Default Construction

Using std::optional with class constructors can be slightly inefficient sometimes. In this lesson, we'll learn a way to fix that.

WE'LL COVER THE FOLLOWING

Movable Types

If you have a class with a default constructor, like:

```
class UserName {
  public:
  UserName() : mName("Default")
  {

  }
  // ...
};
```

How would you create an `optional` that contains `UserName{}`?

You can write:

```
std::optional<UserName> u0; // empty optional
std::optional<UserName> u1{}; // also empty

// optional with default constructed object:
std::optional<UserName> u2{UserName()};
```

That works but it creates an additional temporary object. If we traced each different constructor and destructor call, we would get the following output:

```
UserName::UserName('Default')
UserName::UserName(move 'Default')  // move temp object
```

```
UserName::~UserName('')             // delete the temp object
UserName::~UserName('Default')
```

The code creates a temporary object and then moves it into the object stored in `optional`.

Here we can use a more efficient constructor - by leveraging `std::in_place_t`:

```
std::optional<UserName> opt{std::in_place};
```

With constructor and destructor traces you'd get the following output:

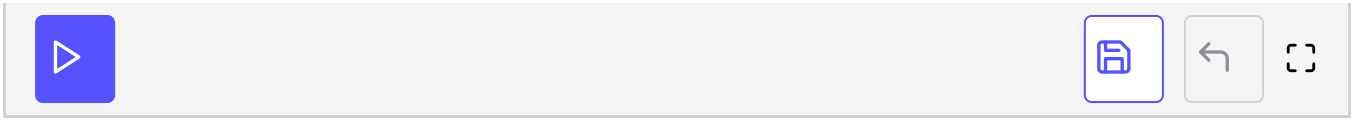
```
UserName::UserName('Default')
UserName::~UserName('Default')
```

The object stored in the optional is created in place, in the same way as you'd call `UserName{}`. No additional copy or move is needed.

See the example below. You'll also see the traces for constructors and destructor.

```
#include <optional>
#include <iostream>
#include <string>

void* operator new(std::size_t count) {
    std::cout << "allocating " << count << " bytes" << std::endl;
    return malloc(count);
}

void operator delete(void* ptr) noexcept {
    std::cout << "global op delete called\n";
    std::free(ptr);
}

class UserName {
public:
    explicit UserName() : mName("Default") {
        std::cout << "UserName::UserName('";
        std::cout << mName << "')\n";
    }

    explicit UserName(const std::string& str) : mName(str) {
        std::cout << "UserName::UserName('";
        std::cout << mName << "')\n";
    }
    ~UserName() {
        std::cout << "UserName::~UserName('";
        std::cout << mName << "')\n";
    }
    UserName(const UserName& u) : mName(u.mName) {
```

```
        std::cout << mName << "')\n";
    }
    UserName(UserName&& u) noexcept : mName(std::move(u.mName)) {
        std::cout << "UserName::UserName(move '";
        std::cout << mName << "')\n";
    }
    UserName& operator=(const UserName& u) { // copy assignment
        mName = u.mName;

        std::cout << "UserName::=(copy '";
        std::cout << mName << "')\n";

        return *this;
    }
    UserName& operator=(UserName&& u) noexcept { // move assignment
        mName = std::move(u.mName);

        std::cout << "UserName::=(move '";
        std::cout << mName << "')\n";

        return *this;
    }

private:
    std::string mName;
};

int main() {
    std::optional<UserName> opt(UserName{});
    //std::optional<UserName> opt{std::in_place};
}
```



Non Copyable/Movable Types

As you saw in the example from the previous section, if you use a temporary object to initialise the contained value inside `std::optional` then the compiler will have to use a move or a copy constructor.

But what if your type doesn't allow that? For example, `std::mutex` is not movable or copyable.

In that case, `std::in_place` is the only way to work with such types.

Apart from the default constructor, we also deal with constructors in which arguments are passed. How does that work with `std::optional`? Let's find out.