- Solution

In this lesson, we'll look at different solution reivews for the last exercise.

WE'LL COVER THE FOLLOWING

Solution 1: Using the `if` Statement

```cpp
// dispatchIf.cpp

#include <chrono>
#include <iostream>

enum class MessageSeverity{
    information,
    warning,
    fatal,
};

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: ";
}

void writeInformation(){ std::cerr << "information" << std::endl; }
void writeWarning(){ std::cerr << "warning" << std::endl; }
void writeUnexpected(){ std::cerr << "unexpected" << std::endl; }

void writeMessage(MessageSeverity messServer){

    writeElapsedTime();
    if (MessageSeverity::information == messServer){
            writeInformation();

    }
    else if (MessageSeverity::warning == messServer){
            writeWarning();
    }
    else{
            writeUnexpected();
    }

}

int main(){

    std::cout << std::endl;

    writeMessage(MessageSeverity::information);
    writeMessage(MessageSeverity::warning);
    writeMessage(MessageSeverity::fatal);

    std::cout << std::endl;

}
```

Explanation

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

The function `writeMessage` in line 25 displays the elapsed time in seconds in line 27 since the start of the program and a log message. It uses an enumeration in line 6 for the message severity. We used the start time in line 12 and the current time in line 15 to calculate the elapsed time. As the name suggests, the `std::steady_clock` cannot be adjusted; therefore, it is the right choice for this measurement. The key part of the program is the part of the function `writeMessage` in line 25, in which we made the decision which message should be displayed. In this case, we used `if-else` statements.

Solution 2: Using the `switch` statement

```
// dispatchSwitch

#include <chrono>
#include <iostream>
```

```
#include <iostream>

enum class MessageSeverity{
    information,
    warning,
    fatal,
};

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: ";
}

void writeInformation(){ std::cerr << "information" << std::endl; }
void writeWarning(){ std::cerr << "warning" << std::endl; }
void writeUnexpected(){ std::cerr << "unexpected" << std::endl; }

void writeMessage(MessageSeverity messSever){

    writeElapsedTime();

    switch(messSever){
        case MessageSeverity::information:
            writeInformation();
            break;
        case MessageSeverity::warning:
            writeWarning();
            break;
        default:
            writeUnexpected();
            break;
  }

}

int main(){

    std::cout << std::endl;

    writeMessage(MessageSeverity::information);
    writeMessage(MessageSeverity::warning);
    writeMessage(MessageSeverity::fatal);

    std::cout << std::endl;

}
```

Explanation

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

The following program is quite similar to the previous one. Only the implementation of the function `writeMessage` changed. The function `writeMessage` in line 25 displays the elapsed time in seconds (line 27) since the start of the program and a log message. It uses an enumeration (line 6) for the message severity. We used the start time (line 12) and the current time (line 15) to calculate the elapsed time. As the name suggested, the `std::steady_clock` cannot be adjusted; therefore, it is the right choice for this measurement. The key part of the program is the part of the function `writeMessage` (line 25), in which we made the decision which message should be displayed. In this case, we used the `switch` statements.

To be honest, I had to look up the syntax for the `switch` statements to make it right.

Solution 3: Using a Dispatch Table

```
// dispatchHasttables

#include <chrono>
#include <functional>
#include <iostream>
#include <unordered_map>

enum class MessageSeverity{
  information,
  warning,
  fatal,
};

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: ";
}

void writeInformation(){ std::cerr << "information" << std::endl; }
void writeWarning(){ std::cerr << "warning" << std::endl; }
void writeUnexpected(){ std::cerr << "unexpected" << std::endl; }

std::unordered_map<MessageSeverity, std::function<void()>> mess2Func{
    {MessageSeverity::information, writeInformation},
    {MessageSeverity::warning, writeWarning},
    {MessageSeverity::fatal, writeUnexpected}
};
```

```
void writeMessage(MessageSeverity messServer){

        writeElapsedTime();


        mess2Func[messServer]();

}

int main(){

  std::cout << std::endl;

  writeMessage(MessageSeverity::information);
  writeMessage(MessageSeverity::warning);
  writeMessage(MessageSeverity::fatal);

  std::cout << std::endl;

}
```

Explanation

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

With the `if-else` or the `switch` statement, we used enumerator for dispatching to the right case. The key to our dispatch table behaves in a similar way.

Dynamic or static polymorphism is totally different. Instead of an enumerator or a key for dispatching to the right action, we used objects which decide autonomously at runtime (dynamic polymorphism) or compile-time (static polymorphism) what should be done.

Let’s move on to CRTP in the next lesson.