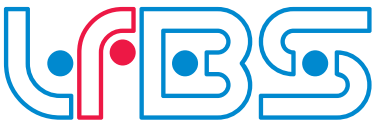
Praktikum Informatik 2
Aufgabenstellung

Wintersemester 2013/2014



Inhaltsverzeichnis

Literaturverzeichnis

Im folgenden Abschnitt finden Sie einen kleinen Auszug aus der zur Verfügung stehenden Literatur zu C++ und objektorientierter Programmierung.

Es sei hier ausdrücklich darauf hingewiesen, dass das Skript nur als Kurzreferenz dient und zur umfassenden Vorbereitung weitere Literatur zwingend erforderlich ist.

1. Stroustrup, Bjarne
   Die C++ Programmiersprache, 4. Auflage
   Addison Wesley 2000
   ISBN: 382731660X

2. Prinz, Peter
   C++- Lernen und professionell anwenden
   mitp 2008
   ISBN: 3826617649

3. Prinz, Peter
   C++- Das Übungsbuch.
   mitp 2007
   ISBN: 3826617657

4. Schildt, Herbert
   C++- Die professionelle Referenz
   mitp 2004
   ISBN: 3826613678

5. Josuttis, Nicolai
   Objektorientiertes Programmieren in C++
   Addison Wesley 2001
   ISBN: 3827317711

6. Breymann, Ulrich
   C++
   Hanser 2005
   ISBN: 3446402535

7. Kuhlins, Stefan
Die C++-Standardbibliothek. Einführung und Nachschlagewerk
Springer 2005
ISBN: 3540256938

8. Schneeweiß, Ralf
Moderne C++ Programmierung
Springer 2007
ISBN: 3540222812

Weitere Literaturhinweise finden Sie auf der Homepage (`http://www.lfbs.rwth-aachen.de/pi2`) unter Literatur und im Lehr- und Lernbereich (L^2P) der „Einführungsveranstaltung zum Praktikum“.

Dort finden Sie außerdem eine Auswahl von Links zu Online-Tutorien und Seiten mit grundlegenden und weiterführenden Informationen zu objektorientierter Programmierung unter C++.

1 Hinweise zur Benutzung des CIP-Pools unter Windows 7

Das Praktikum wird im CIP-Pool der Fakultät für Elektrotechnik und Informationstechnik durchgeführt. Sie finden den CIP-Pool in der 2. Etage des Seminargebäudes. Für die Durchführung des Praktikums wird das Betriebssystem „Windows 7“ und die Entwicklungsumgebung „Visual Studio 2012“ eingesetzt.

1.1 Anmelden im CIP-Pool

Voraussetzung zum Arbeiten mit den CIP-Pool-Rechnern ist, dass Sie eine entsprechende Benutzerkennung (Benutzername) haben. Diese erhalten Sie am ersten Praktikumstermin. In jedem Fall finden Sie Ihre Benutzerkennung auf Ihrem Testatbogen, den Sie am ersten Praktikumstermin bekommen.

1.2 Zur Bedienung das Wichtigste in Kürze

Da die meisten die Modalitäten des CIP-Pools bereits kennen, hier die wichtigsten Dinge in Kurzfassung:

Passwörter

Ihr Passwort für den Windows-Zugang ist beim Eintrag auf die Matrikelnummer gesetzt und muss beim ersten Login geändert werden. In der Startleiste finden sich unter Programme entsprechende Einträge. Passwörter müssen mindestens 7 Zeichen lang sein. Zwischen Groß- und Kleinschreibung wird unterschieden. Sollten Sie Ihr Passwort vergessen haben, können Sie es an der Informationstheke (bei Vorlage eines Lichtbildausweises) zurücksetzen lassen. Mehrfache Fehleingaben des Passwortes führen automatisch zu einer Sperre des Zugangs für diese Benutzerkennung. Die Sperre dauert 30 Minuten. In Ausnahmefällen kann die Sperre an der Informationstheke aufgehoben werden.

Dateien

Jeder Benutzer hat einen eigenen Speicherbereich auf dem Server, den er von allen Rechnern als Laufwerk U: erreichen kann. Maximal können dort **1 GB** gespeichert werden.

Allen Benutzern gemeinsam ist das Laufwerk Q: auf dem Server. Dort können Druckdateien, Dateien zum Datenaustausch mit anderen Benutzern oder größere Dateien gespeichert werden.

Achtung: Dateien dort kann jeder Benutzer lesen, ändern und löschen! Alle Dateien auf diesem Laufwerk werden jedes Wochenende gelöscht!

Auf alle anderen Dateien kann i.A. nur lesend zugegriffen werden. Unberechtigte Zugriffe auf fremde Dateien oder Systemdateien werden protokolliert und können ggf. zu unerfreulichen Nachfragen seitens der CIP-Pool-Betreiber führen.

Im Unterordner Ihrer Veranstaltung (hier: PI2) des Ordners „UserGrp“ auf dem Laufwerk P: finden Sie ggf. Dateien, die Ihnen als Teilnehmer einer bestimmten Veranstaltung bereitgestellt werden (z.B. Musterlösungen, Programmrahmen etc.)

Drucken

Ausdrucke können über die normalen Druckbefehle bzw. Icons in den Programmen erfolgen. Alle Ausdrucke sind **kostenpflichtig** (S/W 5 Cent, Farbe 25 Cent pro Seite). Wählen Sie bitte beim Ausdruck den gewünschten Drucker und holen Sie die Ausdrucke unmittelbar an der Theke ab.

Anmelden

Wenn das Dialogfeld **Willkommen** angezeigt wird, drücken Sie `STRG+ALT+ENTF`, um sich anzumelden.

Anschließend wird das Dialogfeld **Windows-Anmeldung** angezeigt. Geben Sie dort Ihren Benutzernamen (Kennung) und Ihr Kennwort ein.

Nach der Eingabe Ihrer Werte klicken Sie auf `OK` oder benutzen die Eingabetaste. Ihre Umgebung wird dann eingerichtet. Starten Sie danach Programme durch Aufruf aus der Menüleiste bei `Start` → `Alle Programme`.

Abmelden

Jeder Benutzer des CIP-Pools muss sich beim Beenden der Arbeit im CIP-Pool vom Rechner abmelden.

Klicken Sie dazu auf die Schaltfläche `Start` → `Herunterfahren` → `Abmelden`.

Das Abmelden ist für Sie wichtig, da sonst alle Handlungen, die mit diesem Computer gemacht werden, Ihnen angelastet werden. Dies kann für Sie unangenehm (sicherheitsrelevante Handlungen) oder teuer (Drucken) werden.

1.3 Dateien von/nach zu Hause kopieren

Sie können Ihre Praktikumsaufgabe zu Hause vor- bzw. nachbereiten. Dazu können Sie Ihre Quelldateien kopieren. Es stehen Ihnen dafür vorzugsweise selbst mitgebrachte USB-Sticks sowie alternativ auch E-Mail zur Verfügung.

Kopieren Sie bitte von Ihrer Praktikumsaufgabe **nur die .cpp und .h-Dateien**. Zur genauen Vorgehensweise siehe dazu auch die Erläuterungen im nächsten Abschnitt.

2 Visual Studio 2012

2.1 Einführung

Im Folgenden wird erklärt, wie Sie in der Entwicklungsumgebung von Visual Studio 2012 (VS 2012) ein kleines „Hello World“-Programm eingeben. Starten Sie VS 2012 über das Start-Menü. Danach wird ein neues Projekt erstellt, indem unter `Datei` → `Neu` → `Projekt` → `Visual C++` → `Win32` *Win32-Konsolenanwendung* ausgewählt wird. Als Projektname geben Sie *Test* an.

Geben Sie bitte der Einheitlichkeit wegen bei *Speicherort* immer Ihr Homeverzeichnis an. Ihr Homebereich liegt auf Laufwerk `U:` (s. Abb. 2.1).

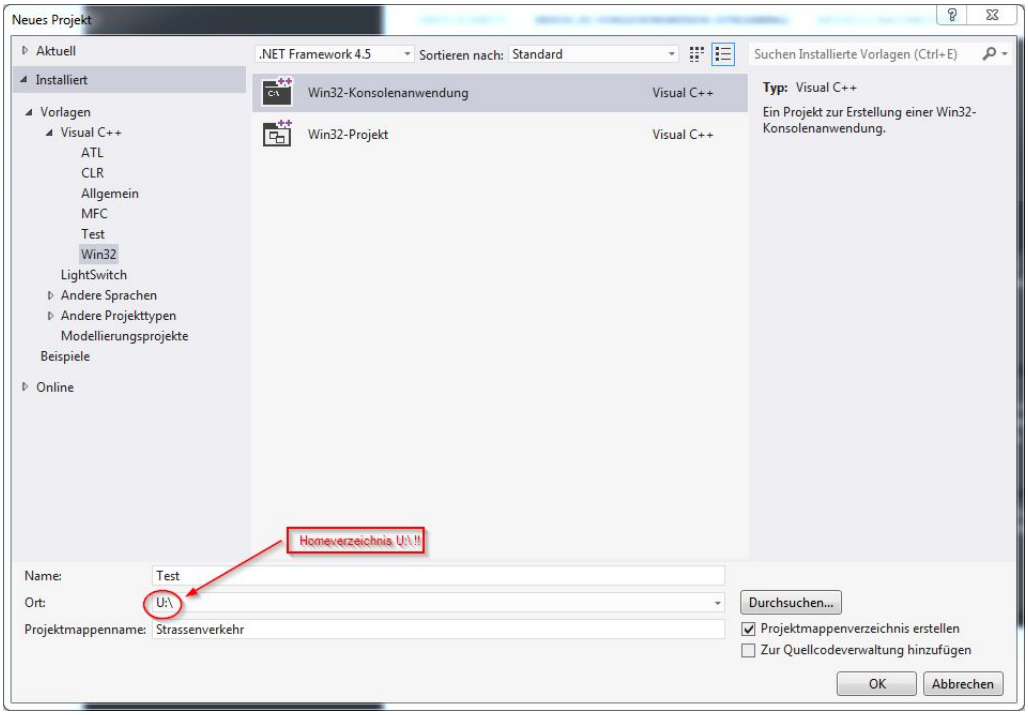


Abbildung 2.1: Neues Projekt erstellen

Im nun erscheinenden *Win32-Anwendungs-Assistent* gehen Sie zunächst auf `Weiter`.

Dann wählen Sie im nächsten Fenster unter *Anwendungseinstellungen* unter dem Unterpunkt *Zusätzliche Optionen* den Eintrag *Leeres Projekt* aus (s. Abb. 2.2). Nun können Sie den Dialog zur Projekterstellung mit `Fertig stellen` beenden.

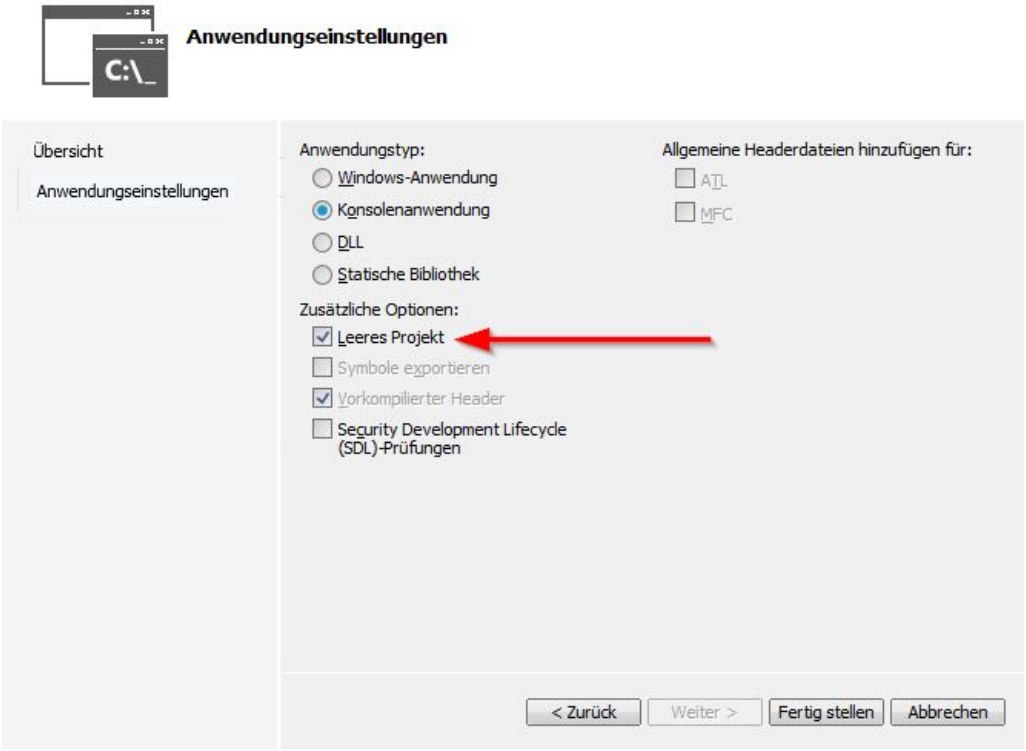


Abbildung 2.2: Leeres Projekt erstellen

VS 2012 erkennt automatisch, dass ihr Homeverzeichniss ein Netzlaufwerk repräsentiert und setzt automatisch wichtige Umgebungsvariablen für den Speicherort von temporären Daten auf ein lokales Verzeichnis. Dies wird Ihnen in einem Dialog mitgeteilt, der lediglich mit `OK` zu bestätigen ist, siehe Abb. 2.3.

