Vererbung

I. Einleitung

Dieses und die folgenden beiden Praktika drehen sich um Netzwerke. Dieses Praktikum bereitet die grafische Darstellung vor, im kommenden Praktikum geht es um die Implementierung als Graph, im übernächsten um Algorithmen auf Graphen. Über diese drei Praktika werden Sie Ihren Code kontinuierlich weiterentwickeln.
Sie werden dabei mit kleinen Netzwerken wie dem in Abbildung 1 arbeiten: Schnelle Netzwerkknoten (Rechtecke) sind untereinander mit schnellen Verbindungen (dicke Linien) verbunden und bilden einen Backbone. An die schnellen Knoten sind langsame Knoten (Kreise) mit langsamen Verbindungen angeschlossen (dünne Linien). Einige langsame Knoten sind auch untereinander verbunden. Die Verbindungen sind mit den jeweiligen Kosten der Datenübertragung beschriftet – also gewissermaßen mit der Übertragungsdauer pro Dateneinheit.

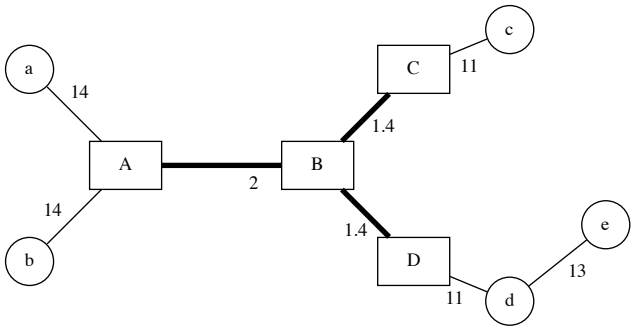


Abbildung 1: Beispielnetzwerk

Rahmen

Wie in den vorherigen Praktika verarbeiten Sie gegebene Eingabedaten mit Ihrem Programm. Die Eingabedaten geben Sie per Standardeingabe ein. Die Ausgabe Ihres Programms läuft über die Standardausgabe.
Am einfachsten bewerkstelligen Sie dies über die Konsole, indem Sie *Pipes* verwenden und die Standardausgabe in eine Datei umleiten. Im folgenden Beispiel wird der Inhalt der Datei `input.txt` an die Standardeingabe Ihres Programms `main` übergeben. Die Ausgabe wird anschließend in die Datei `output.txt` umgeleitet. Evtl. vorhandener Inhalt in dieser Datei wird überschreieben.

```
cat input.txt | ./main > output.txt
```

Unter Windows verwenden Sie `type` statt `cat`.
Die Pflichtabgabe erfolgt über Moodle. Dort erhalten Sie Eingabedaten, die Sie z.B. in `input.txt` kopieren. Die Ausgabe tragen Sie in das vorgesehene Textfeld ein, indem Sie z.B. den Inhalt von `output.txt` dorthin kopieren.

Graphviz[1]

In diesem und den folgenden beiden Praktika werden Sie das Werkzeug *Graphviz*[1] verwenden, um die Ausgabe Ihres Programmes z.B. in eine PDF-Datei zu zeichnen oder unter Linux direkt in einem X11-Fenster auszugeben. Unter der angegebenen URL[1] finden Sie Installationspakete und -anleitungen für mehrere Linux-Distributionen, für macOS sowie für Windows. Bitte installieren Sie das entsprechende Paket auf Ihrem Rechner.
Testen Sie die Installation mit folgenden Eingabedaten (links). Mit dem Befehl im Terminal

```
cat input.dot | neato -Tpdf > output.pdf
```

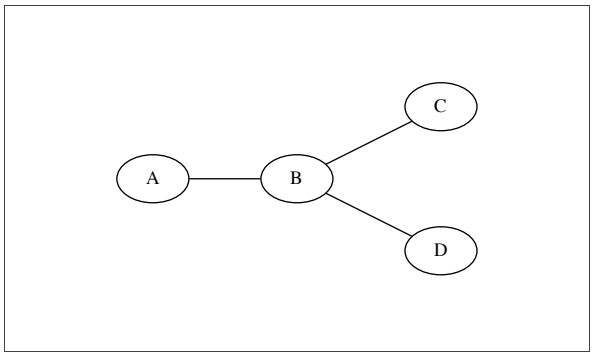
sollte sich die folgende Ausgabe (rechts) ergeben.

