Non-movable/Non-copyable type

To understand further let's take a look at an example of non-movable/non-copyable type.

WE'LL COVER THE FOLLOWING

With C++17 we get clear rules on when elision has to happen, and thus constructors might be entirely omitted. In fact, instead of eliding the copies the compiler defers the “materialisation” of an object.

Why might this be useful?

- To allow returning objects that are not movable/copyable - because we could now skip copy/move constructors
- To improve code portability since every conformant compiler supports the same rule
- To support the *“return by value”* pattern rather than using output arguments
- To improve performance

Let’s look at an Example

Below you can see an example with a non-movable/non-copyable type, based on P0135R0:

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

struct NonMoveable
{
  NonMoveable(int x) : v(x) { }
  NonMoveable(const NonMoveable&) = delete;
  NonMoveable(NonMoveable&&) = delete;
  NonMoveable(NonMoveable&&) = delete;

  std::array<int, 1024> arr;
  int v;
};

NonMoveable make(int val)
{
  if (val > 0)
    return NonMoveable(val);
  return NonMoveable(-val);
}

int main()
{
  auto largeNonMoveableObj = make(90); // construct the object
  cout << "The v of largeNonMoveableObj is: " <<largeNonMoveableObj.v <<endl;
  return largeNonMoveableObj.v;
}
```

The above code wouldn't compile under C++14 as it lacks copy and move constructors. But with C++17 the constructors are not required - because the object `largeNonMovableObj` will be constructed in place.

Please notice that you can also use many `return` statements in one function and copy elision will still work.

> Moreover, it's important to remember, that in C++17 copy elision works only for unnamed temporary objects, and Named RVO is not mandatory.

To understand how mandatory copy elision/deferred temporary materialisation is defined in the C++ Standard, we must understand value categories which are covered in the next section.

In the next lesson, we'll take a look at the new updated value categories introduced in C++ 17. Read on to find out more.