Übung zur Vorlesung Programmieren 2
SoSe 2024

Blatt 5

Operatorüberladung als *friend*, Template-Funktionen und -Klassen (Generizität)

Erfordert die VL bis Kapitel	Operatorüberladung
Abgabe bis	04. Juni 2024, 16:00 Uhr

Richtlinien für Abgaben:

Lösungen, die diese Richtlinien nicht erfüllen, werden mit *0 Punkten* gewertet!
Abgegeben werden alle vorgegebenen, oder für die Aufgaben neu erstellten, Code/Text-Dateien.
Abgaben erfolgen über den Git-Server (↗) des Instituts.
Jeder in der Gruppe ist für die (pünktliche) Abgabe mitverantwortlich. Dafür gilt nicht die Commit-Zeit, sondern der Push-Zeitpunkt auf dem Server.

In Git ...
- wird für jedes Übungsblatt ein neuer Ordner angelegt: *Gruppe_XxYZ/blatt1*, */blatt2*, */blatt3*, ...
- dürfen keine temporären/automatisch generierten Dateien hinzugefügt werden (→ *.gitignore*).
- wird nur auf den Server gepusht, wenn der Code im aktuellen Commit lauffähig ist.
  Nicht lauffähige lokale Commits (für Zwischenstände) sind OK und sogar empfohlen. Diese sollten allerdings mit `"WIP:"` (work in progress) am Anfang der Commit-Nachricht gekennzeichnet werden.

Es wird kein vollständiger Code-Style vorgegeben, aber ...
- innerhalb jeder Abgabe muss die Code-Formatierung konsistent sein.
- es wird sinnvoll und konsistent eingerückt.
- Variablen haben aussagekräftige Namen.
- Compiler-Warnungen im *eigenen* Code, die in der Konsole ausgegeben werden, gelten als Fehler.

Allgemeine Infos:

Die Aufgaben und Erklärungen orientieren sich an (C++17) CMake-Projekten in der CLion IDE (↗), für Studenten kostenlos verfügbar für Windows, macOS und Linux.

Richtlinien & Infos werden bei Bedarf angepasst/erweitert. Sie gelten entsprechend je Übungsblatt.

Hinweise zu Blatt 5:

Zeit sparen: Da wir unsere beiden Spieler und ihre Schiffspositionen inzwischen auswendig kennen, und niemand mehr beim Gegner abguckt, gibt es ab diesem Blatt zwei Optionen in der Basis-*CMakeLists.txt*, um das Testen etwas effizienter zu machen:
`EXERCISE_SKIP_PLAYER_INIT` — überspringt die manuelle Eingabe der Spieler und ihrer Schiffe zum Start.
`EXERCISE_SKIP_WAITING` — entfernt die (Countdown-) Wartezeit zwischen den Spielzügen.

Tests: Für einige Aufgaben gibt es wieder Tests, sodass geprüft werden kann, ob die entsprechenden Lösungen grundlegend korrekt sind. (Die Tests kompilieren erst, wenn die Lösungen implementiert sind.)

Aufgabe 1

6 Punkte

> *friend*-Deklarationen
> Es kann hilfreich sein, wenn Klassen oder Funktionen auf private Member-Funktionen oder -Variablen anderer Klassen zugreifen können. Wie in den VL-Videos beschrieben, kann das mithilfe des Schlüsselworts `friend` erreicht werden. Dazu muss in der Klasse, auf die zugegriffen werden soll, der Zugriff für die zugreifende Klasse/Funktion erlaubt werden.
> Tipp: Der Scope-Resolution-Operator kann nicht nur Klassen/Funktionen in Namespaces identifizieren, sondern auch dabei helfen aus dem aktuellen Namespace heraus Klassen/Funktionen im "globalen" Namespace zu nutzen.

Ersetzt in der Klasse `PlayerSea` die Funktion `print()` durch eine Überladung von `operator<<`, damit Instanzen von `PlayerSea` mittels `cout << playerSea << endl;` ausgegeben werden können.

Deklariert den Operator als `friend` der Klasse `PlayerSea`, damit dieser auf die privaten Funktionen (`print___(..)`) der Klasse zugreifen kann.

Passt anschließend den restlichen Code entsprechend an, sodass der neue Operator genutzt wird.

