Binding

In this lesson, we will list down the different initializers which support structured binding.

Initializers That Support Structured Bindings

Structured Binding is not only limited to tuples, we have three cases from which we can bind from:

1. If the initializer is an array:

```cpp
#include <iostream>

int main() {
  // works with arrays:
  double myArray[3] = { 1.0, 2.0, 3.0 };
  auto [a, b, c] = myArray;
  std::cout << a << " " << b << " " << c;
}
```

In this case, an array is copied into a temporary object, and `a`, `b` and `c` refers to copied elements from the array.

The number of identifiers must match the number of elements in the array.

2. If the initializer supports `std::tuple_size<>`, provides `get<N>()` and also exposes `std::tuple_element` functions:

```cpp
std::pair myPair(0, 1.0f);
auto [a, b] = myPair; // binds myPair.first/second
```

In the above snippet, we bind to `myPair`. But this also means that you can provide support for your classes, assuming you add `get<N>` interface implementation. See an example in the later section.

3. If the initialiser's type contains only non-static data members:

```cpp
struct Point  {
    double x;
    double y;
};

Point GetStartPoint() {
    return { 0.0, 0.0 };
}

const auto [x, y] = GetStartPoint();
```

`x` and `y` refer to `Point::x` and `Point::y` from the `Point` structure.

The class doesn't have to be `POD`, but the number of identifiers must equal to the number of non-static data members. The members must also be accessible from the given context.

> *Note:* In C++17, initially, you could use structured bindings to bind to class members as long as they were public. That could be a problem when you wanted to access such members in a context of friend functions, or even inside a struct implementation. This issue was recognised quickly as a defect, and it's now fixed in C++17. See P0969R0.

Expressive Code With Structured Bindings

If you have a map of elements, you might know that internally they are stored as pairs of `<const Key, ValueType>`.

Now, when you iterate through elements of that map:

```cpp
for (const auto& elem : myMap) { ... }
```

You need to write `elem.first` and `elem.second` to refer to the key and value. One of the **coolest use cases** of structured binding is that we can use it inside a range based for loop:

```cpp
std::map<KeyType, ValueType> myMap;
// C++14:
for (const auto& elem : myMap) {
    // elem.first - is velu key
    // elem.second - is the value
}
// C++17:
for (const auto& [key,val] : myMap) {
    // use key/value directly
}
```

In the above example, we bind to a pair of `[key, val]` so we can use those names in the loop.

Before C++17 you had to operate on an iterator from the map - which is a pair `<first, second>`. Using the real names `key/value` is more expressive than the pair.

The above technique can be used in:

```cpp
#include <map>
#include <iostream>
#include <string>

int main() {
    const std::map<std::string, int> mapCityPopulation {
        { "Beijing", 21'707'000 },
        { "Tokyo", 9'273'000 },
        { "London", 8'787'892 },
        { "New York", 8'622'698 },
        { "Rio de Janeiro", 6'520'000 }
    };

    for (auto&[city, population] : mapCityPopulation)
        std::cout << city << ": " << population << '\n';
}
```

▷   🖫   ↩   ⌞⌝

In the loop body, you can safely use `city` and `population` variables.

The next lesson will teach us how to implement structured binding for custom classes.