Objektorientierte Systeme 1 - SWB2 & TIB2
Labor 2

Aufgabe 1: Ein einfaches Pac-Man-Spiel - Teil 3

In dieser Aufgabe programmieren wir unser Pac-Man-Spiel aus der Hausaufgabe fertig.

Schreiben Sie nun eine Klasse `Spieler`. Ein Spieler kann in dem Labyrinth bewegt werden. Wir werden die Klasse `Spieler` auch für die Geister nutzen. Die Klasse `Spieler` hat folgende Eigenschaften:

a) Die Instanzvariable `name` vom Typ `MyString` speichert den Namen des Spielers.
b) Die Instanzvariable `pos` vom Typ `Position` gibt die aktuelle Position und die Laufrichtung des Spielers im Labyrinth an.
c) Die Instanzvariable `muenzen` gibt an, wie viele Münzen der Spieler bereits in diesem Spiel bereits gesammelt hat.
d) Es gibt zwei Konvertierkonstruktoren. Der eine nimmt einen `MyString` der andere einen C-String als Parameter für den Namen des Spielers.
e) Die Instanzmethoden `getPos()` und `setPos(Position&)` lesen bzw. schreiben die aktuelle Position des Spielers.
f) Die Instanzmethode `setRichtung(Richtung)` setzt die Laufrichtung des Spielers in der Variablen `pos`.
g) Die Instanzmethode `getMuenzen()` liefert den Wert von `muenzen` zurück.
h) Die Instanzmethode `plusMuenze()` erhöht den Wert von `muenzen` um 1.
i) Die Instanzmethode `Spieler & schritt(Labyrinth&)` lässt den Spieler einen Schritt im übergebenen Labyrinth machen. Nutzen Sie dazu die Methode `schritt` der Klasse `Position`.
j) Die Instanzmethode `orientieren(Labyrinth &)` wählt für die Geister eine zufällige Laufrichtung und ist gegeben wie folgt:

```
// Schrittrichtung automatisch wählen lassen
void Spieler::orientieren(Labyrinth & lab) {
    Position postmp = pos;
    // Eine von 4 Richtungen auswählen
    int rint = rand() % 4;
    postmp.r = Richtung(rint);
    while ((lab.getZeichenAnPos(postmp)) == MAUER) {
        rint = rand() % 4;
        postmp.r = Richtung(rint);
    }
    pos.r = Richtung(rint);
}
```

Führen Sie nun die Klassen `Labyrinth` und `Position` aus der Hausaufgabe mit der am Ende der Aufgabe gegebenen Klasse `PacMan` zu einem Projekt zusammen und probieren Sie das Spiel aus. Sie können das folgende Hauptprogramm nutzen:

```
#include "PacMan.hpp"
```

```
const bool kErzeugen = true;

int main() {
    Labyrinth lab;
    if (kErzeugen) {
        lab.erzeugen();
        lab.exportDatei("lab.txt");
    }
    else {
        lab.importDatei("lab2.txt");
        lab.legeMuenzen();
        Spieler s("Demo");
        Spieler g[kAnzGeister] = { Spieler("Geist 1"),
                                   Spieler("Geist 2"),
                                   Spieler("Geist 3") };
        PacMan pm(lab, s, g, kAnzGeister);
        pm.spielen();
    }
}
```