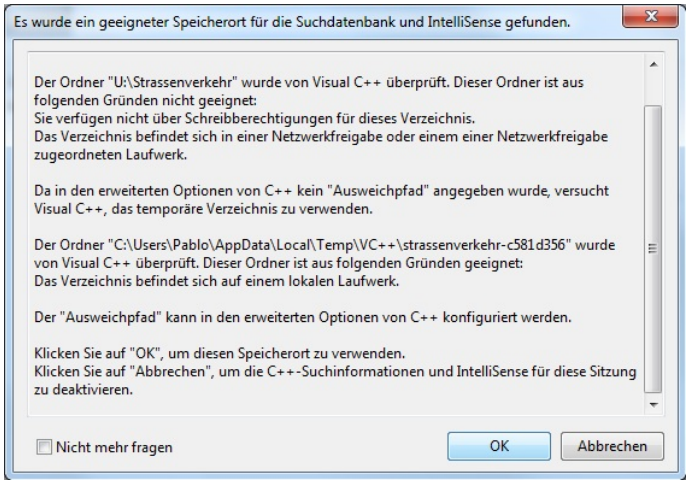


Abbildung 2.3: Hinweis Speicherort Netzlaufwerk

Die wichtigsten Funktionen können über Buttons erreicht werden. Wenn der Mauszeiger kurze Zeit bewegungslos auf einem der Buttons steht, wird die Funktion angezeigt,

die diesem Button entspricht. Verschiedene Tools können an einer unbenutzten Stelle des Toolbarbereichs mit einem Rechtsklick ein- oder ausgeblendet werden.

Folgende Fenster sollte ab jetzt immer eingeschaltet sein: *Projektmappen-Explorer*, *Klassenansicht* und *Ausgabe*. Diese finden Sie unter `Ansicht`. In der Toolbar genügt zunächst die Standard-Option.

Um das „Hello World"-Programm einzugeben, fügen Sie ein neues C++-File dem Projekt hinzu, indem Sie im Menü `Projekt` → `Neues Element hinzufügen` → `Code` C++ -*Datei* wählen. Geben Sie als Name `main` ein und schreiben nun folgendes Programm in das neue Textfenster:

```
#include <iostream>
using namespace std;

void main()
{
  cout << "Hello World" << endl;
}
```

Speichern Sie alles ab, übersetzen Sie das Projekt und starten Sie es. Suchen Sie die entsprechenden Buttons oder Einträge in den Menüs (`Datei, Erstellen, Debuggen`).

Als nächstes wird beschrieben, wie Sie Klassen hinzufügen und bearbeiten. In Visual Studio 2012 ist es üblich, dass für jede Klasse eine Header-Datei (Datei mit Endung .h) und ein CPP-Datei angelegt wird. In der Header-Datei stehen gewöhnlich die Deklarationen der Klasse und in der CPP-Datei die Implementierung der Methoden. Das Anlegen dieser Files sowie ein Skelett für Klassendefinition, Konstruktor und Destruktor erzeugt VS 2012 automatisch.

Erstellen Sie als erstes eine neue Klasse `MyClass` (s. Abbildung 2.4). Wählen Sie dazu unter `Projekt` → `Klasse hinzufügen` → `C++` die Vorlage C++ -*Klasse* aus. Nennen Sie die neue Klasse `MyClass`. Verwenden Sie immer einen virtuellen Destruktor. Die neue Klasse wird automatisch in den Workspace eingefügt.

Implementieren Sie außerdem eine Funktion `print` in die Klassenhierarchie (s. Abbildung 2.5). Diese Funktion bekommt den Rückgabetyp `void` und wird als `public` deklariert. Markieren Sie dazu in der Klassenansicht die Klasse `MyClass` und wählen Sie im Kontextmenü („rechte Maustaste") `Hinzufügen` → `Funktion` aus (auch bei Projekt im Menü erreichbar).

Mit einem Doppelklick auf entsprechende Objekte im Workspace kann schnell zu bestimmten Klassen, Methoden oder Variablen gesprungen werden. Doppelklicken Sie in der Klassenansicht beispielsweise auf `MyClass(void)`, öffnet sich ein Editorfenster mit der cpp-Datei an der Stelle der Implementation des Konstruktors der Klasse `MyClass`.

Fügen Sie im Konstruktor und Destruktor eine Ausgabe ein, um beim Programmlauf zu sehen, dass beide Funktionen aufgerufen werden. Geben Sie in `print()` das gewünschte „Hello World" aus. Definieren Sie dann in `main()` eine Instanz der Klasse `MyClass` und rufen Sie dessen Member Funktion `print()` auf. In Abbildung 2.6 sind alle Programmtexte und der Workspace zu sehen.

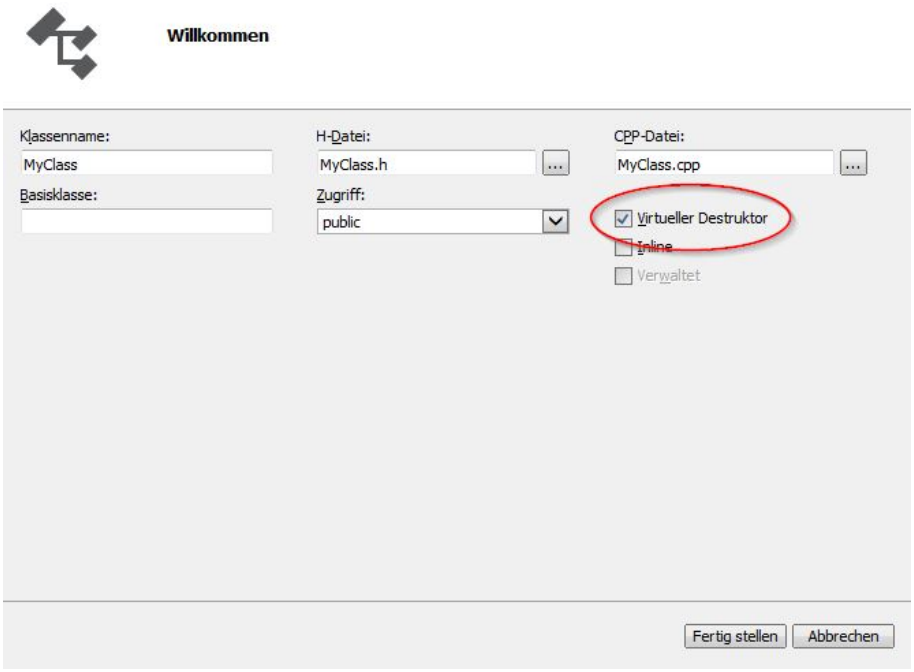


Abbildung 2.4: Klasse erstellen

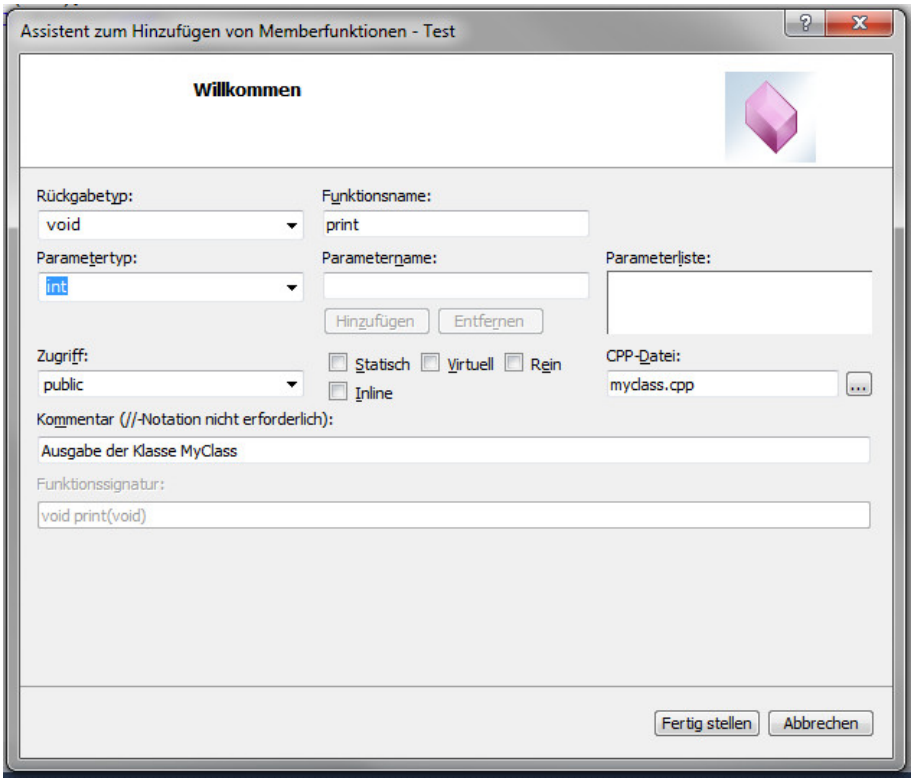


Abbildung 2.5: Funktion einfügen

Speichern Sie alle Files ab (alles speichern Button), übersetzen Sie das Projekt (`Erstellen`) und, falls keine Fehler und keine Warnungen (0 Fehler, 0 Warnung(en) in der Ausgabe, siehe Abbildung 2.6 unten) auftreten, starten Sie es (Starten ohne Debugging) mit der Tastenkombination `Strg + F5`. Es wird ein Fenster geöffnet, in dem folgende Zeilen ausgegeben werden:

```
Konstruktor
Hello World
Destruktor
Drücken Sie eine beliebige Taste...
```

Die ersten Schritte im Programm zum Anlegen einer Klasse sind auch nochmal in einem Video dargestellt. Dieses finden Sie im L^{2}P unter `Lernmaterialien` → `Videos`.

2.2 Allgemeine Hinweise und Tipps

Als Grundregel gilt wie so oft: Wo Hilfe draufsteht ist meist auch Hilfe drin :). Am wichtigsten ist wohl die kontextsensitive Hilfe, die mit `F1` aktiviert wird. Versteht man den Inhalt eines Fensters nicht, einfach Fenster anklicken und `F1` drücken, oder braucht man Hilfe zu einem C++-Befehl, einfach den Cursor im Textfenster auf das entsprechende Wort stellen und F1 drücken. Hat man ein konkretes Problem, kann man `Hilfe` → `Suchen` benutzen.

Alles was durch Rechtsklicks oder Buttons bewirkt wird, kann auch mit Hilfe der Pulldownmenüs in der obersten Zeile erreicht werden. Die Menüpunkte sind meist selbsterklärend, viele sind erst für Fortgeschrittene interessant. Aktionen, die sehr oft ausgeführt werden, erreicht man am schnellsten mit sogenannten Shortcuts. Diese stehen in den Standardmenüs rechts neben dem Text der Aktion. Beispielsweise funktioniert *Rückgängig* (letzte Aktion rückgängig machen) außer über das Menü `Bearbeiten` auch mit `Ctrl+Z` (Control-Taste gedrückt halten und dann Z drücken). Weitere Beispiele sind `F7` für *Projektmappe erstellen* oder `F5` für *Debuggen starten*.

Wenn Sie Dateien mit nach Hause nehmen oder von dort importieren möchten, beachten Sie bitte, dass Sie nur Quellcode- und Header-Dateien transferieren. Die Extension für Quellcodedateien muss „.cpp“ und für Header-Dateien „.h“ sein und müssen so heißen wie die darin implementierte Klasse. Kopieren Sie diese Dateien in Ihr Projektverzeichnis. Falls Sie neue Dateien importieren, müssen Sie diese noch dem Projekt bekannt machen: `Projekt` → `Vorhandenes Element hinzufügen`.

2.3 Debugger

Ein Debugger ermöglicht es, den internen Ablauf von Programmen genauer zu verfolgen und erleichtert damit insbesondere das Auffinden von Fehlern (sogenannten *bugs*), da man das Soll-Verhalten des Programms mit dem tatsächlichen Ablauf Schritt für Schritt vergleichen kann.

