Praktikum 1: Klassen mit dynamisch verwaltetem Speicher

Lernziele

- Wiederholung der Inhalte aus PAD1
- Dynamischer Speicher und Arrays
- Klassen und objektorientierte Programmierung
- Mehrdimensionale Arrays

1 Aufgabenstellung

Im ersten Praktikum entwickeln Sie eine Version des bekannten Spiels Tic-Tac-Toe.[1] Die Ausgabe des Spiels findet auf der Konsole statt. Um allerdings die Ausgabe flexibel zu gestalten, sollen Sie die eigentliche Programmlogik des Spiels von dem Programmbestandteil „Ausgabe“ trennen.

Neben der Klasse `TicTacToe`, welche die Programmlogik für das eigentliche Spiel enthält sollen Sie eine Klasse `Screen` implementieren, welche eine grafische Ausgabeeinheit simuliert. Die `Screen` Klasse stellt hierfür eine Art abstrakten Framebuffer[2] zur Verfügung. Die `TicTacToe`-Klasse „rendert“ dabei seine Ausgaben in diesen Buffer, während Ihre Implementierung der `Screen`-Klasse diesen Buffer wiederum auf der Konsole ausgibt.

Um das Gesamtsystem so einfach wie möglich zu halten, ist der Buffer nicht **pixelbasiert**, sondern **zeichenbasiert**, ein „Pixel“, also die kleinste adressierbare Einheit, stellt damit ein Zeichen dar.

Sie werden die `Screen`-Klasse für den Rest des Semesters `behalten` und für die weiteren Praktika verwenden. Daher sollten Sie darauf achten, hier so sauber und korrekt wie möglich zu arbeiten. Implementieren Sie alle geforderten Features und halten Sie sich an die Vereinbarung der Schnittstelle in der beigefügten Datei `Screen.h`.

Für das spätere Praktikumsprojekt wird die `Screen`-Klasse gegen eine von uns erstellte Klasse mit gleicher Funktionalität ausgetauscht, die allerdings auf dem GUI[3]-Framework **Qt**[4] basiert.

Im folgenden wird zunächst die `Screen`-Klasse erläutert, die Implementierung von TicTacToe erfolgt in Aufgabe 4 auf Seite 3.

2 Klasse `Screen`

Die Klasse Screen verwaltet den oben genannten Zeichen Buffer. Innerhalb dieses Buffers werden einzelne Zeichen durch x und y Koordinaten adressiert. Die Koordinate $0/0$ soll dabei in der linken oberen Ecke sein. Im Gegensatz zur „normalen“ Konsolenausgabe ist so eine direkte Manipulation einzelner Zeichen an einer Koordinate möglich. Dies ist für viele Anwendungsfälle wie beispielsweise Spiele sehr von Vorteil.

2.1 Strukturdefinitionen

In der Klasse `Screen` ist die Definition eines `structs` enthalten: Pos2d. Diese enthält einfach eine x und y Koordinate und dient als Datentyp für eine Position im Buffer bzw. auf einem Screen.

[1] Sollte es nicht bekannt sein: `https://de.wikipedia.org/wiki/Tic-Tac-Toe`
[2] Präziser: Einen Off-Screen Buffer
[3] „Graphical User Interface“: Grafische Oberfläche
[4] `https://en.wikipedia.org/wiki/Qt_(software)`

2.2 Attribute

Die Klasse besitzt die beiden const Attribute `m_width` und `m_height`, um die Breite und Höhe des Feldes zu speichern. `m_screen` stellt das dynamische char array da und enthält den eigentlichen Inhalt des Spielfeldes. Um in späteren Aufgaben Screens identifizieren zu können, benötigt die Klasse das Attribut `m_name`.

Hinweis: Es ist Ihnen freigestellt, alle **privaten** Attribute und Methoden anders zu benennen, Datentypen zu ändern bzw. beliebig neue **private** Attribute und Methoden hinzuzufügen. Die beigelegte Datei `Screen.h` zeigt in Hinsicht auf die privaten Member nur Vorschläge. Auch wie Sie die Verwaltung des Buffers intern gestalten, steht Ihnen völlig frei. **Einzige Einschränkung ist die Verwendung von Heap-Speicher!**

2.3 Konstruktoren

Schreiben Sie einen Konstruktor, welcher als Parameter die Breite und Höhe des Spielfelds übergeben bekommt. Erzeugen Sie weiterhin den für den Buffer benötigten Speicher in der entsprechenden Größe.

Hinweis: Da Sie dynamischen Speicher erzeugen, denken Sie daran einen Kopierkonstruktor und einen Destruktor zu implementieren.