[1] https://www.graphviz.org

Eingabe (`input.dot`):

```
graph {
  A [pos="0,0!"]
  B [pos="1.5,0!"]
  C [pos="3,0.75!"]
  D [pos="3,-0.75!"]
  A -- B
  B -- C
  B -- D
}
```

Ausgabe (`output.pdf`):



popen (für Fortgeschrittene)

Aus Ihrem Programm können Sie andere Programme bzw. Werkzeuge starten und Ein- und Ausgaben über Standardein- und -ausgabe tätigen. Dazu können Sie unter Linux, macOS und anderen posix-konformen Betriebssystemen die Funktion popen verwenden[2]. Unter Windows können Sie _popen verwenden[3]. Damit können Sie den Aufruf von `neato` direkt in Ihr Programm integrieren und benötigen hierfür keine *Pipe*.
Wenn Sie sich für diese Variante entscheiden, sollten Sie auch überlegen, Ihr Programm Kommandozeilenargumente übernehmen und diese an `neato` weitergeben zu lassen. Damit können Sie beispielsweise unterschiedliche Ausgabeformate an der Kommandozeile wählen, was für die Abgabe durchaus sinnvoll ist.

Freispeicherverwaltung

In diesem Praktikum werden Sie Knoten verschiedenen Typs anlegen – die schnellen und die langsamen aus dem Netzwerk-Beispiel. Diese Knoten sind von einer gemeinsamen Basisklasse abgeleitet. Damit Sie diese in einem gemeinsamen Container verwalten und virtuelle Funktionen nutzen können, arbeiten Sie mit Zeigern auf diese Knoten. Damit das einfach funktioniert, allokieren Sie am besten Speicher zur Laufzeit. Dazu verwenden Sie in C++ `new` wie im folgenden Beispiel. Den erhaltenen Speicher müssen Sie selbst mit `delete` wieder freigeben.

Klassendeklaration:

```
class Foo {
  ...
};
```

Speicher allokieren und freigeben:

```
Foo* my_foo = new Foo;
...
delete my_foo;
```

`new` liefert Ihnen dabei die Adresse des neu angelegten Objekts. Diese können Sie in einen Container passenden Typs ablegen, um später auf alle angelegten Objekte zugreifen und den Speicher wieder freigeben zu können.

Smart-Pointer (für Fortgeschrittene)

Damit Sie sich nicht selbst darum kümmern müssen, den zur Laufzeit allokierten Speicher freizugeben, können Sie *Smart-Pointer* – genauer `std::unique_ptr` – verwenden. Diese stehen Ihnen seit C++11 zur Verfügung. Um diese etwas komfortabler anlegen zu können, sollten Sie zusätzlich die mit C++14 hinzugekommene Funktion `std::make_unique` verwenden.
Ein *Smart-Pointer* verwaltet die Lebenszeit des bezeichneten Objekts selbstständig und gibt den allokierten Speicher automatisch wieder frei. Das folgende Beispiel zeigt, wie ein solcher `std::unique_ptr` angelegt werden kann. Ein solcher *Smart-Pointer* lässt sich wie ein handelsüblicher Zeiger bedienen. Der Zugehörige Speicher wird automatisch am Ende des Code-Blocks freigegeben, ohne dass Sie selbst `delete` aufrufen:

```
{
  std::unique_ptr<Foo> my_foo{std::make_unique<Foo>()};
  my_foo->DoSomething();
  ...
}
```

[2] https://pubs.opengroup.org/onlinepubs/009695399/functions/popen.html
[3] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/popen-wpopen?view=vs-2019

II. Einstieg

Beginnen Sie mit einer Hello-World-Applikation. Die Aufgaben in diesem Teil bauen aufeinander auf.

1. Basisklasse

(a) Legen Sie in einem Paar aus Header- (`.hpp`) und Quelltextdatei (`.cpp`) eine Klasse `Foo` an.

(b) Legen Sie für die Klasse `Foo` einen öffentlichen Konstruktor an, der eine Zeichenkette und eine Ganzzahl übernimmt und diese in privaten Membervariablen speichert.

(c) Legen Sie in der Klasse `Foo` eine öffentliche Methode `std::string ToString() const` an, die sämtliche Membervariablen hintereinander in einen `std::string` schreibt und diesen anschließend zurückgibt. Verwenden Sie einen `std::stringstream`, um die geforderte Zeichenkette zu erstellen.