Dateien der Klasse PacMan:

```
// Datei PacMan.hpp
#pragma once

#include "Labyrinth.hpp"
#include "Spieler.hpp"

class PacMan {
    Labyrinth * lab;           // Labyrinth für das Spiel
    Spieler * s;               // Spieler, der das Spiel spielt
    Spieler * g[kAnzGeister]; // Geister, die den Spieler jagen
    int schritte;              // Anzahl der Spielschritte
    int muenzen;               // Anzahl der Münzen im Labyrinth
public:
    // Konstruktor
    PacMan(Labyrinth & l, Spieler & sp, Spieler gArr[],
           int anzGeister);
    // Einen Schritt im Spiel ausführen
    void schritt();
    // Spiel spielen
    void spielen();
};
```

```
// Datei PacMan.cpp
#include <windows.h>
#include <conio.h>
// conio.h für _getch() und _kbhit()
```

```
// Achtung: Nicht Teil des Standards und daher abhängig vom Compiler
#include <iostream>
#include "PacMan.hpp"
using namespace std;

// Konstruktor
PacMan::PacMan(Labyrinth & l, Spieler & sp, Spieler gArr[],
               int anzGeister) {
    schritte = 0;          // Anzahl der Spielschritte
    muenzen = 0;           // Anzahl der Münzen im Labyrinth
    lab = &l;
    s = &sp;
    for (int i = 0; i < anzGeister; i++) {
        g[i] = &gArr[i];
    }
    // Spieler in die Mitte des Labyrinths setzen
    Position zentrum(lab->getSpalten() / 2, lab->getZeilen() / 2);
    s->setPos(zentrum);
    // Anzahl der Münzen vom Labyrinth übernehmen
    muenzen = lab->getMuenzen();
}

// Einen Schritt im Spiel ausführen
void PacMan::schritt() {
    // Schritte zählen
    schritte++;
    // Aktuelle Position des Spielers merken,
    // so dass sie mit WEG überschrieben werden kann.
    Position postmp = s->getPos();
    // Den Spieler einen Schritt machen lassen.
    s->schritt(*lab);
    // Wenn beim neuen Schritt eine Münze gefunden wurde ...
    if (lab->istMuenzeAnPos(s->getPos())) {
        // Anzahl der gesammelten Münzen des Spielers erhöhen
        s->plusMuenze();
        // Anzahl der vorhandenen Münzen reduzieren
        muenzen--;
    }
    // den getanen Schritt des Spielers einzeichnen
    lab->zeichneChar(WEG, postmp, s->getPos());
    lab->zeichneChar(ICH, s->getPos());
    // Dem Spieler 5 Schritte Vorsprung vor den Geistern lassen
    if (schritte == 5) {
        Position zentrum(lab->getSpalten() / 2, lab->getZeilen() / 2);
        for (int i = 0; i < kAnzGeister; i++) {
            g[i]->setPos(zentrum);
            lab->zeichneChar(GEIST, g[i]->getPos());
        }
```

```
    }
    // Jetzt laufen die Geister
    if (schritte > 5) {
        for (int i = 0; i < kAnzGeister; i++) {
            // Geist-Zeichen löschen
            lab->zeichneChar(WEG, g[i]->getPos());
            // Geist wählt zufällig seine Schrittrichtung
            g[i]->orientieren(*lab);
            // Geist macht seinen Schritt
            g[i]->schritt(*lab);
        }
        // Geister einzeichnen
        for (int i = 0; i < kAnzGeister; i++) {
            // Geister sammeln Münzen
            if (lab->istMuenzeAnPos(g[i]->getPos())) {
                // Anzahl der vorhandenen Münzen reduzieren
                muenzen--;
            }
            // Geist einzeichnen
            lab->zeichneChar(GEIST, g[i]->getPos());
        }
    }
}

// Spiel spielen
void PacMan::spielen() {
    // Das Spiel läuft solange noch Münzen im Labyrinth sind
    // und eine Geist nicht auf der Position des Spielers ist.
    // Temporäre Variable für die Keyboard-Eingabe
    char c = 'x';
    // Temporäre Variable, um die gewählte Richtung zu speichern
    Richtung r = s->getPos().r;
    // Position des Spielers einzeichnen
    lab->zeichneChar(ICH, s->getPos());
    // Temporäre Variable für die Abbruchbedingung der Spielschleife
    bool cond = (muenzen > 1);
    while (cond) {
        // Eine Weile warten (Windows)
        Sleep(700);
        // Eine Weile warten (C++11)
        // std::this_thread::sleep_for(std::chrono::milliseconds(700));
        if (_kbhit()) { // wenn Taste gedrückt wurde ...
            c = _getch();
            switch (int(c)) {
                // oben
            case 72: r = OBEN; break;
                // links
            case 75: r = LINKS; break;
```

```
                // rechts
            case 77: r = RECHTS; break;
                // unten
            case 80: r = UNTEN; break;
                // q = quit
            case 113: muenzen = 0; break;
            }
            s->setRichtung(r);
        }
        // Einen Spielschritt ausführen
        schritt();
        // Neue Spielsituation anzeigen
        lab->drucken();
        cout << "Gesammelte Muenzen: " << s->getMuenzen() << endl;
        cout << "Verbleibende Muenzen: " << muenzen << endl;
        cout << schritte << ". Schritt" << endl << endl;
        cout << "Zum Abbrechen q druecken" << endl;
        // Prüfen, ob noch Münzen da sind
        // und ob der Spieler noch nicht gefangen wurde
        cond = (muenzen > 1);
        for (int i = 0; i < kAnzGeister; i++) {
            cond = cond && !(s->getPos().istGleichZu(g[i]->getPos()));
        }
    }
}
```

