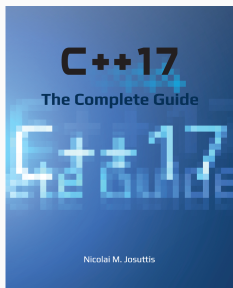
Fun with Structured Bindings

Tim van Deurzen

November 15, 2019

Freelance Software Engineer at Eolas Engineering

C++17 The Complete Guide - Nicolai M. Josuttis



Step 1 - Define the number of values

```
namespace std {
  template <>
  struct tuple_size<MyDecomposableType> :
      std::integral_constant<std::size_t, 2> {};
}
```

Step 2 - Define the types of those values

```
namespace std {
  template <std::size_t N>
  struct tuple_element<N, MyDecomposableType> {
    using type = decltype(
      std::declval<MyDecomposableType>().template get<N>()
    );
  };
}
```

Step 3 - Get or compute each value

```
struct MyDecomposableType {
    template <std::size_t N>
    auto get() const noexcept {
        return 42;
    }
};
```

Using your new structured bindings

```
int main() {
  auto [a, b] = MyDecomposableType{};
}
```

Becomes:

```
// from cppinsights.io:
int main() {
  MyDecomposableType __24 = MyDecomposableType{};
  std::tuple_element<0, MyDecomposableType>::type a =
      __24.get<0>();
  std::tuple_element<1, MyDecomposableType>::type b =
      __24.get<1>();
}
```

Some Ideas for Experiments

- Decompose an integral value into its separate bytes.
- Access the components of an IEEE float or decompose NaN-boxed values into type and value.
- Golang-like error handling:

```
auto silly_function() -> std::expected<T, E> { /* ... */}
// ...
void test() {
  // ...
  if (auto [value, error] = silly_function(); !error) {
    // ...
  }
}
```

BytesOf - Example

```
void test {
  auto [a, b, c, d] = BytesOf(0x04030201);
  // a = 0x01
  // b = 0x02
  // c = 0x03
  // d = 0x04
}
```

BytesOf - Implementation

```
template <typename T>
struct BytesOf {
  T m_value;

  BytesOf(T value) : m_value{value} {}

  template <std::size_t N>
  uint8_t get() const noexcept {
    return (m_value >> (N * bits_per_byte) & 0xff);
  }
};
```

BytesOf - Implementation

```
namespace std {
  template <typename T>
  struct tuple_size<BytesOf<T>> :
    std::integral_constant<std::size_t, sizeof(T)> {};

  template <typename T, std::size_t N>
  struct tuple_element<N, BytesOf<T>> {
    using type = decltype(
      std::declval<BytesOf<T>&>().template get<N>()
    );
  };
}
```

Thank you!