Performance & Memory Consideration

Let's weigh the memory and performance constraints when working with std::optional.

WE'LL COVER THE FOLLOWING

std::optional Memory Footprint

When you use `std::optional`, you'll pay with an increased memory footprint.

The `optional` class wraps your type, prepares space for it and then adds one boolean parameter. This means it will extend the size of your type according to the alignment rules.

Conceptually your version of the standard library might implement `optional` as:

```
template <typename T> class optional
{
  bool _initialized;
  std::aligned_storage_t<sizeof(t), alignof(T)> _storage;

  public:
  // operations
};
```

Alignment rules for the optional are defined as follows in optional.optional:

> The contained value shall be allocated in a region of the optional storage suitably aligned for the type T.

For example, assuming `sizeof(double) = 8` and `sizeof(int) = 4`:

```
/*// sizeof(double) = 8
// sizeof(int) = 4
std::optional<double> od; // sizeof = 16 bytes

std::optional<int> oi; // sizeof = 8 bytes*/

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

int main(){
  cout << "Size of double = " << sizeof(double) << endl;
  cout << "Size of int = " << sizeof(int) << endl;

  std::optional<double> od;
  std::optional<int> oi;

  cout << "Size of optional<double> = " << sizeof(od) << endl;
  cout << "Size of optional<int> = " << sizeof(oi) << endl;
}
```

While `bool` type usually takes only one byte, the optional type needs to obey the alignment rules, and thus the whole wrapper is larger than just `sizeof(YourType) + 1 byte`.

Consider the two pieces of code below:

RangeOpt | Range

```
struct RangeOpt {
    std::optional<double> mMin;
    std::optional<double> mMax;
};

int main() {
    std::cout << sizeof(RangeOpt);
    return 0;
}
```

RangeOpt will take up more space than when you use your custom type in **Range**. In the first case, we're using 32 bytes! The second version is 24 bytes. This is because the second class can "pack" boolean variables at the front of the structure, while two optional objects in `Range` has to align to `double`.

`std::optional` was adapted directly from `boost::optional`. Head over to the

`std::optional` was adapted directly from `boost::optional`. Head over to the next lesson to find out more.