Aufgabe 2: Ein FIFO-Speicher

Ein FIFO-Speicher (First-In First-Out Speicher) ist ein nützlicher aber einfacher Datenspeicher. Es können Datenelemente nacheinander in den Speicher hineingeschoben werden. In der Reihenfolge wie die Datenelemente gespeichert wurden, können sie auch wieder ausgelesen werden.

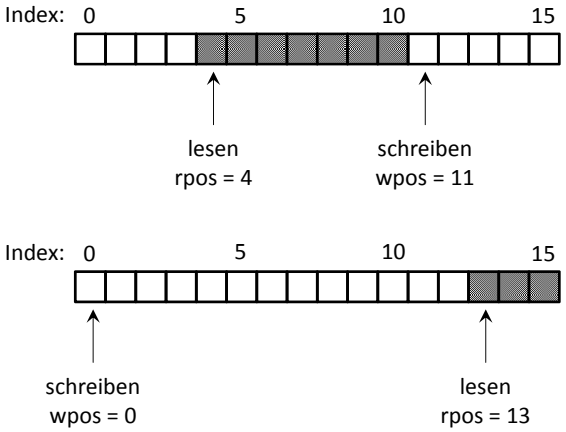
Programmieren Sie die Klasse `Fifo` mit den folgenden Eigenschaften. Die Klasse soll Datenelemente vom Typ **`char`** speichern können.

a) Die Instanzvariable `maxSize` gibt an, wie viele Datenelemente in dem Speicher maximal gespeichert werden können.
b) In der Instanzvariablen `number` wird abgelegt, wie viele Zeichen derzeit im Speicher gespeichert sind.
c) Die Instanzvariable `ptr` vom Typ Zeiger auf **`char`**. Der Zeiger wird auf das **`char`**-Array zeigen, das die Datenelemente aufnimmt.
d) Die Instanzvariablen `wPos` und `rPos`. Dies sind die Positionen des nächsten Schreibens (write) bzw. nächsten Lesens (read) im **`char`**-Array.
e) Der Konstruktor hat einen Defaultparameter mit Wert 20. Dieser Parameter gibt an, wie groß der Speicher ist. Der Speicher wird als dynamisches **`char`**-Array angelegt. Es werden die Instanzvariablen sinnvoll initialisiert.
f) Der Destruktor sorgt dafür, dass der Speicher des dynamischen **`char`**-Arrays wieder freigegeben wird.
g) Die Instanzmethoden `getWPos` und `getRPos` liefern die aktuellen Positionen für Schreiben

bzw. Lesen zurück.

h) Die Instanzmethode `isEmpty` liefert **`true`** zurück, wenn keine Daten gespeichert sind, d.h. der FIFO-Speicher leer ist. Andernfalls wird **`false`** zurückgegeben. Die Instanzmethode `isFull` ist analog für den Fall, dass der Speicher voll ist.

i) Die Instanzmethode `push(`**`char`**`)` schreibt ein Zeichen in den Speicher, wenn dieser noch Platz hat. Da immer wieder geschrieben und gelesen werden kann, soll der Speicher als Ringpuffer realisiert werden, d.h. wenn bereits am höchsten Index des **`char`**-Arrays geschrieben wurde und am kleinsten Index noch Platz ist, so wird wieder beim kleinsten Index geschrieben. Die Methode liefert den Index zurück, an den erfolgreich geschrieben wurde. Wenn das **`char`**-Array voll ist, so soll nicht geschrieben und -1 zurückgegeben werden.



j) Die Instanzmethode `pop` liefert das nächste zu lesende Zeichen vom **`char`**-Array zurück und macht dessen Platz im Array wieder frei. Wenn der Speicher leer ist, wird das NULL-Zeichen zurückgegeben.

