Bonusaufgaben zum C/C++-Praktikum

Objektorientierung in C++

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungsblatt 3

Hinweise zur Abgabe:
verwendet als Grundlage für die Bearbeitung die im Moodle bereitgestellte Vorlage. Beachtet auch die Hinweise auf dem ersten Bonuszettel.

Aufgabe 3.1: `[O]` Tic Tac Toe (18 Punkte)

Im folgenden werden wir eine Version von Tic-Tac-Toe schreiben, die man in der Kommandozeile spielen kann. Das Spiel soll so funktionieren, dass Du gegen den Computer antrittst. Hierbei setzt du immer Kreuze (X) und der Computer Kreise (O). Gleichzeitig soll aber auch ohne größere Änderung möglich sein, zwei Menschen oder zwei Computerspieler gegeneinander spielen zu lassen.

In `board.hpp` findest du drei `enum class`es. Eine `enum class` unterscheidet sich unter Anderem darin, dass immer der Name der `enum class` angegeben werden muss, wenn Elemente referenziert werden. (Beispielsweise `Color::CROSS` statt einfach nur `CROSS`).

- `Field` soll die einzelnen Felder des Spielfeldes darstellen.
- `GameStatus` dient später dazu, den Gewinner des Spieles festzulegen.
- `Color` wird dazu verwendet festzulegen, mit welcher Farbe ein `Player` spielt.

Unter den `enum class`es findest du außerdem Konvertierungsfunktionen, die du für die Bearbeitung der Aufgaben verwenden kannst.

Verantwortlich für den Spielablauf ist die Klasse `GameController`. Sie enthält eine Instanz der Klasse `Board` zum Speichern des aktuellen Spielstandes und besitzt eine Methode `void play(Player&, Player&)` die den Spielablauf realisiert.

Das 3×3-Spielfeld wird dementsprechend von der Klasse `Board` als geschachtelter `std::vector` namens `fields` verwaltet. Der äußere `std::vector` soll die Zeilen, der innere die Spalten adressieren. Durch `fields[0][1]` soll beispielsweise das mittlere Feld in der oberen Zeile erreicht werden.

`Board` enthält darüber hinaus außerdem noch die Methoden `begin()` und `end()` sowie Implementierungen von `operator[]`. Erstere sorgen dafür, dass mit einer `ranged-for`-Schleife über die Zeilen des `Board`s iteriert werden kann:

```
Board board;
for (std::vector<Field>& row : board) {
    ...
}
```

Die Implementierungen von `operator[]` erlauben Zugriff auf die Felder des `Board`s, wie du sie auch für den geschachtelten `std::vector` verwenden würdest:

```
std::cout << board[0][0] << std::endl;
```

Die Klasse `Player` ist eine abstrakte Basisklasse für die Implementierung von Spielern. Sie enthält als Member die Farbe (`CROSS` oder `CIRCLE`) mit der der Spieler Steine auf das Board setzen soll. Die abstrakte Methode `performNextMove(Board&)` implementiert das Spielverhalten, also das Setzen eines Steines wenn ein Spieler am Zug ist. Diese Methode wird in späteren Aufgaben von erbenden Klassen implementiert um verschiedene Computerspieler oder menschliche Spieler zu unterstützen.

In allen folgenden Teilaufgaben sollte darauf geachtet werden, die Sichtbarkeit für jeden Member und jede Methode immer so restriktiv wie möglich zu wählen. Passe die im Template vorgegebenen Sichtbarkeiten entsprechend an und füge ggf. weitere Labels hinzu.

3.1a) Die Klasse `Board` (4 Punkte)

Bitte bearbeite diese Teilaufgabe in den Dateien `board.hpp` und `board.cpp`.

Implementiere den Konstruktor der Klasse `Board`. Dieser sollte den Member `fields` so initialisieren, dass er ein 3×3 Feld darstellen kann, dass anfangs nur leere Felder (`Field::EMPTY`) enthält.

Implementiere außerdem die Methode `std::optional<GameStatus> whoWon() const`. Diese soll das Spielfeld prüfen und zurückgeben, ob jemand gewonnen hat: Wenn es einen Gewinner gibt, soll er im `optional`-Objekt enthalten sein. Ist das Spielfeld voll und hat keiner der Spieler gewonnen, so soll `TIE` zurückgegeben werden. Ist das Spiel noch im Gange, gebe ein leeres `optional`-Objekt zurück.