(d) Testen Sie die Klasse `Foo`. Erstellen Sie dazu in Ihrer `main`-Funktion mehrere Instanzen von `Foo`. Übergeben Sie jedesmal unterschiedliche Werte an den Konstruktor. Geben Sie für jede erstellte Instanz die durch die Methode `ToString` zurückgegebene Zeichenkette am Bildschirm aus.

2. abgeleitete Klasse

(a) Leiten Sie in einem neuen Paar aus Header- und Quelltextdatei von der Klasse `Foo` öffentlich die Klasse `Bar` ab.

(b) Legen Sie für die Klasse `Bar` ebenfalls einen öffentlichen Konstruktor an, der eine Zeichenkette und eine Ganzzahl übernimmt und diese direkt an den Basisklassenkonstruktor weitergibt.

(c) Legen Sie in der Klasse `Bar` ebenfalls eine öffentliche Methode `std::string ToString() const` an, die die Zeichenkette `BAR` sowie sämtliche an den Konstruktor übergebenen Werte hintereinander in einen `std::string` schreibt und diesen anschließend zurückgibt. Verwenden Sie einen `std::stringstream`, um die geforderte Zeichenkette zu erstellen. Verwenden Sie die Methode `ToString` der Basisklasse, um Code-Duplikate zu vermeiden.

(d) Testen Sie die Klasse `Bar`. Erstellen Sie dazu in Ihrer `main`-Funktion mehrere Instanzen von `Bar`. Übergeben Sie jedesmal unterschiedliche Werte an den Konstruktor. Geben Sie für jede erstellte Instanz die durch die Methode `ToString` zurückgegebene Zeichenkette am Bildschirm aus.

3. Container und virtuelle Funktionen

(a) Legen Sie in Ihrer `main`-Funktion einen Container vom Typ `std::vector<Foo*>` an.

(b) Legen Sie die Adressen mehrerer verschiedener `Foo`- und `Bar`-Instanzen in diesem Container ab.

(c) Rufen Sie in einer Schleife die `ToString`-Methode jeder durch die Container-Elemente bezeichneten Instanzen auf und geben die erhaltene Zeichenkette aus. Was fällt Ihnen auf?

(d) Machen Sie die `ToString`-Methode in der Klasse `Foo` virtuell. Warum ändert dies die Ausgabe?

(e) Legen Sie die `Foo`- und `Bar`-Instanzen dynamisch mit `new` zur Laufzeit an. Geben Sie am Ende der `main`-Funktion den Speicher mit `delete` wieder frei.

III. Aufgaben für die Pflichtabgabe

Beginnen Sie mit einer Hello-World-Applikation. Die Aufgaben in diesem Teil bauen aufeinander auf.

4. Knoten-Klasse

Legen Sie eine Klasse für Knoten an – diese wird später Basisklasse der schnellen und langsamen Knoten. Der Konstruktor übernimmt einen Knoten-Namen sowie eine 2D-Position (Fließkommazahlen) für das Zeichnen mit Graphviz.

Die Knotenklasse benötigt eine `ToString`-Methode, mit der der Knoten so ausgegeben wird, dass er von Graphviz verarbeitet werden kann. Die Ausgabe eines Knotens mit Namen `FOO` an der Position $x = 1,2$ und $y = 3,4$ lautet:

```
FOO [pos="1.2,3.4!"]
```

5. **abgeleitete Knoten-Klassen**

   Leiten Sie von Ihrer Knoten-Klasse eine Klasse für schnelle und eine Klasse für langsame Knoten ab. Auch diese benötigen bei Konstruktion einen Knoten-Namen und eine 2D-Position und müssen für Graphviz passend ausgegeben werden können.

   Die Ausgabe eines Knotens mit Namen `FOO` an der Position $x = 1,2$ und $y = 3,4$ lautet – links ein langsamer Knoten, rechts ein schneller Knoten:

   langsamer Knoten:

   ```
   FOO [pos="1.2,3.4!",shape=circle]
   ```

   schneller Knoten:

   ```
   FOO [pos="1.2,3.4!",shape=box]
   ```

6. **Graph[4]-Klasse**

   Legen Sie eine Klasse `Graph` an, die einen Container für Knoten enthält. Einer Instanz dieser Klasse sollen zur Laufzeit Knoten hinzugefügt werden können. Sämtliche enthaltenen Knoten sollen über eine `ToString`-Methode so ausgegeben werden können, dass Sie von Graphviz verarbeitet werden können.