Schreiben Sie ein Hauptprogramm, um Ihre Klasse ausgiebig zu testen.

Aufgabe 3: Grafische Objekte - Teil 1

Das Programm, das sie in diesem und den nächsten Laboren entwickeln werden, kann grafische Objekte (Punkte, Kreise, ...) als Objekte von Klassen darstellen. Sie können sich die Objekte als Datenrepräsentationen von grafischen Objekten in einem Zeichenprogramm vorstellen. Die grafischen Objekte sind definiert über Koordinaten und könnten (später) Eigenschaften haben wie z.B. eine Strichstärke und -art.

Um bei dem Programm den Überblick zu behalten, müssen Sie für jede Klasse eine eigene Headerdatei (*.hpp), welche die Klassendeklaration enthält, und eine eigene Programmdatei (*.cpp) schreiben. Weitere Hilfsfunktionen können Sie auch in eigene Dateien auslagern.

Arbeiten Sie bei diesen Klassen sorgfältig und achten Sie auf vollständige und korrekte Programmierung, damit Sie eine gute Basis für Erweiterungen haben. Denn Sie müssen in den folgenden Laborübungen die Klassen von diesem Labor weiterhin benutzen und ggf. erweitern.

3.1 Sie sollen zwei Klassen definieren: `Point` und `Circle`.

(a) Die Klasse `Point` repräsentiert das geometrische Objekt Punkt.

(b) Die Klasse hat die Instanzvariablen `x` und `y` vom Typ **`double`**. Diese beiden Instanzvariablen repräsentieren die x- und y-Koordinaten eines Punktes.

(c) Die Klasse `Circle` repräsentiert einen Kreis.

(d) Sie hat die Instanzvariablen `centre` vom Typ `Point` und `radius` vom Typ **`double`**. Die Variable `centre` steht also für den Mittelpunkt des Kreises.

(e) In beiden Klassen sollen die Instanzvariablen vor Zugriff von außerhalb der Klasse geschützt sein.

(f) Schreiben Sie für beide Klassen Konstruktoren mit geeigneten Defaultparametern.

(g) Schreiben sie get- und set-Methoden für alle Instanzvariablen.

(h) Die Instanzfunktion `move(double dx, double dy)` verschiebt einen Punkt bzw. einen Kreis um die Werte `dx` und `dy` in x- bzw. y-Richtung.

(i) Beide Klassen sollen eine Funktion `print` besitzen, welche die Koordinaten eines Punktes bzw. die Koordinaten des Mittelpunktes und den Radius eines Kreises auf dem Bildschirm ausgibt. Die Funktion `print` besitzt einen Parameter vom Typ **`bool`**, der steuert, ob nach der Ausgabe ein Zeilenvorschub (`endl`) durchgeführt wird oder nicht. Der Defaultwert für diesen Parameter soll **`true`** sein. Das Format für die gewünschte Ausgabe können Sie der folgenden Ausgabe des Hauptprogramms entnehmen.