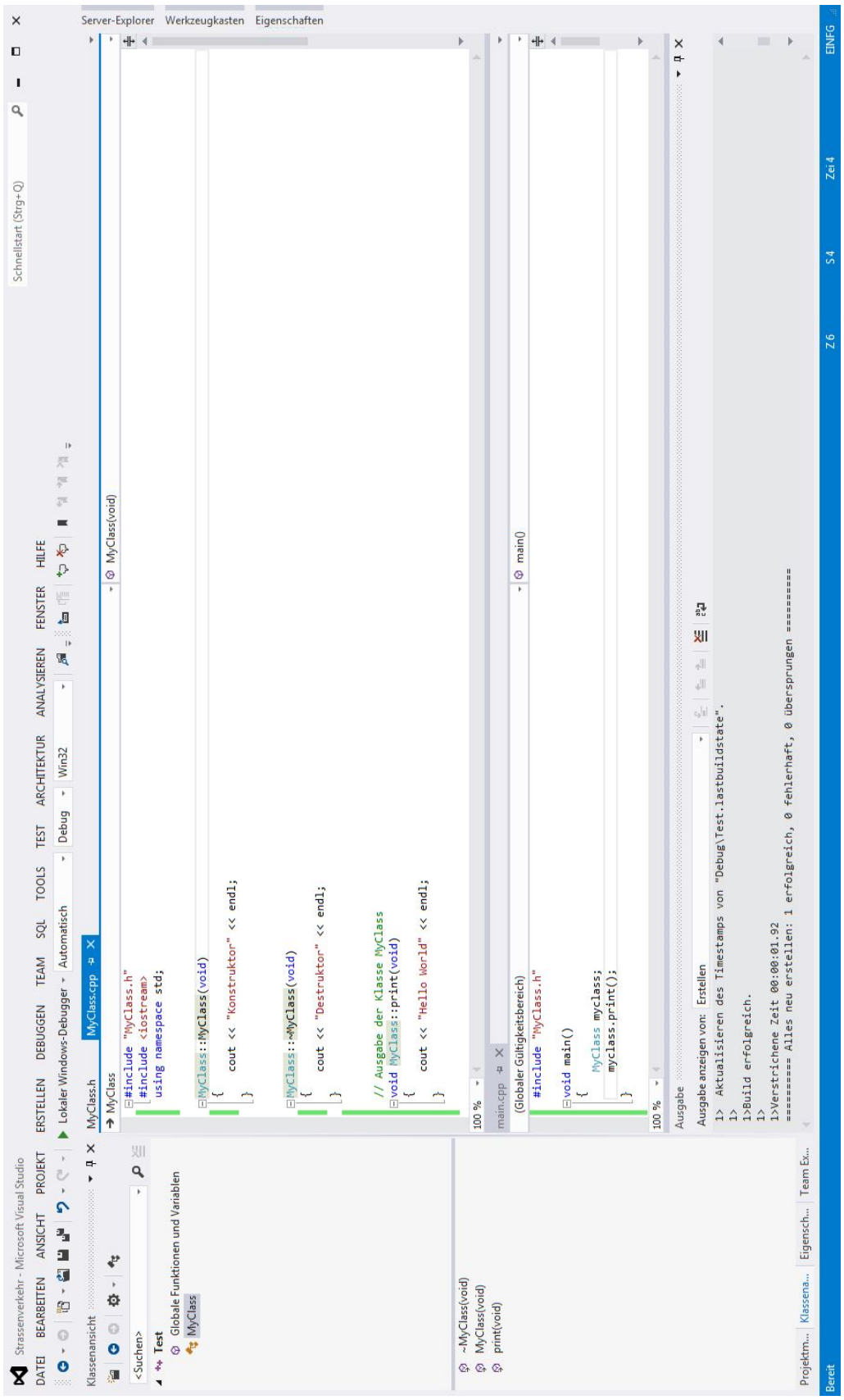


Abbildung 2.6: Komplettes Programm

2.3.1 Grundlagen

Eine allgemeine Vorgehensweise ist, zunächst mit Hilfe des Debuggers im untersuchten Programm sogenannte *Haltepunkte* zu setzen, d.h. die Stellen im Programmcode festzulegen, an denen der Programmlauf unterbrochen werden soll. Wird beim Programmlauf ein Haltepunkt erreicht, so wird die Programmausführung gestoppt und der Debugger zeigt mit einer Markierung auf die entsprechende Stelle im Quellcode. Während eines solchen Programmstopps kann man sich dann über den aktuellen Zustand des Programms informieren. So kann man sich z.B. die Werte von Variablen anzeigen lassen. Indem man nach einem Programmstopp die Ausführung schrittweise fortführt, kann man u.a. verfolgen, welche Programmzweige ausgeführt werden, welche Funktionen aufgerufen werden und wie sich Variablenwerte im weiteren Programmablauf ändern. Durch diese Beobachtungen kann man erkennen, ob – und wenn ja – wo der Programmablauf nicht mit dem beabsichtigten übereinstimmt. Daraus lassen sich dann Rückschlüsse ziehen, welche Teile des Programms fehlerhaft sein könnten und verändert werden sollten.

2.3.2 Die Bedienung des Debuggers

Damit der Debugger in einer Programmzeile anhält, muss, wie zuvor beschrieben, dort ein *Haltepunkt* gesetzt werden. Dazu geht man mit dem Cursor in die gewünschte Zeile und kann nun mit `F9` oder einem Mausklick links neben die Zeile einen Haltepunkt einfügen bzw. einen vorhandenen Haltepunkt wieder entfernen.

Eine nützliche Eigenschaft von Haltepunkten bei der Fehlersuche ist die Bedingung. Damit kann der Programmablauf gezielt unterbrochen werden, z. B. innerhalb einer Schleife in einem bestimmten Schritt:

```
for( int i=0; i < MAX_IT; i++ ) {
  /* ... */
}
```

Die Bedingung für einen Haltepunkt kann mit einem Rechtsklick auf das Symbol neben der markierten Zeile gesetzt werden. Entweder wird der Programmablauf unterbrochen falls ein gewisses Ereignis eintritt (z. B. `i==3`) oder falls ein Wert sich ändert (z. B. `i`).

Anschließend kann unter `Debuggen` → `Debuggen starten` (`F5`) der Debugger gestartet werden, d.h. das Programm wird gestartet und hält vor Abarbeitung der Zeile, in der sich der Haltepunkt befindet, an. Weiterhin wird die Debug-Funktionsleiste in der Toolbar eingeblendet.

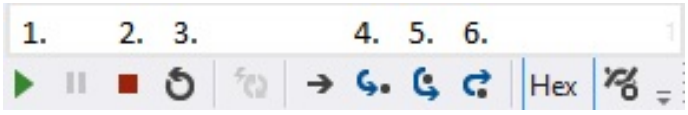


Abbildung 2.7: Debuggen-Funktionsleiste

Die wesentlichen Buttons dieser Funktionsleiste sind:

1. *Weiter.* Ausführung bis zum nächsten Haltepunkt.

2. *Debuggen beenden.*
3. *Neu starten.* Debugger beginnt den Programmdurchlauf von Vorne.
4. *Einzelschritt.* Programm wird zeilenweise ausgeführt. Es wird in Unterprogramme verzweigt.
5. *Prozedurschritt.* Programm wird zeilenweise ausgeführt. Unterprogramme werden übersprungen (aber ausgeführt).
6. *Ausführen bis Rücksprung.* Springt aus einem Unterprogramm zur aufrufenden Funktion zurück

Um sich einen Überblick über die Variableninhalte oder Funktionsaufrufe während eines Programmdurchlaufs zu verschaffen, können verschiedene Fenster eingeblendet werden. Diese finden Sie unter `Debuggen` → `Fenster`.

- *Überwachen* (1-4): Hier kann man die Werte ausgewählter Variablen überwachen.
- *Auto*: Zeigt immer die Variablenwerte der aktuell ausgeführten Programmzeile

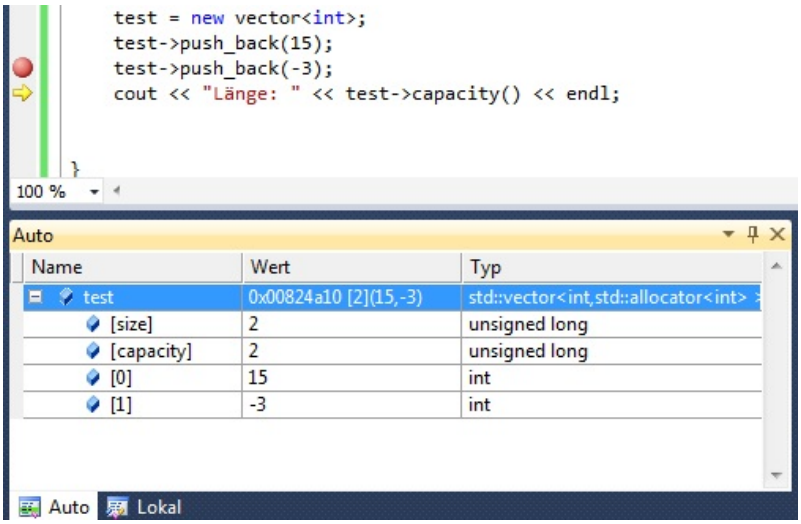


Abbildung 2.8: Debuggen Auto-Fenster

- *Lokal*: Hier werden lokale Variablen für den aktuellen Kontext oder Gültigkeitsbereich angezeigt
- *Aufrufliste*: Hier werden die Funktions- und Prozeduraufrufe angezeigt, die sich derzeit im Stapel befinden.

Sie können mehrere Fenster gleichzeitig anzeigen lassen oder je nach Bedarf Fenster öffnen.

Um den Umgang mit dem Debugger etwas zu üben, ist in der Aufgabenstellung dazu eine eigene Unteraufgabe eingefügt (Aufgabe 1.10). Diese ist von jedem Teilnehmer zu programmieren. Falls Sie noch keine Erfahrung im Umgang mit einem Debugger haben, empfehlen

wir, etwas mehr Zeit in diesen Unterpunkt zu investieren. Testen Sie verschiedene Sprungmöglichkeiten und benutzen Sie die unterschiedlichen Anzeigefenster, um etwas Routine im Umgang mit dem Debugger zu bekommen.

Die Benutzung des Debuggers bei der Fehlersuche ist ein Lernziel des Praktikums und geht daher mit in die Bewertung der Aufgabenblöcke ein

3 Aufgaben

3.1 Motivation

Während des Praktikums sollen alle wesentlichen Elemente objektorientierter Softwareentwicklung und ihre Umsetzung im Sprachumfang von C++ an einem (vereinfachten) Beispiel eingesetzt und geübt werden. Zur Darstellung oft benutzter Datenstrukturen wie Vektor, Liste oder Assoziativspeicher sollen die Klassen der STL (Standard Template Library) kennengelernt und benutzt werden.

Die Aufgabe besteht aus neun aufeinander aufbauenden Teilaufgaben, die schließlich zu der Gesamtlösung führen. Die einzelnen Aufgaben sind zu drei Blöcken mit jeweils drei Aufgaben zusammengefasst. Für jeden Block wird ein Testat abgenommen. Die Aufgaben sollen so implementiert werden, dass für jeden Block ein neues Projekt mit Visual Studio 2012 angelegt wird. Dazu wird der vorhandene Block kopiert und dann erweitert. Somit sollte ein Testprogramm aus Block1 auch am Ende des Praktikums noch funktionieren (außer es ist aufgaben-technisch nicht möglich).

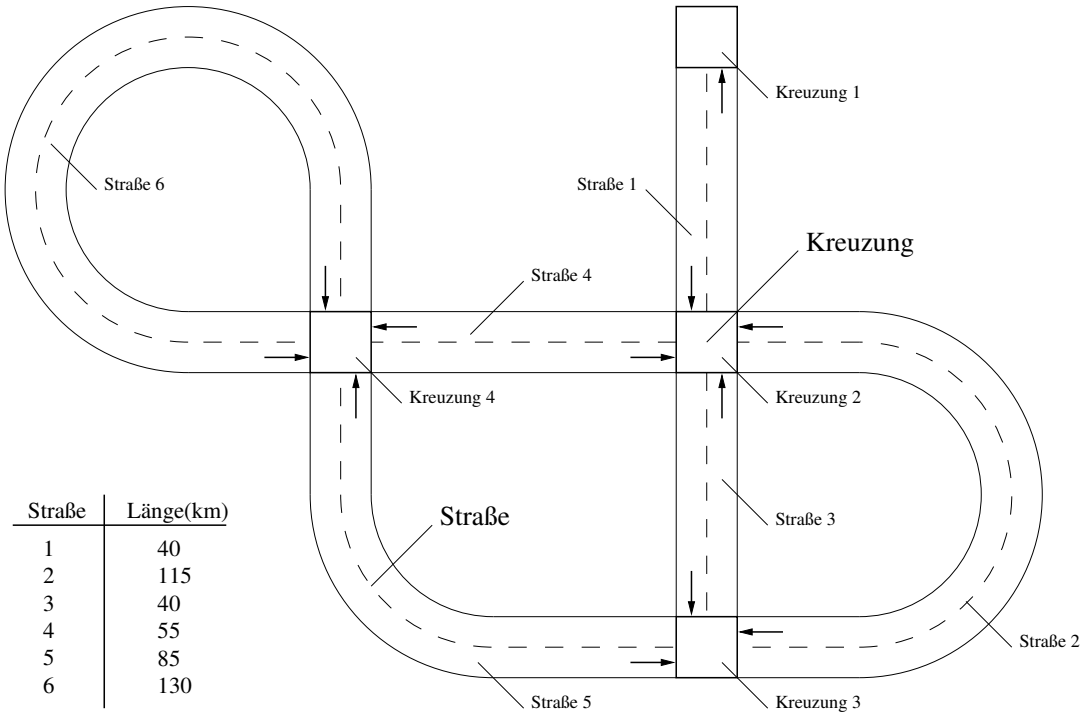
Alle drei Aufgabenblöcke sollen in **einer** gemeinsamen Projektmappe liegen. Darauf wird bei den einzelnen Abnahmen geachtet und es geht auch mit in die Bewertung ein.

3.2 Zielaufgabe

Es soll der Straßenverkehr in einer wenig erschlossenen Gegend modelliert und simuliert werden (s. Abbildung 3.1).

Verschiedene Arten von Fahrzeugen (PKW, Fahrrad) werden zu einem individuellen Startzeitpunkt von einem Knotenpunkt losgeschickt. Jedes Fahrzeug besitzt einen Zeit- und einen Streckenzähler sowohl für die Gesamtstrecke als auch für den Streckenabschnitt, auf dem es sich gerade befindet. Die Daten des Streckennetzes und der eingesetzten Fahrzeuge werden eingelesen.

