { cout << "on real axis"; }
| [r, 1] is _ => cout << r << ", " << i; |
| //interaction with pattern matching proposal P2392R2
inspect(c) |
| is [0, 0] => cout << "on origin";
| is [0, _] => cout << "on imaginary axis";
| is [_, 0] => cout << "on real axis";
| [r, 1] 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.

Motivation
Mathematically the set of complex numbers $\mathbb{C}$ is isomorphic to $\mathbb{R}^2$ as a vector space with the isomorphism $\Phi: \mathbb{C} \to \mathbb{R}^2$ 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 $\_\text{Complex}$ 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 hidden friend function overloads of get are provided. Additionally, the exposition-only tuple-like concept is amended, enabling support for range algorithms like std::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.

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

[version.syn]
#define __cpp_lib_complex_tuple YYYYMML //also in <complex>
#define __cpp_lib_tuple_like YYYYMML //also in <utility>, <tuple>, <map>, <unordered_map>

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]
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`.

---

### Header `<complex>` synopsis

```cpp
namespace std {
    // [complex.transcendentals], transcendentals
    template<class T> complex<T> tanh (const complex<T>&);
    // [complex.tuple], tuple interface
    template<class T> struct tuple_size;
    template<size_t I, class T> struct tuple_element;
    template<class T> struct tuple_size<complex<T>>;
    template<size_t I, class T> struct tuple_element<I, complex<T>>;
    // [complex.literals], complex literals
}
```

### Class template `complex`

```cpp
namespace std {
    template<class T> class complex {
        public:
            using value_type = T;
            constexpr complex(const T& re = T(), const T& im = T());
            constexpr complex(const complex&) = default;
            template<class X> constexpr explicit(complex<X>& x);  // see below
            constexpr T real() const;
            constexpr void real(T);
            constexpr T imag() const;
            constexpr void imag(T);
            constexpr complex& operator= (const T&);
            constexpr complex& operator+=(const T&);
            constexpr complex& operator-=(const T&);
            constexpr complex& operator*=(const T&);
            constexpr complex& operator/=(const T&);
            template<class X> constexpr complex& operator= (const complex<X>&);
            template<class X> constexpr complex& operator+=(const complex<X>&);
            template<class X> constexpr complex& operator-=(const complex<X>&);
            template<class X> constexpr complex& operator*=(const complex<X>&);
            template<class X> constexpr complex& operator/=(const complex<X>&);

            template<size_t I> friend constexpr T& get(complex&) noexcept;
            template<size_t I> friend constexpr T&& get(complex&&) noexcept;
            template<size_t I> friend constexpr const T& get(const complex&) noexcept;
            template<size_t I> friend constexpr const T&& get(const complex&&) noexcept;
    };
}
```

### Non-member operations

```cpp
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

---

3
Effects: Inserts the complex number $x$ onto the stream $o$ as if it were implemented as follows:

```cpp
basic_ostringstream<charT, traits> s;
s.flags(o.flags());
s.imbue(o.getloc());
s.precision(o.precision());
s << '(' << x.real() << ',' << x.imag() << ')';
return o << s.str();
```

Note: In a locale in which comma is used as a decimal point character, the use of comma as a field separator can be ambiguous. Inserting `showpoint` into the output stream forces all outputs to show an explicit decimal point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously. — end note

```cpp
template<size_t I>
friend constexpr T& get(complex& x) noexcept;
template<size_t I>
friend constexpr const T& get(const complex& x) noexcept;
template<size_t I>
friend constexpr T&& get(complex&& x) noexcept;
template<size_t I>
friend constexpr const T&& get(const complex&& x) noexcept;
```

Mandates: $I < 2$ is true.

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

???.?? Value operations

... ???.?? Transcendentals

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

Returns: The complex hyperbolic tangent of $x$.

???.?? Tuple interface

```cpp
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;
};
```

Mandates: $I < 2$ is true.

???.?? Additional overloads

Acknowledgements

Thanks to RISC Software GmbH for supporting this work.