Die Ausgabe des unten gegebenen Hauptprogrammes soll wie folgt aussehen:

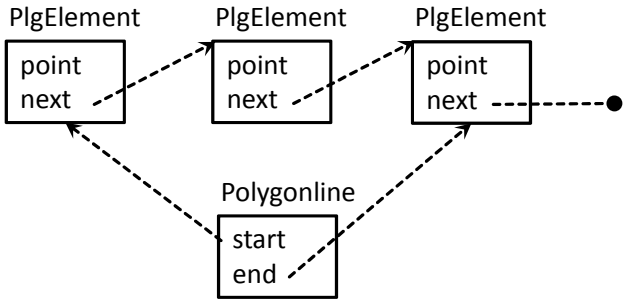
```
Ausgabe 1:
(0, 0)
<(0, 0), 0>
Ausgabe 2:
(1.1, 2.2) == (1.1, 2.2)
<(1.1, 2.2), 3.3>
Ausgabe 3:
(2.1, 3.2)
<(3.1, 4.2), 3.3>
```

```
#include <iostream>
#include "Circle.hpp"
using namespace std;

// Hauptprogramm
int main(void)
{
    Point p;
    Circle c(p);
    cout << "Ausgabe 1:" << endl;
    p.print();
    c.print();
    p.setX(1.1);
    p.setY(2.2);
    c.setCentre(p);
    c.setRadius(3.3);
    cout << "Ausgabe 2:" << endl;
    p.print(false);
    cout << " == (" << p.getX() << ", " << p.getY() << ")"
         << endl;
```

```
    c.print();
    p.move(1.0, 1.0);
    c.move(2.0, 2.0);
    cout << "Ausgabe 3:" << endl;
    p.print();
    c.print();
    return 0;
}
```

3.2 Nun sollen Sie die Klasse `Polygonline` (Polygonlinie, Linienzug) definieren. Die Objekte der Klasse `Polygonline` repräsentiert einen Linienzug. Ein Linienzug ist durch eine Folge von Punkten repräsentiert. Die Folge von Punkten wird in `Polygonline` als eine mit Pointern verkettete Liste dargestellt, da wir nicht wissen, wieviele Punkte die Linie haben wird.
Deklarieren Sie eine Klasse `PlgElement` (Polygonline Element), die die Elemente der verketteten Liste darstellt (siehe Abb.).



Die Klasse `PlgElement` hat zwei Instanzvariablen: `point` vom Typ `Point` und `next` vom Typ Pointer auf `PlgElement`. Die Variable `point` beinhaltet einen Punkt des Linienzugs und `next` verweist auf das nächste `PlgElement` des Linienzugs.
Schreiben Sie einen geeigneten Destruktor diese Klassen, wenn ihr Defaultdestruktor nicht ausreicht.
Die Ausgabe des unten gegebenen Hauptprogrammes soll wie folgt aussehen:

```
Ausgabe 1:
||
|(3, 3)|
Ausgabe 2:
|(1, 1) - (2, 2)|
|(3, 3) - (4, 4) - (5, 5)|
Ausgabe 3:
(2.5, 2.5)
|(1, 1) - (2, 2)|
|(3, 3) - (4, 4) - (5, 5)|
Ausgabe 4:
|(1, 1) - (2, 2) - (3, 3) - (4, 4) - (5, 5)|
|(3, 3) - (4, 4) - (5, 5)|
Ausgabe 5:
|(1, 1.5) - (2, 2.5) - (3, 3.5) - (4, 4.5) - (5, 5.5)|
|(3, 3) - (4, 4) - (5, 5)|
```

```
#include <iostream>
#include "Polygonline.hpp"
using namespace std;

// Hauptprogramm
int main(void)
{
    Point p1(1, 1), p2(2, 2), p3(3, 3), p4(4, 4), p5(5, 5);
    Polygonline l1;
    Polygonline l2(p3);
    cout << "Ausgabe 1:" << endl;
    l1.print();
    l2.print();
    l1.addPoint(p1).addPoint(p2);
    l2.addPoint(p4).addPoint(p5);
    cout << "Ausgabe 2:" << endl;
    l1.print();
    l2.print();
    p2.move(0.5,0.5);
    cout << "Ausgabe 3:" << endl;
    p2.print();
    l1.print();
    l2.print();
    l1.appendPolygonline(l2);
    cout << "Ausgabe 4:" << endl;
    l1.print();
    l2.print();
    l1.move(0, 0.5);
    cout << "Ausgabe 5:" << endl;
    l1.print();
    l2.print();
    return 0;
}
```