Das Modell setzt sich aus drei verschiedenen Verkehrsobjekten zusammen: Fahrzeuge, Wege und Kreuzungen. Eine Verbindung zwischen zwei Kreuzungen wird durch eine Straße realisiert, die aus zwei entgegengerichteten Wegen (Hin- und Rückspur bzw. einlaufender und ausgehender Weg) gebildet wird. Jeder Weg verwaltet eine Liste, welche die auf dem Weg befindlichen Fahrzeuge enthält, jede Kreuzung eine Liste der aus dieser Kreuzung abgehenden Wege. Wege können sowohl fahrende als auch parkende Fahrzeuge annehmen. Zum Startzeitpunkt werden aus parkenden Fahrzeugen fahrende Fahrzeuge.



Straße	Länge(km)
1	40
2	115
3	40
4	55
5	85
6	130

Abbildung 3.1: Simulationsmodell

Alle Verkehrsobjekte beinhalten eine Abfertigungsfunktion. Kreuzungen fertigen dabei die von ihnen abgehenden Wege, Wege die auf ihnen befindlichen Fahrzeuge ab. Zu jedem Zeitschritt werden also durch Abfertigung aller Kreuzungen des Systems nacheinander alle Verkehrsobjekte abgefertigt. Das System wird durch einen globalen Zeittakt gesteuert. In jedem Zeittakt werden alle im System befindlichen Objekte genau **einmal** abgefertigt. Dies wird erreicht, indem jeweils die lokale Zeit mit der globalen Zeit synchronisiert wird. Im Straßensystem herrscht teilweise Überholverbot, d.h. in einem Simulationsschritt darf ein Fahrzeug die Position des vorausfahrenden Fahrzeugs auf bestimmten Wegen nicht überschreiten.

Diese Simulation kann man durch die in Abbildung 3.2 dargestellte Klassenstruktur realisieren. Obwohl sicher auch andere Strukturen und Implementationen möglich wären, gehen wir im Praktikum von dieser Struktur aus. Funktionen werden nur **einmal** aufgeführt, und zwar jeweils in der hierarchisch am höchsten gelegenen Klasse, d.h. sie können durchaus in abgeleiteten Klassen Verwendung finden. Bitte verwenden Sie in Ihrer Implementierung die dort aufgeführten Klassen- und Methodennamen, um den Betreuern bei Fragen eine schnelle Orientierung zu ermöglichen. Selbstverständlich können (und müssen) Sie zur Implementierung und zum Test einzelner Module weitere Klassen und/oder Funktionen einführen.

Die Aufgaben bauen aufeinander auf, so dass am Ende des Praktikums die Simulation komplett implementiert ist. Die Funktionen für die grafische Ausgabe werden Ihnen zur Verfügung gestellt. In der Grafik sind PKWs durch rote, Fahrräder durch grüne Punkte dargestellt.

Bevor Sie mit den Aufgaben beginnen, lesen Sie bitte die gesamte Aufgabenstellung durch, damit Sie wissen, wozu Klassen und Funktionen später genutzt werden.

Im Anschluss finden Sie noch einige Vorgaben und Programmierhinweise.

3.3 Hinweise zur Implementierung

3.3.1 Verwaltung der Verkehrssimulation

Alle Aufgaben sollen innerhalb einer Projektmappe *Strassenverkehr* in Ihrem Userverzeichnis (`U:\`) angelegt werden. Die Projektmappe wird zusammen mit dem ersten Projekt angelegt. Die 9 Aufgaben sind in 3 Aufgabenblöcke unterteilt. Jeder Aufgabenblock soll als Projekt *Aufgabenblock_X(1,2,3)* vom Typ *Win32-Konsolenanwendung* in der Projektmappe angelegt werden. In der zugehörigen `main()`-Funktion soll dann für jede Aufgabe eine entsprechende Funktion `vAufgabe_X()` aufgerufen werden, welche die Funktionalität der entsprechenden Aufgabe testet. Diese Funktion würde der `main()`-Funktion eines separaten Projektes entsprechen.

Unter `Datei` → `Neu` → `Projekt` → `Visual C++` → `Win32` erzeugen Sie zu Beginn gleichzeitig ein neues Projekt *Aufgabenblock_1 (Win32-Konsolenanwendung)* und die Projektmappe *Strassenverkehr*. Markieren Sie *Projektmappenverzeichnis erstellen*. Beim nun erscheinenden *Win32-Anwendungs-Assistenten* klicken Sie auf `Weiter` und markieren im folgenden Dialog *Leeres Projekt* (keine vorkompilierten Header!). Nun das Projekt `Fertig stellen`.

In Ihrem Userverzeichnis (`U:\`) sollte jetzt ein Verzeichnis *Strassenverkehr* mit dem Unterverzeichnis *Aufgabenblock_1* existieren.

3.3.2 Neues Projekt hinzufügen

Wie oben beschrieben, wird innerhalb der Projektmappe *Strassenverkehr* für jeden weiteren Aufgabenblock ein neues Projekt angelegt. Dazu werden die Sourcen des alten Projekts übernommen und dann erweitert. Da in der Vergangenheit dabei immer wieder Probleme aufgetreten sind, gibt es an dieser Stelle nun eine detaillierte Anleitung:

1. Erzeugen Sie **innerhalb** Ihrer Projektmappe *Strassenverkehr* ein neues Projekt mit dem Namen *Aufgabenblock_X (Win32-Konsolenanwendung)*. Diesen Punkt finden Sie unter `Datei` → `Hinzufügen` → `Neues Projekt ...` . In Ihrem Userverzeichnis wird dadurch ein neues Unterverzeichnis *Aufgabenblock_X* angelegt.

2. Kopieren Sie nun mit dem Windows-Dateiexplorer die Sourcen aus dem Verzeichnis des alten Projekts in das neue Projektverzeichnis (wichtig: es sollen nur die *.h und *.cpp Dateien kopiert werden!!!).

3. In Visual Studio 2012 müssen nun die kopierten Dateien dem neuen Projekt noch bekannt gemacht werden. Legen Sie das neue Projekt als Startprojekt fest und gehen nach `Projekt` → `Vorhandenes Element hinzufügen ...` .

   Sollten Sie die Aufgaben auch zu Hause bearbeiten, gelten dieselben Regeln. Bringen Sie nur die *.h und *.cpp Dateien mit zum Praktikum und aktualisieren damit ihr Projekt.

   Immer wenn Sie einem Projekt neue Daten hinzugefügt haben, aktualisieren Sie das Projekt einschließlich der zugehörigen Datenbanken mit `Erstellen` → `Projektmappe neu erstellen`.

3.3.3 Namenskonvention

Benutzen Sie bitte in all Ihren Lösungen folgende Präfixe für Variablen und Funktionen. Es erleichtert Ihnen (und auch uns) das Lesen des Quellcodes, da aus dem Namen unter anderem auch schon der Typ der Variablen/Funktionen ersichtlich ist.

- `protected` und `private` Variablen werden durch ein `p_` ganz vorne am Variablennamen gekennzeichnet.
- Darauf folgen ein oder zwei Buchstaben, die den Typ der Variablen bzw. den Rückgabewert der Funktion beschreiben.

  ```
  i=int; t=struct; d=double; v=void; b=bool;
  s=string; p=pointer; e=enum;
  ```

- Danach folgt dann der eigentliche Variablenname, wobei der erste Buchstabe eines jeden Teilwortes groß geschrieben wird.
- Im speziellen Fall, dass die Funktion eine private/protected Variable setzt oder zurückliefert, ist das erste Wort des eigentlichen Variablennamens `Set` bzw. `Get`.
- Eine Funktion wird dadurch gekennzeichnet, dass dem Namen runde Klammern folgen.

Beispiele:

- `p_iID`: protected/private-Variable vom Typ `int`
- `bIstFertig()`: Funktion, die einen boolschen Wert zurückliefert (`true` oder `false`)
- `vFunktion()`: Funktion, die nichts (`void`) zurückliefert.

3.3.4 Programmierhinweise

- Implementieren Sie für jede Klasse eine Datei *Klassenname.h* zur Deklaration der Variablen und Funktionen. Weiterhin jeweils eine Datei *Klassenname.cpp* zur Definition des Codes. Dies geschieht automatisch, wenn Sie in Visual Studio 2012 den Menüpunkt `Projekt` → `Klasse hinzufügen` (C++-Klasse) verwenden.
- Neben den Funktionen, die zur Lösung der Aufgaben vorgegeben werden, können Sie natürlich zusätzlich noch eigene Funktionen implementieren.
- Kommentieren Sie Ihre Programme ausreichend, sodass auch Außenstehende (Betreuer) Ihren Code nachvollziehen können. Dieser Punkt geht auch mit in die Bewertung ein.
- Achten Sie bei dynamisch angelegten Instanzen darauf, dass diese auch wieder gelöscht werden.
- Entscheiden Sie, ob es bei der Definition von Funktionen, Variablen oder Parametern sinnvoll ist, diese als `const` zu deklarieren.
- Schreiben Sie `using namespace std;` überall wo Sie Elemente der STL verwenden hinter die jeweiligen Includes. Bedenken Sie, dass dies nicht die Regel ist. Siehe dazu auch Kapitel 2 im Skript.
- Verwenden Sie für alle Variablen, die eine Strecke (km) oder eine Zeit (h) verwalten, den Datentyp `double`.
- Alle Dateien, die wir Ihnen im Laufe des Praktikums zur Verfügung stellen, finden Sie im CIP-Pool unter P:\UserGrp\PI2 bzw. im Lehr- und Lernbereich (L2P) des Praktikums unter Lernmaterialien.
- Denken Sie an die aufeinander aufbauende Programmstruktur der Aufgabe (s. Kapitel 3.1)

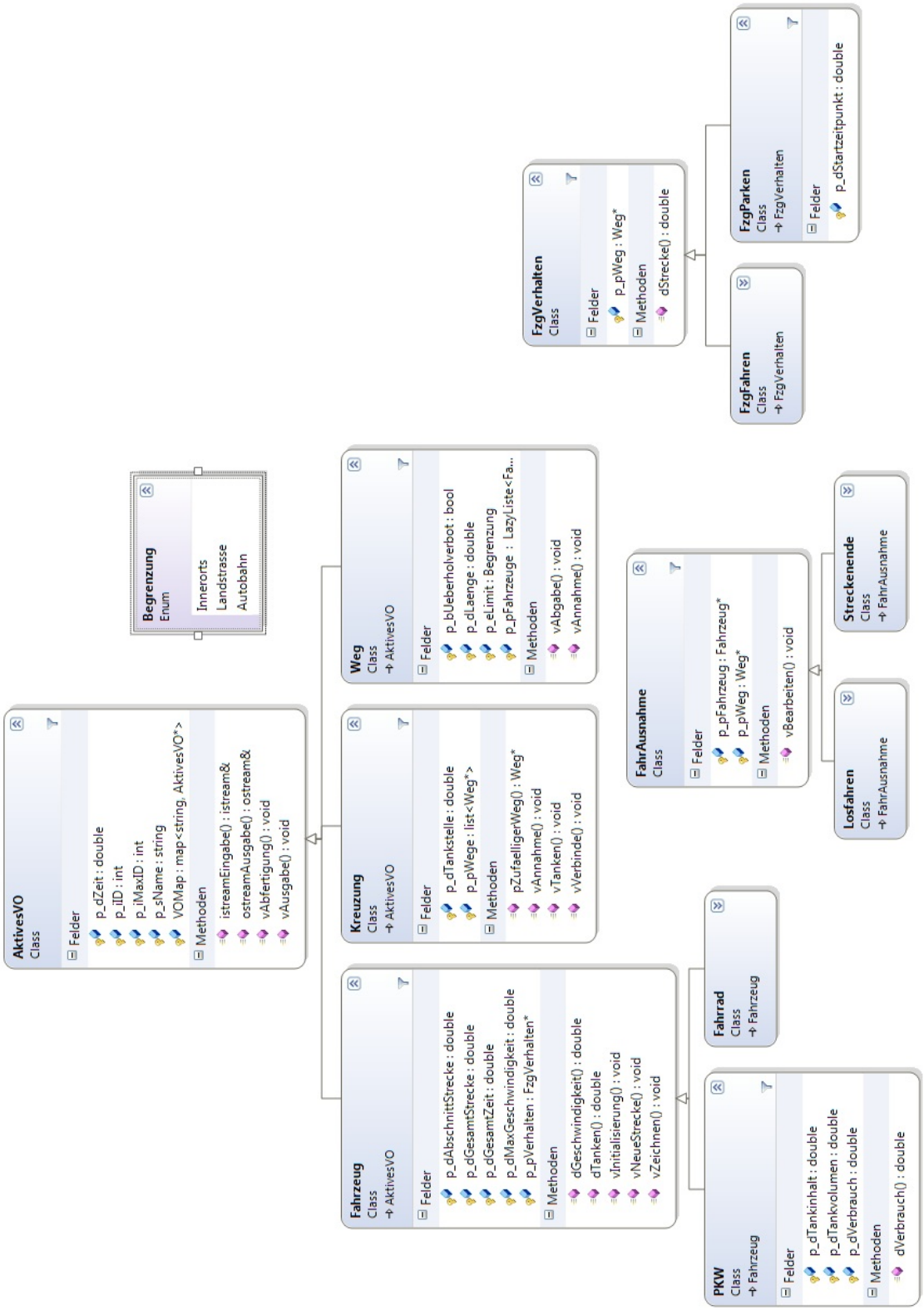


Abbildung 3.2: Klassenhierarchie

Aufgabenblock 1: Grundlagen des Verkehrssystems

3.3.5 Motivation

In diesem ersten Aufgabenblock werden Klassen für die zu simulierenden Fahrzeuge erstellt, PKWs und Fahrräder, die sich selbst abfertigen/fortbewegen können. Ein Mini-Eventhandler ruft eine entsprechende Abfertigungsmethode aller Fahrzeuge mehrmals auf und gibt den aktuellen Stand der Fahrzeuge nach jedem Schritt auf dem Bildschirm aus.