Abgabe/Präsentation:

1. Der Operator is korrekt deklariert und definiert
2. Im Code des Operators kann auf die privaten Member-Funktionen von `PlayerSea` zugegriffen werden.
3. Zur Ausgabe während des Spiels wird der Operator und nicht mehr `print()` verwendet.

Aufgabe 2

6 Punkte

> *Template-Funktionen*
> Templates ermöglichen ähnlich dem Schlüsselwort `auto` Platzhalter (zB. für Basisdatentypen oder Klassen) im Code zu verwenden. So kann eine Template-Funktion die selben Operationen auf verschiedensten Datentypen ausführen, ohne dass diese Datentypen verwandt sind (Vererbung) oder dass je Datentyp eine Überladung der Funktion implementiert werden muss. Neben unnötig doppeltem Code, können solche Überladungen für verschiedene Dateitypen auch schnell zur Fehlerquelle werden, wenn zukünftige Änderungen nicht einheitlich auf alle Varianten angewendet werden.
> Weiterhin ist zu beachten, dass Template-Funktionen nicht in *.cpp*-Dateien definiert werden dürfen, um sie auch außerhalb dieser verwendet zu können.

Die drei Funktionen `sendRegularMissile(..)`, `sendTumblingMissile(..)` und `sendDoubleMissile(..)`, enthalten nahezu identische Anweisungen. Ersetzt sie durch eine einzige Template-Funktion `sendMissile(..)` in *game.inl* und passt den restlichen Code entsprechend an.

Abgabe/Präsentation:

Die Template-Funktion ...

1. ist korrekt deklariert, definiert und kann alle verfügbaren Raketentypen erzeugen.
2. wird in *game.cpp* korrekt benutzt.
3. ist nicht länger als eine einzelne `send___Missile(..)` Funktion (außer Scope-Resolution-Operatoren).

Aufgabe 3

28 Punkte

> *Template-Klassen*
> Mit Template-Klassen können diese Datentypen-Platzhalter für ganze Klassen definiert werden.
> Ein Beispiel, dass sowohl in der VL als auch in der Übung schon häufig eingesetzt wurde, ist die Klasse `std::vector<T>`, für dynamische Arrays.
> C++ bietet standardmäßig kein 2D-Array. Ein Grund dafür ist u.a., dass solche Datenstrukturen für verschiedene Anwendungen oft ganz verschieden aufgebaut sein müssten.

Implementiert in *Grid2D.h/.inl* die Klasse `Grid2D`, die mithilfe eines Vektors von Vektoren eine 2D-Array-Struktur abbildet. Nutzt einen Template-Parameter, sodass sie Daten beliebigen Typs enthalten kann.

Die Klasse soll möglichst intuitiv wie ein 2D-Array verwendet werden können. Überladet dazu für die Klasse `operator[]` und `operator()`. Sie sollen jeweils eine Referenz auf das angegebene Element liefern, sodass der Element-Zugriff mit `grid(coordX, coordY)` oder `grid[coords]` funktioniert, statt wie bisher mit `grid[coordY][coordX]`. Je Operator werden zwei Überladungen benötigt: eine "normale" Variante und eine als `const` (`const` oder nicht `const` muss für Rückgabewert und den Operator selbst übereinstimmen).

Konstruktor und Operatoren sollen Exceptions werfen, wenn die Parameter ungültige Werte enthalten.

Ersetzt in *Output.h* die "manuelle" `vector-vector`-Deklaration von `OutputGrid` durch ein `Grid2D` mit `OutputGridCell` als internem Datentyp. Findet alle weiteren Stellen im Code, die von dieser Typ-Änderung betroffen sind, und passt sie entsprechend an, sodass der neue Konstruktor und die neuen Operatoren verwendet werden.
Hinweis: Die weiteren Code-Stellen sind dieses Mal nicht mit *TODOs* gekennzeichnet. Nutzt also die entstehenden Compiler-Fehler, um sie zu finden.

Abgabe/Präsentation:

Die Template-Klasse ...

1. ist korrekt deklariert und definiert.
2. wird im restlichen Code korrekt genutzt.

Aufgabe 4 (Bonus)

(2 Punkte)

Warum werden bei Aufgabe 3 sowohl Operator-Varianten mit als auch ohne `const` benötigt?
Formuliert eine Antwort in dem vorgegebenen Kommentar-Abschnitt in *Grid2D.h*.

Gesamt: 40 Punkte