Accessing The Stored Value

This lesson goes into detail on the different ways to access optional type values.

Probably the most important operation for optional (apart from creation) is the way you can fetch the contained value. You have already seen it working with `operator*`.

However, here are several options:

- `operator*` and `operator->` - if there's no value the behaviour is undefined!
- `value()` - returns the value, or throws std::bad_optional_access
- `value_or(defaultVal)` - returns the value if available, or `defaultVal` otherwise

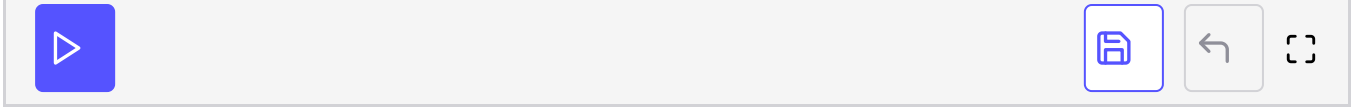
To check if the value is present you can use the `has_value()` method or just check if `(optional)` as optional is contextually convertible to bool.

Here's an example:

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

int main() {
  // by operator*
  std::optional<int> oint = 10; std::cout<< "oint " << *oint << '\n';
  // by value()
  std::optional<std::string> ostr("hello"); try
  {
    std::cout << "ostr " << ostr.value() << '\n';
  }
  catch (const std::bad_optional_access& e) {
    std::cout << e.what() << '\n';
  }
  // by value_or()
  std::optional<double> odouble; // empty
  std::cout<< "odouble " << odouble.value_or(10.0) << '\n';

}
```



And here's a handy pattern that checks if the value is present and then accesses it:

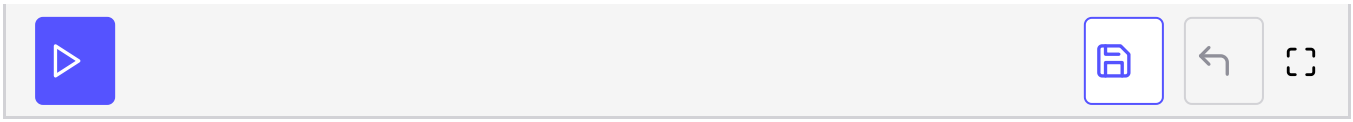
```
#include <iostream>
#include <optional>
using namespace std;

// compute string function:
std::optional<std::string> maybe_create_hello()
{
  return "Hello World";
}

int main() {

  // ...
  if (auto ostr = maybe_create_hello(); ostr)
    std::cout << "ostr " << *ostr << '\n';
  else
    std::cout << "ostr is null\n";

}
```



There is still a large variety of tools at our disposal when working with `std::optional`. We'll visit some of them in the rest of this section.