Praktikum 2: Klassen, `const` & Kopierkonstruktoren

Lernziele

- Arbeiten mit zweidimensionalen Arrays bzw. dynamisch erzeugtem Speicher
- Besonderheiten bezüglich Konstruktor/Destruktor/Zuweisung bei Verwendung von dynamischem Speicher
- Einhalten der „const correctness“
- (**Frühzeitiges** Stellen von Fragen **vor** dem Praktikumstermin!)

1 Aufgabenstellung

Im Praktikum PAD2 werden Sie in diesem Semester einen so genannten „Dungeon Crawler“ implementieren. Dabei geht es darum, eine Spielfigur (den Helden/die Heldin) rundenweise durch eine vorgefertigte Welt zu führen, wobei sich die Figur pro Runde um genau ein Feld bewegen kann.

Die Figur kann dabei auf freundlich oder feindliche gesinnte Nicht-Spieler Figuren treffen (Non-Player Characters oder NPCs), wobei es zu Interaktionen zwischen Spieler und NPC kommen kann.

In Praktikum 2 sollen Sie ein Grundgerüst implementieren, welches im Laufe des Semester noch erweitert wird. Achten Sie daher auf eine saubere Arbeitsweise, da der Quellcode Sie noch einige Wochen begleiten wird!

Hinweis: In den meisten Fällen werden Zeiger (und dynamisch erzeugte Objekte) verwendet. Stellen Sie sicher, dass Sie dies tatsächlich so implementieren!

Hinweis: Achten Sie ebenfalls darauf, nicht von den Klassen- und Methodennamen abzuweichen.

Hinweis: Stellen Sie für das Projekt bei den Compilereinstellungen den Standard `C++11` ein.

Hinweis: Sie können bei Bedarf die angegebenen Klassen beliebig um Attribute und Methoden erweitern.

1.1 Klassenübersicht

`Tile` stellt eine Kachel dar, also ein Feld der Spielwelt. Aktuell gibt es nur zwei verschiedene Typen von Kacheln: „Boden“ und „Wand“.

`DungeonMap` ist die Karte bzw. das Spielfeld des Dungeon Crawlers. Die Karte besteht im Wesentlichen aus einem dynamisch erzeugten, zweidimensionalem Array aus Kacheln (bzw. Zeiger auf Kacheln)

`Character` ist die Klasse für Spielfiguren. In diesem Praktikum werden Held/Heldin und NPCs noch nicht unterschieden. Auch verfügt eine Figur noch nicht über irgendwelche Attribute, abgesehen von einem `char`, der das zum Anzeigen verwendete Zeichen enthält.

`GameEngine` stellt die „Managerklasse“ für das gesamte Spiel dar. Das `GameEngine`-Objekt enthält einen `vector` aus Characterzeigern sowie ein `DungeonMap`-Objekt. Auch findet hier die Verarbeitung der Spielzüge statt. Prinzipiell werden hier die Mechaniken implementiert, allerdings werden viele Verarbeitungsschritte in den anderen Klassen implementiert. So bleibt das Spiel an sich flexibel und erweiterbar. Implementieren Sie also nicht alles in dieser Klasse!

1.2 Klasse `Tile`

1.2.1 `enum` zur Unterscheidung von Kacheltypen

Innerhalb der Klasse soll ein `public`-`enum` deklariert werden, welcher die aktuell verwendeten Kacheltypen `Floor` (Boden) und `Wall` (Wand) definiert.

1.2.2 Attribute

Die Klasse benötigt zwei Attribute: Ein Objekt vom Typ des eben deklarierten `enum`s um die Art des Objektes darzustellen und einen `Character*`. Dieser soll auf die Figur verweisen, welche auf dieser Kachel steht. Stellen Sie sicher, dass dieser den Wert `nullptr` hat, wenn **keine** Figur auf dieser Kachel steht.[1]

1.2.3 Konstruktoren

Implementieren Sie **genau einen** Konstruktor für die Klasse. Dieser soll das einen Parameter vom Typ aus 1.2.1 besitzen und das entsprechende Attribut dazu setzen. Denken Sie auch an die korrekte Initialisierung des `Character*`.

