std::variant Creation

This lesson elaborates on creating and initializing std::variant

WE'LL COVER THE FOLLOWING

There are several ways you can create and initialize `std::variant`.

Creating/Initializing std::variant : An Example

```
#include <iostream>
#include <variant>
#include <string>
#include <vector>
using namespace std;

// monostate for default initialisation:
class NotSimple
{
public:
    NotSimple(int, float) { }
};

int main() {
// default initialisation: (the first type has to have a default ctor)
    std::variant<int, float> intFloat;
    std::cout << intFloat.index() << ", val: " << std::get<int>(intFloat) << '\n';

// monostate for default initialization:
// std::variant<NotSimple, int> cannotInit; // error
    std::variant<std::monostate, NotSimple, int> okInit;
    std::cout << okInit.index() << '\n';

// pass a value:
    std::variant<int, float, std::string> intFloatString { 10.5f };
    std::cout << intFloatString.index()<< ", value " << std::get<float>(intFloatString) << '\
// ambiguity
// double might convert to float or int, so the compiler cannot decide
//std::variant<int, float, std::string> intFloatString { 10.5 };
```

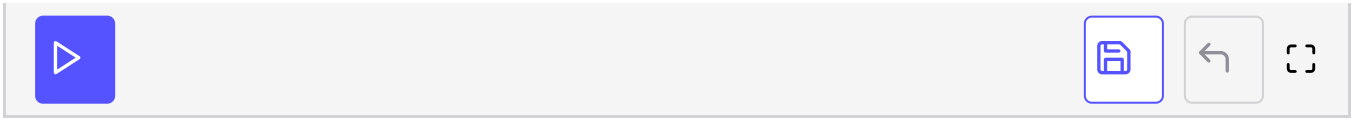
```
// ambiguity resolved by in_place
    variant<long, float, std::string> longFloatString {
        std::in_place_index<1>, 7.6 // double!

    };
    std::cout << longFloatString.index() << ", value "
              << std::get<float>(longFloatString) << '\n';

// in_place for complex types
    std::variant<std::vector<int>, std::string> vecStr {
        std::in_place_index<0>, { 0, 1, 2, 3 }
    };
    std::cout << vecStr.index() << ", vector size "

              << std::get<std::vector<int>>(vecStr).size() << '\n';

// copy-initialize from another variant:
    std::variant<int, float> intFloatSecond { intFloat };
    std::cout << intFloatSecond.index() << ", value "
              << std::get<int>(intFloatSecond) << '\n';
}
```



Elaboration of Example

- By default, a variant object is initialized with the first type
    - if that's not possible when the type doesn't have a default constructor, then you'll get a compiler error
    - you can use `std::monostate` to pass it as the first type in that case
    - `std::monostate` allows you to build a variant with "no-value" so it can behave similarly to `std::optional`
- You can initialize it with a value, and then the best matching type is used – if there's an ambiguity, then you can use a version `std::in_place_index` to explicitly mention what type should be used
- `std::in_place` also allows you to create more complex types and pass more parameters to the constructor

About std::monostate

In the example, you might notice a special type called `std::monostate`. This is just an empty type that can be used with variants to represent an empty state. The type might be handy when the first alternative doesn't have a default constructor. In that situation, you can place `std::monostate` as the first

alternative (or you can also shuffle the types, and find the type with a default constructor).

In the next lesson, we discuss more about some features of `std::variant`.