Constructors With Many Arguments

The basic principle is the same as it was with default constructors. The 'in_place' command will come in handy once again.

WE'LL COVER THE FOLLOWING

- std::make_optional()

Another use case is a situation where your type has more arguments in a constructor. By default `optional` can work with a single argument (r-value ref), and efficiently pass it to the wrapped type. But what if you'd like to initialize `Point(x, y)`?

You can always create a temporary copy and then pass it in the construction:

```
#include <iostream>
#include <optional>
using namespace std;

struct Point {
  Point(int a, int b) : x(a), y(b) { }
  int x;
  int y;
};

int main() {
  std::optional<Point> opt{Point{0, 0}};
  cout << opt->x << ", " << opt->y << endl;
}
```

or use `in_place` and the version of the constructor that handles variable argument list:

```
template< class... Args >
constexpr explicit optional( std::in_place_t, Args&&... args );
```

```
// or initializer_list:

template< class U, class... Args >

constexpr explicit optional( std::in_place_t,
                             std::initializer_list<U> ilist,
                             Args&&... args );
```

Your code can look like this:

```
std::optional<Point> opt{std::in_place_t, 0, 0};
```

The second option is quite verbose and omits to create temporary objects. Temporaries, especially for containers or larger objects, are not as efficient as constructing in place.

std::make_optional()

If you don't like `std::in_place` then you can look at the `make_optional` factory function.

The code:

```
auto opt = std::make_optional<UserName>();
auto opt = std::make_optional<Point>(0, 0);
```

Is as efficient as:

```
std::optional<UserName> opt{std::in_place};
std::optional<Point> opt{std::in_place_t, 0, 0};
```

`std::make_optional` implements in place construction equivalent to:

```
return std::optional<T>(std::in_place, std::forward<Args>(args)...);
```

And also thanks to mandatory copy elision from C++17 there is no temporary object involved.

Now that our optional type variable is created, how would we return it from a function? That's what the next lesson is all about.