1.2.4 Methoden

- Implementieren Sie getter-Methoden für die beiden Attribute
- Schreiben Sie eine Methode `bool hasCharacter()`, welche `true` zurück liefert, wenn eine Figur auf dieser Kachel steht
- Schreiben Sie eine setter-Methode für den `Character*`

Die Bewegung einer Figur von einem Feld zum anderen wird durch die `Tile`-Klasse realisiert. Dazu implementieren Sie zwei Methoden (`onLeave` und `onEnter`), die aufgerufen werden um das Verlassen bzw. das Betreten einer Kachel darzustellen.

void onLeave(Tile* toTile) wird aufgerufen, um die auf dieser Kachel stehende Figur auf die Kachel `toTile` zu bewegen. Ist die Bewegung möglich, so muss der `Character*` dieses Objekte auf `nullptr` gesetzt werden, und die `onEnter`-Methode des Zielfeldes aufgerufen werden (Denken Sie daran, den `Character*` vor dem Setzen auf `nullptr` zu sichern, denn Sie brauchen diesen für den Aufruf von `onEnter`!).

void onEnter(Character* c, Tile* fromTile) wird in der Regel von der `onLeave`-Methode aufgerufen und simuliert das „Ankommen" einer Figur auf diesem Feld. Setzen Sie hier den `Character*` des Objektes entsprechend. Der Parameter `fromTile` wird, je nach Ihrer Implementierung, noch nicht zwangsläufig verwendet.

Die Idee hinter diesen beiden Methoden ist, das eine Kachel „selbst" entscheiden kann, ob sie „betretbar" ist oder nicht. Auch kann eine Kachel beim „Betreten werden" Aktionen ausführen. Dies wird im späteren Verlauf des Praktikums noch verwendet.

Aktuell sollen Sie in diesen beiden Methoden nur unterscheiden, ob das Zielfeld betretbar ist (=ob es sich dabei um den Typ `Floor` handelt) oder nicht und entsprechend der Informationen die Figur den Zug ausführen lässt oder nicht.

1.3 Klasse DungeonMap

Die Spielwelt besteht aktuell aus einem zweidimensionalen, dynamisch erzeugten Array aus `Tile`-Zeigern. Durch die Position im Array (Reihe/Spalte) ist die Position der Kachel in der Spielwelt definiert.

Hinweis: Verwenden Sie unbedingt Zeiger anstatt `Tile`-Objekte, da sonst spätere Praktika nicht funktionieren werden!

Hinweis: Der Datentyp für die Speicherung eines zweidimensionalen Arrays ist `Zieldatentyp**`. Ist der Zieldatentyp ein Zeiger, wie in diesem Fall, so ist der Datentyp `Tile***`. Für das Praktikum ist die Verwendung eines solchen Arrays **zwingend** vorgeschrieben!

1.3.1 `struct`-Typ zur Darstellung einer Position

Eine Position wird definiert durch die Kombination aus **Reihe** und **Spalte**. Definieren Sie ein `struct` für die Position.

1.3.2 Attribute

Speichern Sie die Höhe und Breite der Spielwelt in zwei **Konstanten** ab. Als einziges weiteres Attribut besitzt die Klasse den `Tile***` für die Speicherung der Spielwelt.

[1] Beachten Sie dies vor allem bei den Konstruktoren!

1.3.3 Konstruktoren & Destruktor

Da die Klasse mit dynamisch erzeugtem Speicher arbeitet, müssen Sie diesen im Destruktor der Klasse wieder frei geben. Ebenfalls müssen Sie den Kopierkonstruktor entweder **korrekt** implementieren, oder aber auf `private` setzen so dass dieser nicht aufgerufen werden kann. Objekte der Klasse werden damit „unkopierbar".

Implementieren Sie zwei Konstruktoren:

`DungeonMap(int height, int width)` legt ein zweidimensionales Array der geforderten Größe an und initialisiert alle `Tile*` darin mit `nullptr`.

`DungeonMap(int height, int width, const vector<string>& data)` legt ebenfalls ein Array der korrekten Größe an, initialisiert die `Tile*` allerdings entsprechend des Parameters `data` mit dynamisch erzeugten `Tile`-Objekten:
Hierzu wird eine zeichenweise Darstellung der Spielwelt übergeben, wobei das Zeichen `'.'` für ein `Tile`-Objekt vom Typ `Floor` steht und das Zeichen `'#'` für eine Wand.