Um sich einen Überblick zu verschaffen, lesen Sie den ersten Aufgabenblock zunächst komplett durch

3.3.6 Lernziele

- Deklaration und Definition von Klassen
- Implementierung von Konstruktoren und Destruktoren
- Kapselung von Daten und Zugriff auf private Member
- Verwendung von static Variablen
- Vererbung
- Einsatz der STL (string, vector)
- Unterscheidung der Klassenbereiche public, private, protected
- Unterscheidung einfache/virtuelle/rein virtuelle Vererbung
- Überladen von Operatoren

Aufgabe 1: Fahrzeuge (Einfache Klassen)

1. Starten Sie Visual Studio 2012 und erstellen Sie in Ihrem Homebereich ein neues Projekt mit dem Namen *Aufgabenblock_1*. Gleichzeitig erstellen Sie dabei auch eine Projektmappe mit dem Namen *Strassenverkehr*. Siehe auch Kapitel 3.3.1.

2. Implementieren Sie eine Klasse `Fahrzeug` zur Verwaltung verschiedener Fahrzeuge. Die Klasse soll zunächst lediglich private Data Member haben, in denen der Name des Fahrzeugs (`p_sName`) und eine ID (`p_iID`) zu jedem Objekt gespeichert wird. Benutzen Sie für den Namen den Datentyp `string`. Die ID soll im Konstruktor aufgrund einer hochzählenden Klassenvariablen `p_iMaxID` vergeben werden, d.h. jedes Objekt bekommt eine eindeutige Nummer.

   Implementieren Sie einen Standardkonstruktor, der den Namen mit einer leeren Zeichenkette (`""`) initialisiert. Definieren Sie einen weiteren Konstruktor, der einen Namen als string bekommt. Geben Sie (zum Test) in den Konstruktoren und dem Destruktor eine Meldung aus, welche den Namen der erzeugten bzw. gelöschten Objekte enthält.

3. Beim Programmieren ist es meist ratsam, schnell ein lauffähiges Programm zu haben. Erzeugen Sie eine neue C++-Datei (`main.cpp`), die die Funktion `vAufgabe_1()` aufruft und implementieren Sie diese Funktion innerhalb der Datei `main.cpp`. Erzeugen Sie in dieser Funktion einige Elemente statisch (über Deklaration) und einige dynamisch (mit `new`), wobei Sie jeweils beide Konstruktoren verwenden sollen. Lassen Sie für die dynamisch erzeugten Fahrzeuge mit Konstruktorparameter für den Fahrzeugnamen diesen interaktiv eingeben. Am Ende der Funktion löschen Sie die dynamisch erzeugten Objekte wieder (in anderer Reihenfolge!). Erzeugen und starten Sie das Programm und testen Sie das korrekte Erzeugen und Löschen der Objekte.

4. Erweitern Sie die Klasse um Membervariablen für die mögliche Maximalgeschwindigkeit des Fahrzeugs (`p_dMaxGeschwindigkeit`), die bisher zurückgelegte Gesamtstrecke (`p_dGesamtStrecke`), die gesamte Fahrzeit des Objektes (`p_dGesamtZeit`) und die Zeit, zu der das Fahrzeug zuletzt abgefertigt wurde (`p_dZeit`).

   Fügen Sie einen weiteren Konstruktor hinzu, der einen Namen und die maximale Geschwindigkeit als Parameter bekommt. Da Sie zur Vorbesetzung von Variablen die verschiedenen Konstruktoren einer Klasse nicht gegenseitig aufrufen können, bietet sich, bei Vorhandensein vieler Konstruktoren mit sich wiederholenden Zuweisungen, die Implementierung einer Initialisierungsfunktion an. Daher schreiben Sie die **private** Funktion `vInitialisierung()`, die alle Variablen mit `0` bzw. `""` vorbesetzt und die ID hochzählt. Rufen Sie diese Funktion zu Beginn jedes Konstruktors auf.

5. Da dieses Programm noch nicht viel am Bildschirm ausgibt, schreiben Sie eine Memberfunktion `vAusgabe()`. Diese Memberfunktion soll fahrzeugspezifische Daten ausgeben. Die Ausgabe soll so formatiert werden, dass unter einer Überschrift, die vom Hauptprogramm ausgegeben wird, die Daten tabellarisch aufgelistet werden, in etwa folgendermaßen:

   ```
   ID   Name    :   MaxKmh    GesamtStrecke
   ++++++++++++++++++++++++++++++++++++++++
   1    PKW1    :   40.00          0.00
   2    AUTO3   :   30.00          0.00
   ```

   Benutzen Sie für die Formatierung keine feste Anzahl von Leerzeichen, sondern die IO-Manipulatoren der Standard C++ Bibliothek (`<iomanip>`).

   Beachte: Bei Verwendung von `setiosflags()` zum Setzen der Ausgabeausrichtung (rechts-/linksbündig) sollte zunächst die andere Ausrichtung mittels `resetiosflags()` zurückgesetzt werden.

6. Testen Sie diese neue Memberfunktion, indem Sie Ihre Hauptfunktion erweitern und die erzeugten Fahrzeuge mit Hilfe der neuen Memberfunktion ausgeben.

7. Bevor die Abfertigungsfunktion der Fahrzeuge geschrieben werden kann, muss erst noch eine globale Uhr programmiert werden, damit die Fahrzeuge wissen, wieviele Zeiteinheiten (Stunden) sie abfertigen sollen. Zur Realisierung dieser Uhr definieren Sie innerhalb von `main.cpp` eine globale Variable `dGlobaleZeit`, die Sie mit `0.0` initialisieren.

Beachte: Vor Benutzung dieser Variablen innerhalb anderer Klassen muss sie der Klasse erst mittels der `extern`-Deklaration bekannt gemacht werden.

8. Schreiben Sie nun die Memberfunktion `Fahrzeug::vAbfertigung()`, welche dafür sorgt, dass die Fahrzeuge sich fortbewegen. Dazu wird mit Hilfe der globalen Uhr ermittelt, wieviel Zeit seit der letzten Abfertigung vergangen ist, und entsprechend dieser Information wird der Zustand des Fahrzeugs aktualisiert. Sorgen Sie durch einen Zeitvergleich dafür, dass ein Fahrzeug in einem Zeitschritt nur *einmal* abgefertigt wird, auch wenn es versehentlich zweimal innerhalb eines Zeitschritts aufgerufen wird. Lassen Sie das Fahrzeug mit maximaler Geschwindigkeit fahren.

9. Erweitern Sie Ihre Hauptfunktion: Fertigen Sie Fahrzeuge über eine gewisse Zeitspanne ab. Erhöhen Sie dazu in einer Schleife im Hauptprogramm `vAufgabe_1()` die globale Uhr jeweils um einen Zeittakt und fertigen Sie in der Schleife die Fahrzeuge ab. Wählen Sie als Zeittakt auch Bruchteile von Stunden. Geben Sie die jeweiligen Fahrzeugdaten nach jeder Abfertigung mit der Funktion `vAusgabe()` aus.

10. Im letzten Unterpunkt sollen Sie die grundlegenden Möglichkeiten des Debuggers zur Fehlersuche nutzen. Dies soll auch bei der Abnahme vorgeführt werden! Die Benutzung des Debuggers wird in Kapitel 2.3 beschrieben.

    Hierzu schreiben Sie eine weitere Funktion `vAufgabe_1_deb()` und erzeugen 4 verschiedene Fahrzeuge. Die Zeiger dieser Fahrzeuge legen Sie in einem **Feld** ab. Zur Kontrolle durchlaufen Sie das Feld und geben die Daten aller beinhalteten Fahrzeuge aus. Weisen Sie nun dem vorletzten Feldelement eine Null zu (`feld_name[2] = 0`). Wiederholen Sie die Ausgabe der Fahrzeugdaten. Was passiert?

    Um den (hier offensichtlichen) Fehler zu finden, setzen Sie einen Haltepunkt in die zweite Ausgabeschleife und starten das Programm erneut. Durchlaufen Sie dann Schritt für Schritt das Programm. Machen Sie sich mit den verschiedenen Sprungmöglichkeiten des Debuggers bekannt.

    Beobachten Sie außerdem in einem Fenster für die Variablenüberwachung den Inhalt der Variablen `feld_name[i]`. Was fällt auf, wenn das vorletzte Feldelement erreicht wird?

Aufgabe 2: Fahrräder und PKW (Unterklassen, vector)

1. Implementieren Sie zwei neue Klassen `PKW` und `Fahrrad`, die jeweils von der Basisklasse `Fahrzeug` abgeleitet werden. Überlegen Sie, welche Variablen `private` bleiben sollten und welche `protected` werden. Überlegen Sie weiterhin, welche Funktionen `virtual` werden. Die Testausgaben im Konstruktor/Destruktor der Klasse `Fahrzeug` können Sie wieder auskommentieren.

2. Wenn es keine Unterschiede zwischen PKWs und Fahrrädern geben würde, wäre es sinnlos, sie mit zwei verschiedenen Klassen zu unterscheiden. Fügen Sie der Klasse `PKW` die Variablen `p_dVerbrauch` (Verbrauch/100 km), `p_dTankinhalt` (in Liter) sowie `p_dTankvolumen` (55 l, falls nicht anders initialisiert) hinzu. Der Tankinhalt wird

jeweils auf die Hälfte des Tankvolumens initialisiert. Weiterhin bekommt die Klasse `PKW` eine Methode `dVerbrauch()`, die den bisherigen Gesamtverbrauch ermittelt.

Ergänzen Sie die Klasse um einen entsprechenden Konstruktor, mit dem Sie zusätzlich zu den fahrzeugspezifischen Membervariablen auch Verbrauch und (optional) Tankvolumen setzen können. Nutzen Sie für die Einbeziehung der Konstruktoren der Basisklasse eine Initialisierungsliste.

Des Weiteren schreiben Sie eine Funktion `dTanken` mit optionalem Parameter `dMenge` zum nachträglichen Betanken der PKWs. Wird kein Wert übergeben (Defaultparameter), soll vollgetankt werden, ansonsten wird der gewünschte Wert getankt. Beachten Sie, dass maximal das Tankvolumen aufgefüllt werden kann. Geben Sie jeweils die tatsächlich getankte Menge zurück. Implementieren Sie die Funktion in der Klassenhierarchie so, dass für alle Unterklassen von `Fahrzeug` automatisch entschieden wird, ob getankt wird oder nicht (Fahrräder und Fahrzeuge ohne Tank tanken bekanntlich nicht, d.h. die Funktion macht nichts und gibt immer 0 Liter zurück).

Bei jedem Abfertigungsschritt soll der Tankinhalt aktualisiert werden, bis der Tank leer ist. PKWs ohne Tankinhalt sollen liegenbleiben bis wieder nachgetankt wird. Dann sollen sie normal weiterfahren. Zur Vereinfachung soll die Reserve so groß sein, dass der PKW im letzen Abfertigungsschritt noch die komplette Teilstrecke fahren kann. Implementieren Sie dazu für `PKW` eine eigene Funktion `vAbfertigung()`, die die zusätzliche Funktionalität von PKW implementiert. Für die allgemeine Abfertigung soll aber weiterhin `Fahrzeug::vAbfertigung()` aufgerufen werden. Der Gesamtverbrauch und der aktuelle Tankinhalt sollen außerdem noch in `vAusgabe()` ausgegeben werden.

Beachte: Um Codeduplizierung in den abgeleiteten Klassen zu vermeiden, sollen die Daten, die zu `Fahrzeug` gehören, immer von `Fahrzeug::vAusgabe()` ausgegeben werden. Verwenden Sie diese Funktion also auch in den Ausgabefunktionen der abgeleiteten Klassen.

3. Da Fahrradfahrer nicht immer mit maximaler Geschwindigkeit fahren können, soll eine Memberfunktion `dGeschwindigkeit()` implementiert werden. Sie wird in `Fahrzeug` als virtuell deklariert und in `PKW` und `Fahrrad` überschrieben. PKWs sollen immer mit ihrer vollen Geschwindigkeit fahren, Fahrradfahrer dagegen werden langsamer. Jeweils ausgehend von der gefahrenen Gesamtstrecke soll die Geschwindigkeit pro 20 km um 10 % abnehmen, minimal jedoch 12 km/h betragen. Während eines Berechnungsschritts ist die Geschwindigkeit als konstant anzusehen. *Beispiel:* Nach 50 gefahrenen Kilometern beträgt die Geschwindigkeit im nächsten Zeittakt noch 81 % der Maximalgeschwindigkeit, falls diese noch mehr als 12 km/h beträgt.

   Stellen Sie nun `Fahrzeug::vAbfertigung()` auf diese Funktion um (statt Maximalgeschwindigkeit). Schreiben Sie eine Funktion `vAusgabe()` die für jedes `Fahrzeug`, zusätzlich zu den Fahrzeugdaten die aktuelle Geschwindigkeit ausgibt.