2.4 Methoden

Implementieren Sie folgende Methoden, welche in der Datei `Screen.h` bereits deklariert sind:

- `getChar(Pos2d pos)` gibt das `char` an der übergebenen Stelle `pos` zurück und sollte daher `const` deklariert werden. Schreiben Sie eine zweiten getter, welcher nicht `const` ist und eine Referenz auf das `char` zurück gibt.
- `setChar(Pos2d pos, char c)` setzt das übergebene `char c` an der ebenfalls übergebenen Position `pos`.
- `setInt(Pos2d pos, const int &item)` speichert statt eines `chars` ein `int` an die übergebene Position.
- `setString(Pos2d pos, const string &item)` speichert analog zu den beiden vorherigen settern einen `string`.
- `fill(char c)` füllt das komplette Spielfeld mit dem übergebenen `char c` auf.
- `draw()` gibt das komplette Spielfeld auf der Konsole aus.

Hinweis: Die beiden Methoden `setInt` und `setString` füllen häufig mehr als nur ein Zeichen aus. Die übergebene Position `pos` ist in diesem Fall einfach die Startposition; Füllen Sie von dort aus nach rechts auf.

3 Sub-Screens

Die Idee eines Sub-Screens ist es, einen Screen in einen anderen unter zu bringen[5]. Ein Sub-Screen ist dabei ein eigentlich eigenständiger Screen komplett mit eigenem Buffer, der sich allerdings nicht selbst auf der Konsole ausgibt, sondern in einen anderen Screen[6]. Ein Sub-Screen hat damit auch ein eigenes Koordinatensystem mit der Position $0/0$ in der linken oberen Ecke des Sub-Screens.

Wird einem Screen ein Sub-Screen hinzugefügt, so wird ein **Ankerpunkt** mit übergeben. Der Ankerpunkt gibt an, an welcher Stelle im Screen sich die linke obere Ecke des Sub-Screens befindet. Das bedeutet eine Koordinate x_{sub}/y_{sub} in einem Sub-Screen hat die „globalen" Koordinaten

$$(x_{sub} + x_{ankerpunkt}/y_{sub} + y_{ankerpunkt})$$

[5] Evtl. hilft Ihnen auch das Konzept Picture-in-Picture weiter.
[6] Den übergeordneten, oder „Parent"-Screen

bezogen auf den Haupt-Screen.

Durch das arbeiten mit diesen „lokalen“ Koordinaten eines Sub-Screens bleibt Ihre Ausgabe flexibel, da Sie den gesamten Sub-Screen einfach durch Ändern des Ankerpunktes verschieben können, **ohne** an Ihrer eigentlichen Ausgabe neue Koordinaten errechnen zu müssen.

Für die Funktionalität der Sub-Screens besitzt die Screen Klasse die Attribute

- `Pos2d m_anchor`, der Ankerpunkt dieses Screens
- `string m_name`, um diesen Screen per Name suchbar zu machen
- `vector<Screen*> m_screens`, welcher alle Sub-Screens dieses Screens speichert

Sowie folgende drei Methoden:

- `void addSubScreen(Screen* subs, Pos2d anchor, const string& name)` zum Hinzufügen eines Sub-Screens. Die Methode setzt für den übergebenen Screen `anchor` sowie `name` und speichert den Zeiger in `m_screens`.
- `Screen* getSubScreen(const string& name)` sucht in `m_screens` einen Screen mit diesem Namen und gibt ihn zurück. Gibt es keinen solchen Screen, geben Sie `nullptr` zurück.
- `void draw(Screen& s)` ist **nicht** `public` und wird lediglich klassenintern verwendet. Mit dieser Methoden zeichnet sich ein Screen **nicht** direkt auf die Konsole, sondern in den übergebenen Screen `s`.

Die Methode `draw()` ohne Parameter ist `public` und zeichnet den Screen sowie alle Sub-Screens auf die Konsole. Hierzu sollten Sie zunächst alle Sub-Screens dieses Screens in diesen Screen zeichnen lassen[7] und anschließend den Buffer auf die Konsole zu schreiben.

4 Tic-Tac-Toe

Implementieren Sie das Spiel „Tic-Tac-Toe“ als Klasse und benutzen Sie zur Ausgabe auf der Konsole **ausschließlich** die Klasse `Screen`! Falls Sie in `TicTacToe` Debug-Ausgaben machen wollen, so verwenden Sie `cerr` anstatt `cout`.

Erklärung: Eine Unix-konforme Konsole bietet zwei Ausgabeströme: Die Standardausgabe (`cout`) und die Fehlerausgabe (`cerr`). So können Sie Fehlerausgaben und „normale“ Ausgaben voneinander trennen. Wenn Sie in NetBeans unter Project Properties » Run » Console Type den Wert Standard Output einstellen, wird die Ausgabe von `cerr` in roter Farbe dargestellt.

Das Spiel soll von zwei Spielern gespielt werden, die abwechseln Reihe und Spalte eingeben. Ein Spiel ist beendet, wenn entweder ein Spieler drei Felder in horizontaler, vertikaler oder diagonaler Ebene hat oder das Spielfeld voll ist. Jeder Spieler soll einen Namen besitzen, der zu Beginn des Spiels abgefragt wird.

Geben Sie die Namen und das Spielfeld jeweils in Sub-Screens aus.

[7] Mit der eben erwähnten Methoden `draw(Screen& s)`