IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Aufgabe 04

Hinweise
Abgabe: Stand des Git-Repositories am 17.5.2022 um 9 Uhr.
Die Dateien zur Bearbeitung dieser Aufgabe erhalten Sie, indem Sie die neue Aufgabe aus dem Aufgabenrepository in Ihr lokales mergen. Dies geschieht mit `git pull common main` innerhalb Ihres Repositories. Die Lösungen committen Sie bitte in Ihr lokales Repository und pushen sie in Ihr Repository auf dem Gitlab-Server.

Achtung: In den Vorlesungen 3 und 4 wurden Regeln zur C++-Programmierung vorgestellt. Außerdem sollen beim Kompilieren keine Warnings mehr erscheinen. Eine Nichtbeachtung dieser Regeln führt zu Punktabzug, sofern keine überzeugende Begründung für die Missachtung geliefert wird.

Teilaufgabe 1 (2 Punkte)

In dieser Aufgabe soll eine Graphen-Datenstruktur eines gerichteten Graphen erstellt werden. In `graph.h` sind die entsprechenden Klassen `Vertex, Edge, Graph` bereits definiert. Bei Aufruf der kompilierten Datei wird ein Testgraph aufgebaut und ausgegeben. Die Ausgabe soll nach Abschluss der Aufgabe wie folgt aussehen:

```
$ ./graph
-----------------------------------------
Vertex Name: Vertex1
Input Edges:
Edge ID: 2
Output Edges:
Edge ID: 0
Edge ID: 1

Vertex Name: Vertex2
Input Edges:
Edge ID: 0
Output Edges:
```

Appelstraße 4 | 30167 Hannover
fon 0511 762-19691 | fax 0511 762-19694
www.ims.uni-hannover.de | info@ims.uni-hannover.de

```
Vertex Name: Vertex3
Input Edges:
Edge ID: 1
Output Edges:
Edge ID: 2

-----------------------------------------
```

Programmieren und vervollständigen sie in `graph.cpp` die Methoden und Konstruktoren von Vertex, Edge und Graph, die nicht bereits in `graph.h` inline definiert wurden!

Hinweise:

- Weitere Erläuterungen finden Sie in den Kommentaren in `graph.h`.
- Die ID eines Vertex ist sein Index in `Graph::vertices`. Die ID einer Edge ist ihr Index in `Graph::edges`.
- Die Member `inEdgeIds` und `outEdgeIds` der Klasse `Vertex` enthalten jeweils die IDs der ankommenden bzw. abgehenden Kanten. In `outEdgeIds` und `inEdgeIds` können die IDs unsortiert abgelegt werden
- Bei den Konstruktoren ist darauf zu achten, dass alle Datenmember initialisiert werden. Verwenden Sie dafür Initialisierungslisten! Dies bedeutet nicht, dass alle Member zwingend im Konstruktor bereits ihren finalen Wert annehmen müssen.

Teilaufgabe 2 (1,5 Punkte)

Beantworten Sie folgende Fragen in einer von Ihnen erzeugten Datei im Repo!
Hinweis: `int *p` ist identisch mit `int* p`.

a) Was gibt der folgende Code aus? Begründen Sie ihre Antwort!

```
std::vector<int> v = {10, 9};
int* p1 = &v[0];
*p1     = --*p1 * *(p1 + 1);
std::cout << v[0]<<", "<<v[1] << std::endl;
```

b) Erklären Sie jede der folgenden Definitionen! Ist eine davon illegal? Wenn ja, warum? `int i = 0;`

a) `short* p1 = &i;`

b) `int*  p2 = 0;`

c) `int*  p3 = i;`

d) `int*  p4 = &i;`

c) Sei `p` ein Pointer auf `int`. Unter welcher Bedingung wird *Code1* und unter welcher *Code2* ausgeführt? Welche Probleme können dabei auftreten?

```
if (p) {Code1}
if (*p) {Code2}
```

d) Es sei ein Pointer `p` gegeben. Kann man herausfinden, ob `p` auf ein gültiges Objekt zeigt? Wenn ja, wie? Wenn nein, warum nicht?

Teilaufgabe 3 (1,5 Punkte)

Welche der folgenden `unique_ptr`-Deklarationen sind illegal oder führen möglicherweise im Folgenden zu Programmfehlern? Erklären Sie, worin die Probleme jeweils bestehen!

```
double e   = 2.7182;
double* dp = &e;
double* dp2 = new double(3.1415);
double* dp3 = new double(1.618);
double& dr1 = *dp3;
std::vector<double> v = {1.5, 2.5};
using DoubleP = std::unique_ptr<double>;
```

a) `DoubleP pd0(std::make_unique<double>(3.1415));`

b) `DoubleP pd1(dp2);`

c) `DoubleP pd2(dp);`

d) `DoubleP pd3(pd1.get());`

e) `DoubleP pd4(&e);`

f) `DoubleP pd5(e);`

g) `DoubleP pd6(pd0);`

h) `DoubleP pd7(&v[0]);`

i) `DoubleP pd8(&dr1);`

Teilaufgabe 2

a)
```
std::vector<int> v={10, 9};
int* p1 = &v[0];
*p1 = --*p1 ** (p1+1);
std::cout<< v[0] << ", " << v[1]<<std::endl;
```

da $p_1 = \&v[0]$, $*(p_1+1) = *(\&v[1]) = 9$ $\quad --*p_1 = 9$

So: $*p_1 = v[0] = 81 \quad v[1] = 9$

b) `int i=0;`

a) `short *p1 = &i` illegal, da Type von i int ist, p1 darf nicht auf short variable zeigen

b) `int* p2 = 0;` Legal, initialliert p als Nullpointer

c) `int* p3 = i;` illegal, p3 ist eine Adresse, sollint*p=&i;

d) `int* p4 = &i` Legal, P zeigt auf int i

c) `if(p) {code 1};`

wenn p != nullptr ist, wird code 1 ausgeführt, kein Problem dabei auftreten.

`if (*p) {code 2};`

wenn *p != 0 ist, wird code 2 ausgeführt.

Problem auftreten: wenn p ein Nullpointer ist, p zeigt nicht auf ein Objekt. *p weiß man nicht.

d) Ja, mit if(p) weißt man ob ein Objekt gültig ist. wenn p ein Nullpointer ist, das Objekt ist ungültig.

Sonst zeigt p auf ein gültiges Objekt.

Teilaufgabe 3

a) Legal, initialisierst pd0 mit neualloziert Speicher

b) Legal, initialisierst pd1 mit einem auf dynamischen Speicher zeigenden Adresse

c) Legal, initialiert pd2 mit einen Rawpointer

d) Legal, initialiert pd3 mit dem selben Pointer pd1

e) Legal, initialiert pd4 mit dem nicht dynamischen azolliert Speicher

f) Illegal, pd5 ist ein unique_ptr mit Type double*, e ist ein double Wert 2.7182, Typ passt nicht

g) Illegal, unique_ptr darf nicht kopiert werden

h) Legal, initialisiert pd7 mit Adresse des Vector

i) Illegal, double p pd8(&dr1) heißt, initialisiert pd8 mit pd3, aber unique_ptr darf nicht kopiert werden.