1.3.4 Methoden

Implementieren Sie folgende Methoden:

`void place(Position pos, Character* c)` platziert eine Spielfigur `c` auf der Kachel an der Position `pos`.

`Position findTile(Tile* t)` ermittelt die Position der angegebenen Kachel `t` und gibt diese zurück.

`Tile* findTile(Position pos)` liefert einen Zeiger auf die Kachel an der angegebenen Position `pos`.

`Position findCharacter(Character* c)` ermittelt die Position der angegebenen Figur `c` und gibt diese zurück.

`void print()` gibt die Spielwelt auf der Konsole aus. Für Bodenkacheln soll das Zeichen `'.'`, für Wände das Zeichen `'#'` und für Spielfiguren das in den jeweiligen Objekten gespeicherte Zeichen verwendet werden.

1.4 Klasse `Character`

Diese Klasse ist (in der aktuellen Ausbaustufe) sehr einfach gehalten. Sie enthält ein Attribut vom Typ `char` welches das für die Figur verwendete Zeichen enthält. Das Attribut soll (ausschließlich) über einen Konstruktor gesetzt werden, ebenfalls soll ein getter dafür existieren.

Implementieren Sie eine Methode `int move()`, in der der Benutzer nach einer Bewegungsrichtung gefragt wird. Mögliche Richtungen sind die Zahlen 1 bis 9 wobei die Richtungen mit dem Ziffernblock auf der Tastatur übereinstimmen (8 bedeutet „nach oben", 1 bedeutet „links unten" usw.). Durch Auswahl der 5 kann eine Spielfigur in dieser Runde stehen bleiben.
Die gewählte Richtung soll zurück gegeben werden.

1.5 Klasse `GameEngine`

Die Klasse soll über zwei Attribute verfügen: Ein `DungeonMap`-Objekt und einen `vector<Character*>`, also die Spielwelt und die am Spiel beteiligten Spielfiguren.

Die Klasse soll einen Konstruktor besitzen, der die gleichen Parameter besitzt wie der zweite Konstruktor der Klasse `DungeonMap`. Das entsprechende Attribut soll mit genau diesen Parametern erzeugt werden (Initialisierungsliste!). Der Konstruktor soll ebenfalls eine Spielfigur dynamisch erzeugen und im `vector` abspeichern, und diese irgendwo auf der Spielwelt platzieren.

Darüber hinaus sollen Sie drei Methoden implementieren, die den Spielablauf bestimmen:

bool finished() liefert `true`, wenn das Spiel als beendet gelten soll. Dies kann beispielsweise passieren, wenn der Held eine bestimmte Position erreicht oder aber eine gewisse Anzahl an Spielrunden abgelaufen ist. Dies wird in nachfolgenden Praktika noch erweitert.

void turn() stellt einen Spielzug dar. Dazu wird der `vector` mit den Spielfiguren durchlaufen, von jedem Objekt die `move()`-Methode aufgerufen und der entsprechende Zug ausgeführt. Als Implementierungshilfe hier die durchzuführenden Schritte:

1. Ermitteln der Bewegungsrichtung durch Aufruf von `move()`
2. Zwischenspeichern der aktuellen Position der Figur mittels `findCharacter`
3. Zwischenspeichern des `Tile*` der Kachel, auf dem sich die Figur aktuell befindet (`findTile`)
4. Ändern der Spielerposition bezüglich der gewünschten Bewegungsrichtung
5. Zwischenspeichern des `Tile*` der Kachel, zu der sich der Spieler bewegen will (`findTile`)
6. Auf dem `Tile*` aus Schritt 3 die Methode `onLeave` aufrufen, als Parameter Ergebnis von Schritt 5 verwenden

void run() :

```
void GameEngine::run()
{
  while (!finished())
    turn();
}
```

2 `main`

Sie können folgenden Quellcode in der `main` verwenden:

```
vector<string> data{
 "##########",
 "####.....#",
 "###......#",
 "##.......#",
 "#........#",
 "#........#",
 "#........#",
 "#........#",
 "#........#",
 "##########",};

GameEngine ge(10,10,data);
ge.run();
```