Hinweise

- Ein `std::optional<>`-Objekt enthält entweder eine Instanz des Template-Parameter-Typs oder kein Objekt. Ob eine `optional`-Instanz ein Objekt enthält, kann man Beispielsweise mit `bool has_value()` feststellen. Um auf das enthaltene Element zuzugreifen, kannst du entweder `operator*` oder `T& value()` nutzen.

3.1b) Ausgabe des Spielfeldes (2 Punkte)

Bitte bearbeite diese Teilaufgabe in den Dateien `board.hpp` und `board.cpp`.

Implementiere den Operator `std::ostream& operator<<(std::ostream&, const Board&)`, der das Spielfeld auf der Konsole ausgibt. Das Ergebnis dieser Methode kann beispielsweise so aussehen:

```
| X | O | O |

| X |   |   |

|   |   |   |
```

Das Board kann dann folgendermaßen auf der Konsole ausgegeben werden:

```
Board board;
std::cout << board << std::endl;
```

Hinweise

- `std::count` ist eine Instanz der Klasse `std::ostream`. Du kannst die `std::ostream` Instanz also genauso verwenden, wie du auch `std::cout` verwenden würdest.
- Der Rückgabewert von `operator<<` ist die gleiche Instanz von `std::ostream`, die die Funktion als Parameter bekommt. Das erlaubt wie im Beispiel oben das hintereinanderhängen von Aufrufen des Operators, also dass beispielsweise nach `board` noch `std::endl` ausgegeben wird.

3.1c) Der menschliche Spieler (2 Punkte)

Bitte bearbeite diese Teilaufgabe in den Dateien `human_player.hpp` und `human_player.cpp`.

Implementiere die Methode `void performNextMove(Board&)` der Klasse `HumanPlayer`, die Dich auf der Konsole fragt, an welcher Stelle du Dein Kreuz setzen möchtest. Bei ungültigen Eingaben soll ein Fehler auf der Konsole ausgegeben werden. Dann soll so lange erneut gefragt werden, bis die Eingabe gültig ist. Anschließend soll an der entsprechenden Stelle ein Kreuz gesetzt werden. Das kann beispielsweise so aussehen:

```
Wo wollen sie ihr Kreuz setzen? (Zählend von 0)
Eingabeformat: <Zeile> <Spalte>, zum Beispiel '2 0':
>> keine lust
Eingabe ist ungültig
Wo wollen sie ihr Kreuz setzen? (Zählend von 0)
Eingabeformat: <Zeile> <Spalte>, zum Beispiel '2 0':
>> 0 0
Alles klar.
```

3.1d) Zufälliges Spiel des Computers (2 Punkte)

Bearbeite diese Aufgabe in `random_player.hpp` und `random_player.cpp`.

Implementiere die Methode `performNextMove(Board&)` in `RandomPlayer`, die zufällig auf dem Spielfeld einen Kreis setzt. Überschreibe dabei keine Zeichen aus vorherigen Spielschritten.

3.1e) "Normales" Spiel des Computers (2 Punkte)

Bearbeite diese Aufgabe in `normal_player.hpp` und `normal_player.cpp`.

In den Dateien `perfect_player.hpp` und `perfect_player.cpp` ist die Klasse `PerfectPlayer` vorgegeben, welche immer die bestmögliche Spielentscheidung anhand des Minimax-Algorithmus trifft. Implementiere mithilfe der `performNextMove(Board&)`-Methode dieser Klasse, sowie deiner Implementation in `RandomPlayer` die `performNextMove(Board&)`-Methode in `NormalPlayer`. Dieses soll zufällig mit gleicher Wahrscheinlichkeit entweder die Methode `performNextMove(Board&)` von `PerfectPlayer` oder die von `RandomPlayer` aufrufen.

3.1f) Der Spielablauf (4 Punkte)

Implementiere die Funktion `void play(Player&, Player&)` in `tictactoe.cpp`, die folgenden Spielablauf realisiert: Beim Aufruf wird zufällig einer der beiden `Player` Anfangsspieler ausgewählt. Dann spielen beide Spieler abwechselnd, bis das Spiel vorbei ist. Anschließend wird Gewinner, falls existent, ausgegeben und gefragt ob erneut gespielt werden soll.

3.1g) tictactoe.cpp (2 Punkte)

Implementiere `main()` in `tictactoe.cpp`. Frage hierbei erst den Nutzer, gegen welchen der drei Computerspieler er spielen möchte, erstelle je eine Instanz des `HumanPlayer` und des Computerspielers, sowie eine Instanz von `GameController` und starte mit diesen anschließend die `play`-Methode.