- Examples

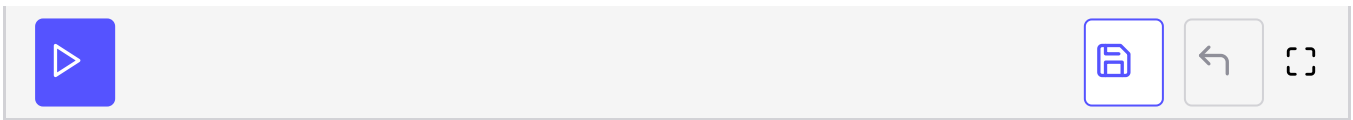
Let's have a look at a couple of examples of polymorphism.

WE'LL COVER THE FOLLOWING

Example 1: Dispatch with Dynamic Polymorphism

```
// dispatchDynamicPolymorphism.cpp

#include <chrono>
#include <iostream>

auto start = std::chrono::steady_clock::now();

void writeElapsedTime(){
    auto now = std::chrono::steady_clock::now();
    std::chrono::duration<double> diff = now - start;

    std::cerr << diff.count() << " sec. elapsed: ";
}

struct MessageSeverity{
        virtual void writeMessage() const {
                std::cerr << "unexpected" << std::endl;
        }
};

struct MessageInformation: MessageSeverity{
        void writeMessage() const override {
                std::cerr << "information" << std::endl;
        }
};

struct MessageWarning: MessageSeverity{
        void writeMessage() const override {
                std::cerr << "warning" << std::endl;
```

```
        }
};


struct MessageFatal: MessageSeverity{};

void writeMessageReference(const MessageSeverity& messServer){

        writeElapsedTime();
        messServer.writeMessage();

}

void writeMessagePointer(const MessageSeverity* messServer){

        writeElapsedTime();
        messServer->writeMessage();

}

int main(){

    std::cout << std::endl;

    MessageInformation messInfo;
    MessageWarning messWarn;
    MessageFatal messFatal;

    MessageSeverity& messRef1 = messInfo;
    MessageSeverity& messRef2 = messWarn;
    MessageSeverity& messRef3 = messFatal;

    writeMessageReference(messRef1);
    writeMessageReference(messRef2);
    writeMessageReference(messRef3);

    std::cerr << std::endl;

    MessageSeverity* messPoin1 = new MessageInformation;
    MessageSeverity* messPoin2 = new MessageWarning;
    MessageSeverity* messPoin3 = new MessageFatal;

    writeMessagePointer(messPoin1);
    writeMessagePointer(messPoin2);
    writeMessagePointer(messPoin3);

    std::cout << std::endl;

}
```



> Note: `std::cerr` of the class `std::ostream` represents the standard error stream. This is not a runtime error.

Explanation

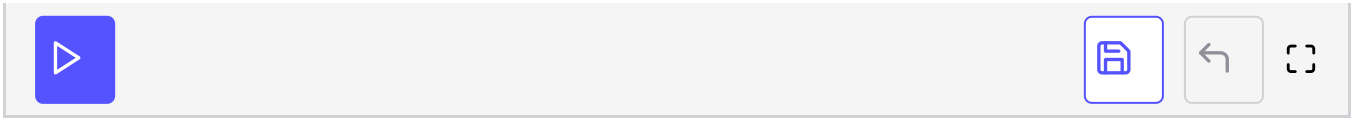
The structs in lines 15, 21, and 27 know, what they should display if used. The key idea is that the static type `MessageSeverity` differs from the dynamic type such as `MessageInformation` (line 61); therefore, the late binding will kick in and the `writeMessage` methods in lines 71, 72, and 73 are of the dynamic types. Dynamic polymorphism requires a kind of indirection. We can use references (57-59) or pointers (67-69).

From a performance perspective, we can do better and make the dispatch at compile time.

Example 2: Dispatch with Static Polymorphism

```
// DispatchStaticPolymorphism.cpp

#include <chrono>
#include <iostream>

auto start = std::chrono::steady_clock::now();

void writeElapsedTime(){
    auto now = std::chrono::steady_clock::now();
    std::chrono::duration<double> diff = now - start;

    std::cerr << diff.count() << " sec. elapsed: ";
}

template <typename ConcreteMessage>
struct MessageSeverity{
  void writeMessage(){
    static_cast<ConcreteMessage*>(this)->writeMessageImplementation();
  }
  void writeMessageImplementation() const {
    std::cerr << "unexpected" << std::endl;
  }
};

struct MessageInformation: MessageSeverity<MessageInformation>{
  void writeMessageImplementation() const {
    std::cerr << "information" << std::endl;
  }
};

struct MessageWarning:
MessageSeverity<MessageWarning>{
  void writeMessageImplementation() const {
    std::cerr << "warning" << std::endl;
  }
};

struct MessageFatal:
MessageSeverity<MessageFatal>{};
```

```
template <typename T>
void writeMessage(T& messServer){

        writeElapsedTime();
        messServer.writeMessage();

}

int main(){

    std::cout << std::endl;

    MessageInformation messInfo;
    writeMessage(messInfo);

    MessageWarning messWarn;
    writeMessage(messWarn);

        MessageFatal messFatal;
    writeMessage(messFatal);

    std::cout << std::endl;

}
```



> Note: `std::cerr` of the class `std::ostream` represents the standard error stream. This is not a runtime error.

Explanation

In this case, all concrete structs in lines 25, 31, and 38 are derived from the base class `MessageSeverity`. The method `writeMessage` serves as an interface that dispatches to the concrete implementations `writeMessageImplementation`. To make that happen, the object will be upcasted to the `ConcreteMessage`:

```
static_cast<ConcreteMessage*>(this)->writeMessageImplementation();
```

This is the dispatch at compile time; therefore, this technique is called static polymorphism.

To be honest, it took me a bit of time to get used to it but applying static polymorphism like that on line 42 is actually quite easy.

We'll solve an exercise in the next lesson.