4. Schreiben Sie eine neue Funktion `vAufgabe_2()`: Lesen Sie die Anzahl der zu erzeugenden PKWs und Fahrräder ein, erzeugen Sie dynamisch entsprechende Objekte der Klassen `PKW` und `Fahrrad`. Speichern Sie die Zeiger auf die erzeugten Fahrzeuge in

einem `vector` der STL. Fertigen Sie diese Objekte mehrmals ab. Nach 3 Stunden tanken Sie die PKWs nochmals voll (im Testprogramm, nicht innerhalb von `dTanken()`). Geben Sie die Ergebnisse (Daten aller Fahrzeuge) nach jeder Abfertigung aus.

Beachte: Testen Sie Zeiten auf Gleichheit immer über den Absolutbetrag der Differenz gegen eine Toleranz ϵ, da Fließkomma-Werte aufgrund der Rundung fast nie genau gleich werden. Die Funktion für den Absolutbetrag `fabs()` finden Sie in der Bibliothek `<math.h>`.

Aufgabe 3: Ausgabe der Objekte (Operatoren überladen)

1. Wie man fundamentale Datentypen (`char`, `int`, `double`,...) mit Hilfe des Ausgabeoperators ausgibt haben Sie ja schon programmiert. Nun wollen wir die Ausgabe für ein Objekt definieren. Dazu müssen Sie für die entsprechende Klasse (`Fahrzeug`) den Ausgabeoperator `operator<<()` überladen.

   Implementieren Sie daher zunächst in die bestehende Klassenhierarchie eine weitere Ausgabefunktion `ostreamAusgabe()`. Diese Memberfunktion bekommt einen `ostream` (Referenz) als Parameter übergeben. Dieser stream wird mit allen fahrzeugspezifischen Daten (s. `vAusgabe()`) beschrieben und als Referenz (keine Kopie!!) wieder an die aufrufende Funktion zurückgegeben. Überladen Sie nun den Ausgabeoperator, indem Sie dort die Funktion `ostreamAusgabe()` verwenden.

   Beachte: Überladen Sie den Operator außerhalb der Klasse. Warum? Kommen Sie mit einer einzigen Definition für alle von Fahrzeug abgeleiteten Klassen aus? Verwenden Sie bitte keine `friend`-Deklaration.

   Testen Sie den Ausgabeoperator, indem Sie Fahrzeuge, PKWs und Fahrräder damit auf `cout` ausgeben:

   Beispiel: `cout << aPKW << endl << aFahrrad << endl;`

   Verwenden Sie ab jetzt zur Ausgabe von Daten nur noch den <<-Operator.

2. Überladen Sie in der Klasse `Fahrzeug` den Vergleichsoperator `operator<()`. Dieser soll den Wert `true` liefern, falls die bisher zurückgelegte Gesamtstrecke vom aktuellen Objekt kleiner als die vom Vergleichsobjekt ist.

3. Überdenken Sie für die Klasse `Fahrzeug` die Verwendung des *Copy-Konstruktors* und des *Zuweisungsoperators* `operator=()` (Was passiert mit den Member-Variablen z.B. `p_iID` ?). Sollte man hier die Standardformen (byteweise kopieren) benutzen, eigene definieren oder die versehentliche Benutzung ganz verhindern? Realisieren Sie die für Sie sinnvollste Variante und testen Sie, ob der gewünschte Zweck erreicht wird. Begründen Sie Ihre Entscheidung.

4. Verwenden Sie alle in dieser Aufgabe neu erstellten Operatoren in Ihrer Hauptfunktion `vAufgabe_3()`.

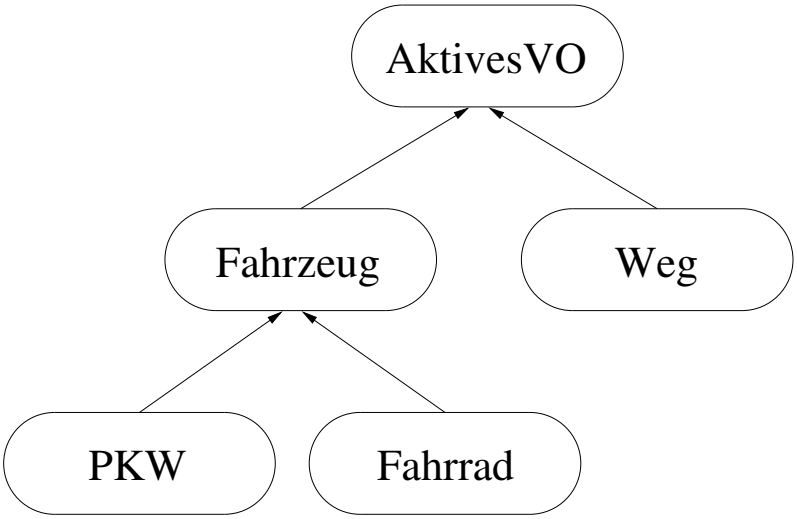


Abbildung 3.3: Klassenhierarchie Aufgabenblock 2

Aufgabenblock 2: Erweiterung des Verkehrssystems

3.3.7 Motivation und Lernziele

In diesem zweiten Aufgabenblock wird die Klassenhierarchie um eine Klasse `Weg` erweitert. Da diese Klasse einige Eigenschaften mit Fahrzeugen gemeinsam hat (Name, Abfertigungszeit, Abfertigunsfunktion, Ausgabefunktion usw.), ist es sinnvoll, die Klassenhierarchie um eine abstrakte Oberklasse von `Fahrzeug` zu erweitern und `Weg` von dieser Klasse abzuleiten. Die gemeinsamen Dienste werden dann in diese abstrakte Oberklasse verlagert. Dies ist eine bei der objektorientierten Programmierung häufig auftretende Situation.

Ein Weg verwaltet eine Liste von Fahrzeugen und kann sich abfertigen, indem alle auf dem Weg befindlichen Fahrzeuge abgefertigt werden.

Für die Berechnung der Strecke, die ein Fahrzeug in einem Simulationsschritt zurücklegt, wird eine neue Klasse erstellt, ein sogenanntes Verhaltensmuster. Jedes Fahrzeug kennt eine Instanz dieser Klasse und kann in seiner Abfertigung diese Instanz fragen, wie weit es fahren darf. Auftretende Sondersituationen (parkendes Fahrzeug fährt los, fahrendes Fahrzeug kommt am Ende des Weges an) werden durch Ausnahmebehandlung (Exceptions) abgehandelt. Um die Simulation etwas anschaulicher zu machen, wird eine Bibliothek mit Funktionen zur grafischen Darstellung zur Verfügung gestellt. Diese soll im Programm benutzt werden.

Oft wiederkehrende Datenstrukturen und Algorithmen können durch Templates allgemein beschrieben werden. Die STL stellt eine Fülle solcher vorgefertigter Strukturen bereit. Einige davon sollen hier benutzt werden. Schließlich soll für eine spezielle Listenart (verzögerte Aktualisierung) ein Template selbst erstellt werden.

Um sich einen Überblick zu verschaffen, lesen Sie den zweiten Aufgabenblock zunächst komplett durch

Aufgabe 4: Verkehrsobjekte, Wege und Parken (Oberklasse, list)

1. Fügen Sie der bereits vorhandenen Projektmappe *Strassenverkehr* ein neues Projekt vom Typ *Win32-Konsolenanwendung* mit dem Namen *Aufgabenblock_2* hinzu. Kopieren Sie mit dem Windows-Dateiexplorer alle Sourcen (nur *.h und *.cpp Dateien!!) aus *Aufgabenblock_1* und machen Sie diese Dateien dem neuen Projekt bekannt (s. Kapitel 3.3.2).

2. Als erstes soll eine neue abstrakte Oberklasse `AktivesVO` (AktivesVerkehrsobjekt) geschaffen werden, welche die gemeinsamen Eigenschaften von `Fahrzeug` und einer neuen Klasse `Weg` zusammenfasst. Fahrzeuge und Wege sind aktive Verkehrsobjekte, die einen Namen, eine ID und eine lokale Zeit besitzen. Sie können abgefertigt und ausgegeben werden. Integrieren Sie bitte `Fahrzeug` in diese neue Klassenhierarchie, indem Sie die Variablen für Name, ID und lokale Zeit sowie alle Funktionen zur gemeinsamen Nutzung von `Fahrzeug` und `Weg` aus der Klasse `Fahrzeug` in die Klasse `AktivesVO` übertragen.

   Überlegen Sie, welche Methoden/Variablen `private`, `protected` oder `public`, welche Methoden virtuell oder rein virtuell sein sollten. Beachten Sie, dass Methoden in `Fahrzeug` angepasst bzw. gelöscht werden müssen. `AktivesVO` ist eine abstrakte Klasse, besitzt also eine rein virtuelle Methode. Überlegen Sie, welche Funktion hierzu am Besten geeignet ist.

   Beachte: Um Codeduplizierung zu vermeiden, sollen bei der Ausgabe die entsprechenden `ostreamAusgabe()` -Methoden der übergeordneten Klassen mitbenutzt werden. Dieses Prinzip gilt auch für die Konstruktoren. So soll etwa `ostreamAusgabe()` in Fahrrad zunächst die Methode von `Fahrzeug` aufrufen, diese zunächst die Methode von `AktivesVO`.

   `AktivesVO::ostreamAusgabe()` soll nur die ID und den Namen des Objekts ausgeben.

3. Richten Sie die Klasse `Weg` als Unterklasse von `AktivesVO` ein. Wege haben zusätzlich zu den geerbten Eigenschaften eine Länge in km (`p_dLaenge`), eine Liste von Fahrzeugen (`p_pFahrzeuge`), welche sich aktuell auf dem Weg befinden und eine maximal zulässige Geschwindigkeit `p_eLimit`. Die Liste beinhaltet *Zeiger* auf Fahrzeugobjekte *(Warum können/sollten Sie keine Fahrzeugobjekte speichern?)*. Zur Implementierung benutzen die Containerklasse `list` der STL.

   Es soll für Wege drei unterschiedliche Kategorien (Innerorts, Landstraße und Autobahn) mit unterschiedlichem Geschwindigkeit Limit (50 km/h, 100 km/h und Unbegrenzt) geben. Definieren Sie dazu einen eigenen Datentyp `Begrenzung` als statische Aufzählung (`enum`).

   `Weg` soll einen Standardkonstruktor und einen Konstruktor mit Namen, Länge und optionalem Geschwindigkeitslimit (default unbegrenzt) als Parameter haben. Außerdem soll die Funktion `vAbfertigung()` so implementiert werden, dass beim Aufruf alle auf dem Weg befindlichen Fahrzeuge abgefertigt werden.

Beachte: Wenn zwei Klassen jeweils Variablen der anderen als Element enthalten, wie hier `Fahrzeug` und `Weg`, können Sie nicht in beiden Headerdateien jeweils die andere Headerdatei inkludieren, da dies zu einer Rekursion führen würde. Es reicht, in den Headerdateien jeweils den *Namen* der Klassen bekannt zu machen, also einfach `class Fahrzeug;` bzw. `class Weg;`. In den cpp-Dateien müssen aber dann die entsprechenden Headerdateien eingebunden werden, da dort die Prototypen der Funktionen benötigt werden.

Implementieren Sie eine Funktion `ostreamAusgabe()` für `Weg`, damit der überladene Ausgabeoperator verwendet werden kann. Die Funktion soll die ID, den Namen und die Länge des Weges ausgeben.

Beispiel: `3 Weg1 100`

4. Da Fahrzeuge *später* auf verschiedenen Wegen fahren sollen, führen wir hier eine zusätzliche Membervariable `p_dAbschnittStrecke` ein. Diese speichert immer nur die auf dem aktuellen Weg zurückgelegte Strecke. Fügen Sie diese Variable Ihren Berechnungen und Ausgaben für `Fahrzeug` hinzu.

5. Testen Sie Ihr altes Hauptprogramm. Es sollte noch unverändert funktionieren. In `vAufgabe_4()` testen Sie zusätzlich die neue Klasse `Weg`, indem Sie einen Weg erzeugen und ihn mit dem `<<`-Operator auf die Standardausgabe ausgeben.

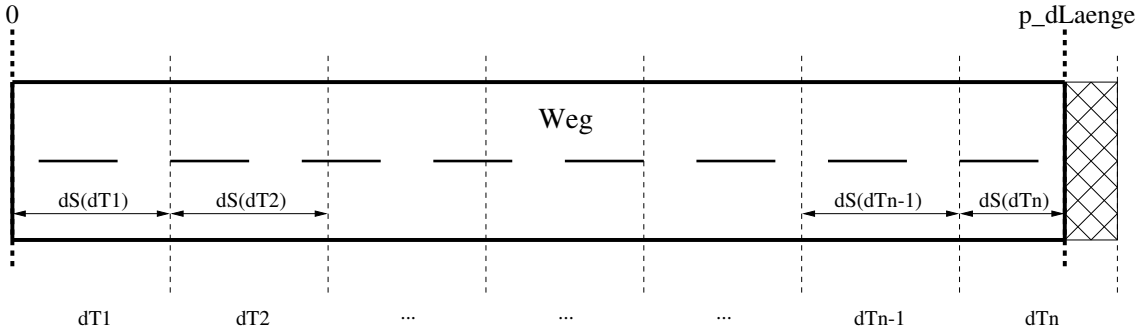
6. Damit man für ein Fahrzeug verschiedene Verhaltensweisen realisieren kann, wird diese Klasse um eine Membervariable `p_pVerhalten` (*Zeiger*), die eine Instanz der noch zu implementierenden Klasse `FzgVerhalten` speichert, erweitert. Durch Austausch dieses Objektes kann das Verhalten des Fahrzeugs verändert werden, ohne ein neues Fahrzeug erstellen zu müssen. Initialisieren Sie `p_pVerhalten` mit einem sinnvollen Wert.

