Operations on std::optional

Creating and accessing is just the tip of the iceberg. There are several operations that can be performed on std::optional.

WE'LL COVER THE FOLLOWING

Let's see what other operations are available for the type.

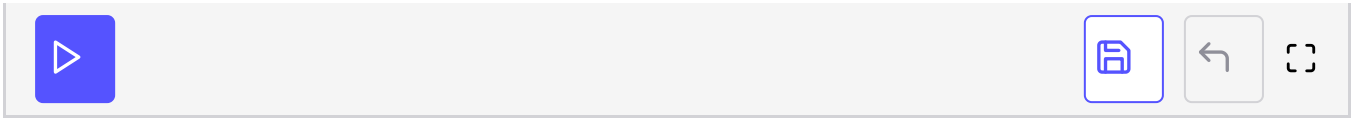
Changing the Value & Object Lifetime

If you have an existing optional object, then you can quickly change the contained value by using several operations like `emplace`, `reset`, `swap`, assign. If you assign (or reset) with a `nullopt` then if the optional contains a value, its destructor will be called.

Here's a quick summary:

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

class UserName {
public:
    explicit UserName(std::string str) : mName(std::move(str)) {
        std::cout << "UserName::UserName('" << mName << "')\n";
    }
    ~UserName() {
        std::cout << "UserName::~UserName('" << mName << "')\n";
    }
    UserName(const UserName& u) : mName(u.mName) {
        std::cout << "UserName::UserName(copy '" << mName << "')\n";
    }
    UserName(UserName&& u) : mName(std::move(u.mName)) {
        std::cout << "UserName::UserName(move '" << mName << "')\n";
    }
    UserName& operator=(const UserName& u) { // copy assignment
        mName = u.mName;
```

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

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

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

        return *this;
    }

private:
    std::string mName;
};

int main() {
    std::optional<UserName> oEmpty;

    // emplace:
    oEmpty.emplace("Steve");

    // calls ~Steve and creates new Mark:
    oEmpty.emplace("Mark");

    // reset so it's empty again
    oEmpty.reset(); // calls ~Mark
    // same as:
    //oEmpty = std::nullopt;

    // assign a new value:
    oEmpty.emplace("Fred");
    oEmpty = UserName("Joe");
}
```



Each time the object is changed, a destructor of the currently stored `UserName` is called.

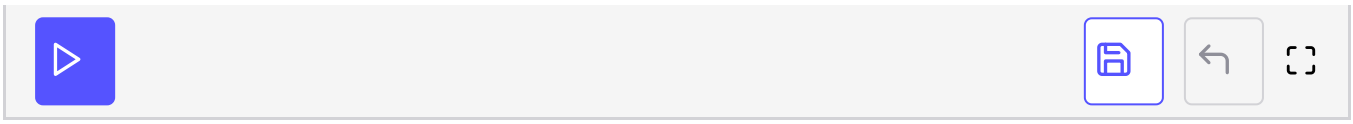
Comparisons

`std::optional` allows you to compare contained objects almost “naturally”, but with a few exceptions when the operands are `nullopt`.

See below:

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

int main() {
    std::optional<int> oEmpty;
    std::optional<int> oTwo(2);
```

```
    std::optional<int> oTen(10);

    std::cout << std::boolalpha;
    std::cout << (oTen > oTwo) << '\n';
    std::cout << (oTen < oTwo) << '\n';
    std::cout << (oEmpty < oTwo) << '\n';
    std::cout << (oEmpty == std::nullopt) << '\n';
    std::cout << (oTen == 10) << '\n';
}
```



When operands contain values (of the same type), then you'll see the expected results. But when one operand is `nullopt` then it's always "less" than any optional with some value.

To get a better idea, let's take a look at a few examples.