   Die Ausgabe einer Instanz der Klasse `Graph` mit einem langsamen Knoten `FOO` an der Position $x = 1,2$ und $y = 3,4$ und einem schnellen Knoten `BAR` an der Position $x = 5,6$ und $y = 7,8$ lautet:

   ```
   graph {
     FOO [pos="1.2,3.4!",shape=circle]
     BAR [pos="5.6,7.8!",shape=box]
   }
   ```

7. **Verarbeitung von Eingabedaten**

   Ihr Programm verarbeitet nun Eingaben über die Standardeingabe:

   - Zunächst wird in einer einzelnen Zeile die Anzahl der anzulegenden Knoten angegeben.
   - Anschließend erfolgt die zeilenweise Eingabe von Knoten im Format

     ```
     [Typ] [Name] [x] [y]
     ```

     Hier umschließen eckige Klammern wie in den vorherigen Praktika das jeweils bezeichnete Eingabefeld. Der Typ `L` bezeichnet langsame Knoten, der Typ `H` bezeichnet schnelle Knoten.
   - Das Ende der Eingabe wird durch Drücken von Strg+D signalisiert.

   Die Eingabe links soll die Ausgabe in der Mitte erzeugen. `neato` macht daraus die Zeichnung rechts.

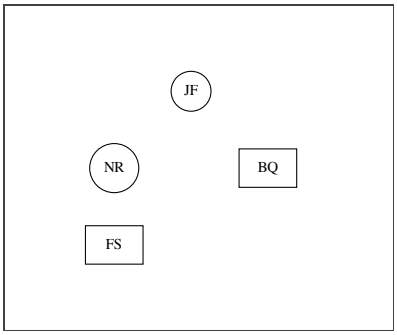
   Eingabe:

   ```
   4
   H BQ 2 1
   H FS 0 0
   L JF 1 2
   L NR 0 1
   ```

   Ausgabe Ihres Programms:

   ```
   graph {
     BQ [pos="2,1!",shape=box]
     FS [pos="0,0!",shape=box]
     JF [pos="1,2!",shape=circle]
     NR [pos="0,1!",shape=circle]
   }
   ```

   Ausgabe von `neato`:

[4] In der nächsten Vorlesung wird der Begriff *Graph* näher erläutert. Sie brauchen in diesem Praktikum noch nicht über Graphen Bescheid zu wissen.

IV. Vertiefung

Arbeiten Sie mit Ihrer Projektmappe aus Abschnitt III. weiter.

8. `Printable`-Konzept

In dieser Aufgabe lassen Sie eine Klasse ein Konzept implementieren, das an anderer Stelle benötigt wird. In diesem Fall ist es das Konzept `Printable`, das hier eine implementierte `ToString`-Methode fordert und die Klasse damit automatisch kompatibel für einen lediglich an einer Stelle implementierten Ausgabe-Stream-Operator macht.

(a) Erstellen Sie eine abstrakte Klasse `Printable`. Diese hat lediglich eine öffentliche, konstante *pure virtual*-Methode `ToString` ohne eigene Implementierung.

(b) Implementieren Sie in derselben Header-Datei, in der Sie die Klasse `Printable` angelegt haben, den Ausgabe-Stream-Operator für Instanzen von `Printable` als *inline*-Funktion.

(c) Leiten Sie Ihre Knoten-Basisklasse sowie Ihre Graph-Klasse öffentlich von `Printable` ab. Markieren Sie die jeweilige `ToString`-Methode mit `override`. Beheben Sie ggf. auftretende Compiler-Fehler.

(d) Löschen Sie (falls vorhanden) sämtliche Implementierungen des Ausgabe-Stream-Operators für Instanzen der Knoten-Klasse oder für von dieser abgeleitete Klassen. Lediglich die Implementierung des Ausgabe-Stream-Operators für `Printable`-Instanzen soll erhalten bleiben.

(e) Geben Sie nun (falls noch nicht geschehen) die Knoten- und Graph-Informationen mittels Ausgabe-Stream-Operator aus, ohne die Methode `ToString` selbst in Ihrem Code aufzurufen. Überlegen Sie, wieso dies möglich ist, obwohl Sie für keine der Knoten-Klassen einen Ausgabe-Stream-Operator implementiert haben.