   Unter Verhalten verstehen wir z. B. , dass Fahrzeuge unter bestimmten Bedingungen nicht immer die theoretisch mögliche Strecke fahren oder dass zwischen fahrenden und parkenden Fahrzeugen unterschieden werden kann.

   Daher implementieren Sie nun eine neue Klasse `FzgVerhalten`. Da das Verhalten u.a. vom jeweiligen Weg abhängt, bekommt die Klasse einen Konstruktor, der einen Zeiger auf Weg als Parameter bekommt und speichert. Weiterhin soll eine Funktion `dStrecke(Fahrzeug*, double)` angeboten werden, die ermittelt, wie weit ein Fahrzeug innerhalb des übergebenen Zeitraums (`double`) fahren kann. Die bisherige Berechnung der aktuellen Teilstrecke in `Fahrzeug::vAbfertigung()` wird also durch den Aufruf der Funktion `dStrecke()` ersetzt. Beachten Sie, dass `dStrecke()` in jedem Abfertigungsschritt nur einmal aufrufen wird.

   Bei jedem Start eines Fahrzeugs auf einem neuen Weg soll nun eine Instanz von `FzgVerhalten` erzeugt und in `Fahrzeug` gespeichert werden. Dies geschieht am Besten durch eine neue Memberfunktion `Fahrzeug::vNeueStrecke(Weg*)`, die ein geeignetes Objekt erzeugt und in `p_pVerhalten` speichert.

   Beachte: `FzgVerhalten`-Instanzen, auf die es keinen Verweis mehr gibt (Speicherloch), sollen vermieden werden. Was passiert mit der alten Instanz, wenn das Fahrzeug auf einen neuen Weg gesetzt wird?



dT : Zeittakt der Abfertigung

dS(dT) : Strecke, die das Fahrzeug in dem Zeittakt dT zurücklegt
Dieser Wert wird von dStrecke() berechnet und zurückgegeben

Abbildung 3.4: Funktionsweise dStrecke()

Da es zur Zeit noch keine Einschränkungen für die Fahrzeuge gibt, soll die Funktion `dStrecke()`, wie in Abbildung 3.4 gezeigt, die aufgrund der übergebenen Zeitspanne fahrbare Strecke zurückliefern, falls dadurch die Weglänge noch nicht überschritten wird ($dT_1 \ldots dT_{n-1}$). Im Zeittakt dT_n soll nur die bis zum Wegende verbleibende Strecke zurückgegeben werden, womit das Fahrzeug genau am Ende des Weges ankommt. Im letzen Zeittakt dT_{n+1} wird dann erkannt, dass das Fahrzeug am Ende des Weges steht. Zunächst soll das Programm dann beendet werden (`exit(1)`).

Schreiben Sie nun eine Funktion `Weg::vAnnahme(Fahrzeug*)`, die ein Fahrzeug auf dem Weg annimmt. Dazu muss es in die Liste der Fahrzeuge eingetragen werden. Damit man die eingetragenen Fahrzeuge auch sehen kann, werden diese in Klammern an die Ausgabe des Weges angehängt.

Beispiel: `3 Weg1 100 ( BMW Audi BMX )`

7. Testen Sie Ihre neue Klasse, indem Sie einen Weg und zwei Fahrzeuge erzeugen, diese auf den Weg setzen und den Weg abfertigen.

8. Der Simulation sollen nun parkende Fahrzeuge hinzugefügt werden. Parkende Fahrzeuge benötigen ein anderes Verhaltensmuster, da diese sich nicht fortbewegen.

   Erweitern Sie dazu die Klasse `FzgVerhalten` zu einer Klassenhierarchie, wobei Sie zwei Klassen `FzgFahren` und `FzgParken` von `FzgVerhalten` ableiten.

   `FzgVerhalten` soll als *abstrakte Oberklasse* implementiert werden. `FzgFahren` soll das Verhalten wie vorher bei `FzgVerhalten` haben (Implementierten Sie daher für `FzgFahren` den Code nicht doppelt). Die Klasse `FzgParken` hat einen Konstruktor, der zusätzlich zum Weg den Startzeitpunkt (`double`) des Fahrzeugs übergeben bekommt. `FzgParken::dStrecke()` liefert bis zum Erreichen des Startzeitpunktes den Wert `0.0` zurück. Wenn die Startzeit erreicht wurde, soll das Programm mit einer entsprechenden Meldung beendet werden (`exit(2)`).

   Auf einem Weg sollen sich sowohl parkende als auch fahrende Fahrzeuge befinden können. Um beide zu unterscheiden, soll die Funktion `vAnnahme()` überladen werden.

Bekommt sie nur einen Zeiger auf Fahrzeug als Argument, dann nimmt sie wie bisher ein fahrendes Fahrzeug an. Wird jedoch ein Zeiger auf Fahrzeug *und* eine Startzeit (`double`) übergeben, nimmt sie ein parkendes Fahrzeug an. Alle Fahrzeuge sollen weiterhin zusammen in der vorhandenen Liste verwaltet werden.

Überladen Sie entsprechend auch die Funktion `Fahrzeug::vNeueStrecke()`.

9. Testen Sie, ob das Programm beim Starten bzw. am Streckenende wie gewünscht endet.

10. Über die Klasse `FzgVerhalten` haben Fahrzeuge und die davon abgeleiteten Klassen, Kenntnis vom befahrenen Weg. Um eine Berücksichtigung der Maximalgeschwindigkeit (`Weg::p_eLimit`), die für den befahrenen Weg gilt, zu erreichen, erweitern Sie die Methode `PKW::dGeschwindigkeit()`.

Aufgabe 5: Losfahren, Streckenende (Exception Handling)

1. Sie haben nun an zwei Stellen im Programm ein `exit()`. Anstatt das Programm mit `exit()` zu verlassen, soll nun jeweils eine Ausnahme (*Exception*) geworfen werden (`throw`), die dann in der Abfertigung des Weges aufgefangen (`catch`) und abgearbeitet werden kann. Da Sie zwei verschiedene Arten von Ausnahmen werfen, ist es vernünftig, eine Klassenhierarchie für diese Ausnahmefälle zu erstellen.

   Leiten Sie dazu zwei Klassen `Losfahren` und `Streckenende` von einer abstrakten Klasse `FahrAusnahme` ab. `FahrAusnahme` soll einen Zeiger auf `Fahrzeug` und einen Zeiger auf `Weg` als Membervariable besitzen. Diese speichern jeweils das Fahrzeug und den Weg, bei denen die Ausnahme aufgetreten ist. Implementieren Sie auch einen entsprechenden Konstruktor. Weiterhin hat die Klasse eine rein virtuelle Funktion `vBearbeiten()`. Geben Sie in den beiden Bearbeitungsmethoden der Unterklassen vorerst nur Fahrzeug, Weg und Art der Ausnahme aus.

   Lassen Sie parkende Fahrzeuge losfahren, indem Sie die entsprechende Funktion `vNeueStrecke()` aufrufen. Beim Auftreten der Ausnahmen (bisher `exit()`) sollen nun die entsprechenden Objekte geworfen und in der Abfertigungsroutine des Weges aufgefangen werden. Nachdem ein Ausnahmeobjekt gefangen wurde, wird für dieses einfach nur die Bearbeitungsfunktion `vBearbeiten()` ausgeführt.

   Beachte: Fangen Sie beide Ausnahmen mit nur *einem* `catch`-Block. Wieso ist das möglich?

2. Testen Sie in `vAufgabe_5()` die gerade implementierte Ausnahmebehandlung. Setzen Sie fahrende und parkende Fahrzeuge auf einen Weg und fertigen diesen ab. Kontrollieren Sie, ob die Ausgaben der Bearbeitungsfunktionen zu den richtigen Zeitpunkten auftreten.

   Beachte: Die Ausnahme „Streckenende“ wird beim Erreichen des Wegendes bei jeder folgenden Abfertigung erneut geworfen. Da wir noch keine Fahrzeuge von der Liste entfernen, ist dieses Verhalten kein Fehler.

3. Um die Simulation anschaulicher zu machen, soll die Abfertigung der Fahrzeuge nun grafisch dargestellt werden. Dazu wurde ein Client/Server-Modell entwickelt, bei dem der Server vom Client über TCP/IP Kommandos empfängt und diese dann in eine grafische Darstellung umsetzt.

   Der Server startet automatisch, wenn Sie die Grafikschnittstelle initialisieren (zu Hause kopieren Sie dazu die Datei SimuServer.jar in Ihr Arbeits- oder Projektverzeichnis). Falls der Server nicht startet, beachten Sie bitte die Hinweise in der zugehörigen Datei readme.txt.

   Die Grafikschnittstelle wird Ihnen durch die Dateien *SimuClient.h, SimuClient.lib* und *SimuClient.dll* zur Verfügung gestellt. Um die Grafikschnittstelle nutzen zu können, kopieren Sie zunächst *SimuClient.h* und *SimuClient.lib* in Ihr Projektverzeichnis. Diese Dateien müssen nun noch dem Projekt bekannt gemacht werden (lassen Sie sich dazu im Auswahlmenü Alle Dateien anzeigen, da *SimuClient.lib* standardmäßig nicht angezeigt wird). Einen eventuell erscheinenden Dialog zur Erstellung einer neuen Regeldatei für die lib beantworten Sie mit Nein. Kopieren Sie die Datei *SimuClient.dll* in ihr Arbeitsverzeichnis. Um die Datei *SimuClient.dll* angezeigt zu bekommen, müssen Sie vorher im Windows-Dateiexplorer unter `Anzeige` → `Optionen` den Button `Alle Dateien anzeigen` auswählen.

   Die Grafikschnittstelle stellt folgende Funktionen zur Verfügung:

   - `bool bInitialisiereGrafik( int GroesseX, int GroesseY, char* Adresse = ''localhost'' );`

     Mit dieser Funktion stellen Sie eine Verbindung zum Grafikserver her, standardmäßig der eigene Rechner (dritter Parameter entfällt). Ansonsten kann die IP-Adresse des Servers angegeben werden. `GroesseX` und `GroesseY` bestimmen die Größe der Grafikdarstellung. Verwenden Sie hier z.B. folgende Werte: `GroesseX=800`; `GroesseY=500`

   - `void vSetzeZeit( double Zeit );`

     Mit dieser Funktion können Sie die globale Zeit in der Titelzeile des Ausgabefensters angezeigen lassen.

   - `bool bZeichneStrasse( string Namehin, string NameRueck, int Laenge, int AnzahlKoord, int* Koordinaten );`

     Diese Funktion zeichnet eine Straße, die aus den beiden durch ihren Namen identifizierten Wegen besteht. Der Verlauf der Straße wird durch einen Polygonzug mit mindestens 2 Punkten (Gerade) skizziert. Die Koordinaten der Polygonpunkte werden im Array `Koordinaten` übergeben. Das Array enthält `AnzahlKoord` X/Y-Paare.

     Beachte:

     a) Verwenden Sie für die Namen nur Buchstaben, Ziffern, `"_"` und `"-"`. Es dürfen keine Leerzeichen enthalten sein.

     b) Achten Sie darauf, dass die X-/Y-Koordinatenwerte innerhalb der vorher definierten (`bInitialisiereGrafik()`) Grenzen liegen.

c) Das Array muss genau ($2 \times AnzahlKoord$) `int`-Elemente enthalten.

Für eine gerade Straße benutzen Sie für Koordinaten z.B. die Werte `{ 700, 250, 100, 250 }`.

```
bool bZeichnePKW( string PKWName, string WegName,
 double Rel_Position, double KmH, double Tank );

bool bZeichneFahrrad( string FahrradName,
 string WegName, double Rel_Position, double KmH );
```

Diese Funktionen zeichnen jeweils eine symbolische Darstellung des PKW/Fahrrads auf dem durch seinen Namen identifizierten Weg. Die relativ zur Weglänge zurückgelegte Strecke (Wert zwischen 0 und 1) wird mit `Rel_Position` angegeben. Dem Parameter `KmH` wird der Wert aus der Funktion `dGeschwindigkeit()`, dem Parameter `Tank` der aktuelle Tankinhalt übergeben.

Um beim Zeichen, abhängig vom Fahrzeugobjekt-Typ, die korrekte Zeichenfunktion aufzurufen, soll für PKW und Fahrrad eine Funktion `vZeichnen(Weg*)` implementiert werden. Dazu wird in `Fahrzeug` die Funktion virtuell deklariert und in der jeweiligen Unterklasse überschrieben. Die Funktion bekommt den Weg, auf dem das Fahrzeug gezeichnet werden soll, als Zeiger übergeben und ruft dann die passende Zeichenfunktion (s. o.) auf.

- `void vBeendeGrafik();`

  Mit dieser Funktion wird die Verbindung zum Grafikserver getrennt, das Fenster wird automatisch geschlossen.

