Be Careful With Braces When Returning

Let's observe some seemingly ordinary code and understand why it can potentially be harmful.

WE'LL COVER THE FOLLOWING

You might be surprised by the following code:

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

std::optional<std::string> CreateString()
{
  std::string str {"Hello Super Awesome Long String"};
  return {str}; // this one will cause a copy
  // return str; // this one moves
}

int main(){
  std::optional<std::string> ostr = CreateString();
  cout << *ostr << endl;
}
```

According to the Standard if you wrap a return value into braces `{}` then you prevent move operations from happening. The returned object will be copied only.

This is similar to the case with non-copyable types:

```
std::unique_ptr<int> foo() {
  std::unique_ptr<int> p;
  return {p}; // uses copy of unique_ptr and so it breaks...
  // return p; // this one moves, so it's fine with unique_ptr
}
```

Definition In The Standard

The Standard says class.copy.elision/3

> In the following copy-initialization contexts, a move operation might be used instead of a copy operation:
>
> - If the expression in a return statement (`[stmt.return]`) is a (possibly parenthesized) id-expression that names an object with automatic storage duration declared in the body or parameter-declaration-clause of the innermost enclosing function or lambda-expression
> - If the operand of a throw-expression is the name of a non-volatile automatic object (other than a function or catch- clause parameter) whose scope does not extend beyond the end of the innermost enclosing try-block (if there is one).

Try playing with the following example. The code shows a few examples with `std::unique_ptr`, `std::vector`, `std::string` and a custom type.

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

std::vector<int> CreateVec() {
    std::vector<int> v { 0, 1, 2, 3, 4 };
    std::cout << std::hex << v.data() << '\n';
    //return {v}; // this one will cause a copy
    return (v); // this one moves
    //return {std::move(v)}; // this one moves
    //return v; // this one moves as well
}

std::optional<std::vector<int>> CreateOptVec() {
    std::vector<int> v { 0, 1, 2, 3, 4 };
    std::cout << std::hex << v.data() << '\n';
    return {v}; // this one will cause a copy
    //return v; // this one moves
}

std::optional<std::string> CreateOptStr(){
    std::string s { "Hello Super Long String" }; // prevent SSO
    std::cout << std::hex << static_cast<void *>(s.data()) << '\n';
    //return {s}; // this one will cause a copy
    return s; // this one moves
}
```

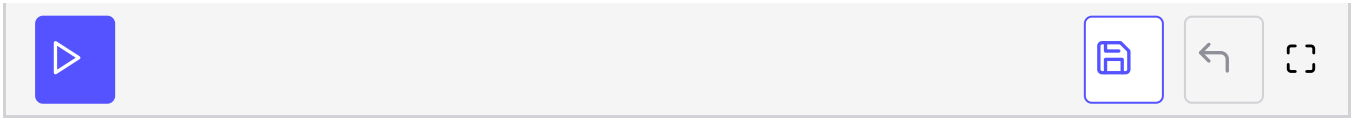
```
std::unique_ptr<int> CreatePtr() {
    std::unique_ptr<int> p;
    //return {p};  // uses copy of unique_ptr and so it breaks...
    return p; // this one moves, so it's fine with unique_ptr
}

int main() {
    std::cout << "CreateVec:\n";
    auto v1 = CreateVec();
    std::cout << std::hex << v1.data() << '\n';

    std::cout << "CreateOptVec:\n";
    auto v = CreateOptVec();
    std::cout << std::hex << v->data() << '\n';

    std::cout << "CreateOptStr:\n";
    auto s = CreateOptStr();
    std::cout << std::hex << static_cast<void *>(s->data()) << '\n';

    auto p = CreatePtr();
}
```



Creating and returning are out of the way. Now, we shall learn how to access the information inside out `optional variable`.