Um die Abfertigung der PKWs und Fahrräder zu simulieren, erzeugen Sie nun zwei Wege (Länge = 500.0 km), auf die Sie jeweils ein fahrendes Fahrzeug setzen. Die Wege fassen Sie grafisch zu einer Straße zusammen (Hin- und Rückweg).

Wenn alle Funktionen integriert sind, führen Sie Ihre Simulation aus. Um die Simulation besser verfolgen zu können, rufen Sie die Funktion `vSleep(500)` in Ihre Abfertigungsschleife auf, wodurch jeweils eine Verzögerung von 500 ms erreicht wird. Je nach Rechenleistung des verwendeten Computers können Sie die Verzögerung anpassen.

Wählen Sie einen Zeittakt von 0.3 Stunden, um zu erkennen, ob die Fahrzeuge das Ende des Weges erreichen.

Aufgabe 6: Verzögertes Update (Template)

1. Wenn die Ausnahmesituationen aus der vorigen Teilaufgabe eintreten, soll nun auch die entsprechende Aktion ausgeführt werden:

- **Fahrzeug startet**: Für das später noch einzuführende Überholverbot ist es sinnvoll, die parkenden Fahrzeuge vorne, die fahrenden Fahrzeuge hinten in der Liste stehen zu haben. Benutzen Sie daher für die Aufnahme der Fahrzeuge entsprechend `push_front()` bzw. `push_back()`. Die Liste hat dann folgenden Aufbau:

  ```
  front...parkend...Wegende fahrend...Weganfang fahrend...back
  ```

  Zum Starten muss das parkende Fahrzeug aus der Liste entfernt und als fahrendes Fahrzeug sofort wieder gespeichert werden. Schreiben Sie zum Löschen der Fahrzeuge aus der Liste eine Funktion `Weg::vAbgabe(Fahrzeug*)`, die den gewünschten Zeiger aus der Liste entfernt. Mit den Funktionen `Weg::vAbgabe()` und `Weg::vAnnahme()` kann nun die Bearbeitungsfunktion von Losfahren entsprechend angepasst werden.

- **Fahrzeug kommt am Wegende an**: Passen Sie die Bearbeitungsfunktion von `Streckenende` so an, dass ankommende Fahrzeuge aus der Liste entfernt werden.

Testen Sie diese Funktionen in `vAufgabe_6()`. Wahrscheinlich kommt es zu einer Speicherschutzverletzung bei der Abfertigung des Weges, da der Iterator über die Fahrzeuge nach dem Löschen eines Fahrzeugs nicht mehr definiert ist.

2. Probleme, die auftreten können, sind Iteratoren, die auf nicht mehr existente Elemente zeigen (siehe Punkt 1) oder die Nichtabfertigung von Fahrzeugen, die in der Liste durch Umsetzen von Elementen nach hinten gerutscht sind. Um diese Probleme zu vermeiden, soll eine allgemeine Templateklasse `LazyListe` implementiert werden, die das Einfügen und Löschen von Elementen bis zum Aufruf einer Methode `LazyListe::vAktualisieren()` aufschiebt. Zur Vereinfachung geben wir Ihnen das Gerüst der Templateklassen in Form der Dateien *LazyListe.h* und *LazyAktion.h* vor. Ergänzen Sie alle Bereiche, die mit `...` gekennzeichnet sind.

   `LazyListe` besteht intern aus zwei Listen, der eigentlichen Objektliste (`list<T> p_ListeObjekte`) mit den zu speichernden Elementen vom Templatetyp `T` (hier Zeiger auf Fahrzeuge) und einer Liste zum Zwischenspeichern der noch auszuführenden Aktionen (`list<LazyAktion*> p_ListeAktionen`).

   Wir unterscheiden bei der LazyListe zwischen Lese- und Schreibfunktionen. Leseoperationen können sofort auf der eigentlichen Liste durchgeführt werden, Schreiboperationen müssen als Aktion zwischengespeichert werden.

   Für die Schreib-Aktionen wird eine Klassenhierarchie mit einer abstrakten Oberklasse `LazyAktion` angelegt, die lediglich die Funktion `vAusfuehren()` und einen Zeiger auf die zu bearbeitende Liste (`p_ListeObjekte`) beinhaltet. Für jede Schreibfunktion wird eine zugehörige Unterklasse von `LazyAktion`, also `LazyPushFront`, `LazyPushBack` und `LazyErase` abgeleitet. Dem Konstruktor der Unterklassen wird der jeweilige Parameter der Schreibfunktion und ein Zeiger auf die eigentliche Liste übergeben, da sonst kein Zugriff auf die Liste möglich wäre. Die in den Unterklassen überladene Funktion `vAusfuehren()` führt dann die eigentliche Operation aus.

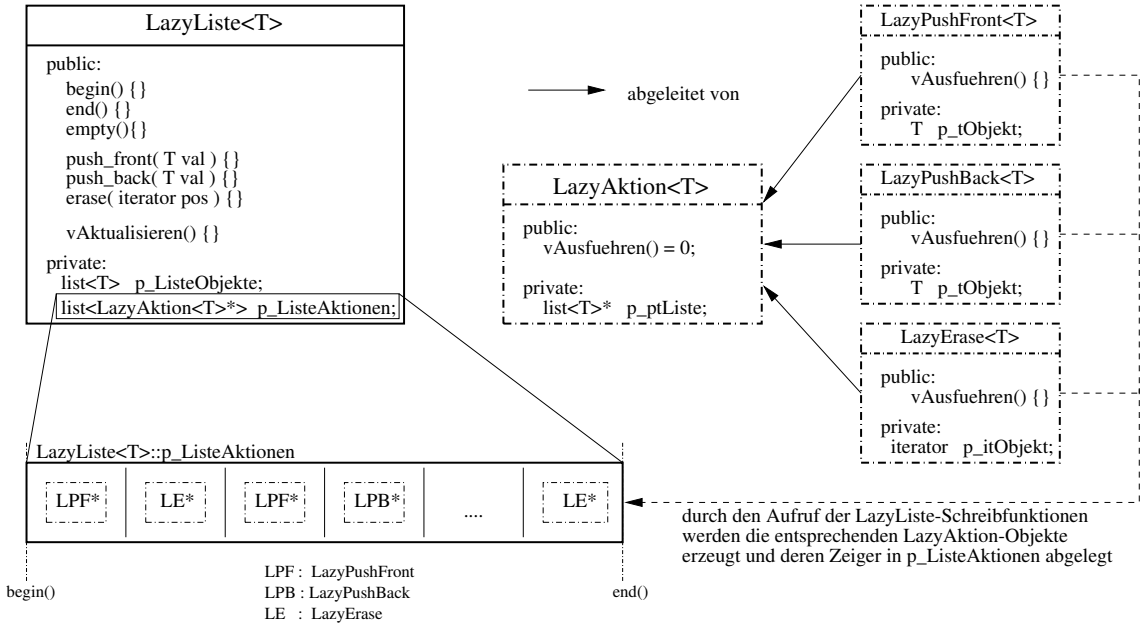


Abbildung 3.5: Prinzipielle Funktionsweise der LazyListe

Die Funktion `vAktualisieren()` der `LazyListe` durchläuft die Liste der Aktionen und führt für jedes Element der Liste die Funktion `vAusfuehren()` aus. Beachten Sie erstens, dass die benutzten Objekte nicht mehr gebraucht werden und zweitens, dass sie aus der Liste entfernt werden sollen.

Folgende Funktionen sollen für die LazyListe implementiert werden:

- `iterator begin()`: gibt einen Iterator zurück, der auf das erste Element zeigt
- `iterator end()`: gibt einen Iterator zurück, der hinter das letzte Element zeigt
- `bool empty()`: gibt `true` zurück, wenn das Objekt keine Elemente enthält
- `void push_front( T val)`: fügt `val` vor dem ersten Element ein
- `void push_back( T val)`: fügt `val` am Ende des Objektes ein
- `void erase(iterator pos)`: löscht das Element an Position `pos`
- `void vAktualisieren()`: aktualisiert `p_ListeObjekte`

Die Implementierung der `LazyListe` soll in zwei Header-Dateien erfolgen: *LazyAktion.h* für die Klasse `LazyAktion` und ihre Unterklassen und *LazyListe.h* für die Klasse `LazyListe` selbst. Man kann natürlich auch alles in einer .h-Datei implementieren oder für alle Klassen eigene .h-Dateien erstellen. Es soll aber hier exemplarisch ein einfach verschachteltes Template benutzt werden. Die Funktionsweise der LazyListe ist in Abbildung 3.5 dargestellt.

3. Testen Sie in `vAufgabe_6a()` Ihre neue Liste, indem Sie eine `LazyListe` von ganzzahligen Zufallszahlen zwischen 1 und 10 erzeugen. Folgende Aktionen sollen dann nacheinander auf der Liste ausgeführt werden:

- Liste ausgeben
- innerhalb einer Schleife alle Elemente > 5 mit `erase()` löschen
- Liste wieder ausgeben (da `vAktualisieren()` noch nicht ausgeführt wurde, sollte hier dieselbe Ausgabe erfolgen)
- `vAktualisieren()` auf die `LazyListe` anwenden
- Liste nochmal ausgeben (jetzt sollte sich die `LazyListe` geändert haben).
- Zum Schluss fügen Sie am Anfang und am Ende der Liste noch zwei beliebige Zahlen ein und geben die Liste zur Kontrolle nochmal aus.

Tipp: Eine ganzzahlige Zufallszahl zwischen `0` und `n-1` ermitteln Sie wie folgt:

```
int zahl = rand() % n;
```

Für die Funktion `rand()` benötigen Sie die Headerdatei `<stdlib.h>`. Die Funktion `rand()` liefert bei jedem Programmaufruf immer wieder dieselbe Zufallsfolge. Dies ist hilfreich, um einen „zufälligen“ Programmablauf besser nachvollziehen zu können. Um bei jedem Programmlauf eine andere Zufallsfolge zu erhalten, können Sie die Folge mit der Funktion `void srand(unsigned int seed)` initialisieren. Unterschiedliche Parameter für `seed` liefern unterschiedliche Folgen von Zufallszahlen. Diese Funktion sollte sinnvollerweise nur einmal im Programm aufgerufen werden. Eine häufig benutzte Initialisierung ist die Zeit. Die Funktion `time(0)` liefert die Anzahl Sekunden, die seit Nulldatum (bei Windowssystemen 1.1.1970 00:00 Uhr) vergangen sind. Für diese Funktion benötigen Sie die Headerdatei `<time.h>`.

Bitte benutzen Sie im Praktikum **immer dieselbe Zufallsfolge**, damit Ihre Tests reproduzierbar sind.

4. Ersetzen Sie bei der Fahrzeugliste in `Weg` die einfache Liste nun durch eine entsprechende `LazyListe` (denken Sie an das Aktualisieren!!). Testen Sie nun nochmal `vAufgabe_6()`. Die Speicherschutzverletzung sollte nun nicht mehr auftreten. Achten Sie darauf, ob die Fahrzeuge auf dem Weg richtig umgesetzt und am Ende des Weges aus der Liste gelöscht werden.

   Um in Visual Studio 2012 eine lästige Warnung bezüglich des Templates zu unterdrücken, schreiben Sie überall dort, wo Sie *LazyListe.h* einbinden, folgende Zeile vor die Headerdateien: `#pragma warning(disable:4786)`. Sollte die Warnung Nummer 4786 später auch bei anderen STL-Includes auftreten, können Sie auch dort diese Warnung ausschalten. Bitte keine anderen Warnungen ausschalten!

Aufgabenblock 3: Simulation des Verkehrssystems

3.3.8 Motivation

Bisher können nur einzelne, nicht zusammenhängende Wege erzeugt werden und die Fahrzeuge nur auf diesem Weg fahren. Im letzten Aufgabenblock soll dies nun zu einem vollständigen Verkehrsnetz zusammengefügt werden. Dazu sollen zunächst parkende Fahrzeuge zum Startzeitpunkt losfahren und beim Erreichen des Weges diesen auch verlassen. Zur Verknüpfung der Wege zu einem Verkehrsnetz werden dann Kreuzungen eingefügt. Um nicht das komplette Projekt neu übersetzen und erzeugen zu müssen, wenn man am Verkehrssystem etwas ändern oder mehr Fahrzeuge fahren lassen will, wird im letzten Schritt das Verkehrsnetz aus der Beschreibung in einer `ASCII`-Datei erzeugt.

Um sich einen Überblick zu verschaffen, lesen Sie den dritten Aufgabenblock zunächst komplett durch

3.3.9 Lernziele

- Nichttriviales Ändern und Erweitern eines bestehenden Projektes
- Erweiterung der grafischen Darstellung
- STL (map)
- Daten aus einer Datei einlesen

Aufgabe 7: Überholverbot

Aufgabe 7/8 kann man unter dem Thema „Methodenprogrammierung“ zusammenfassen. Es impliziert, dass die Algorithmen in diesem Kapitel nicht so genau vorgegeben sind wie in den bisherigen Aufgaben.

1. Fügen Sie der bereits vorhandenen Projektmappe Strassenverkehr ein neues Projekt vom *Typ Win32-Konsolenanwendung* mit dem Namen *Aufgabenblock_3* hinzu. Kopieren Sie mit dem Windows-Dateiexplorer alle Sourcen (nur *.h und *.cpp Dateien!!) aus *Aufgabenblock_2* und machen Sie diese Dateien dem neuen Projekt bekannt (s. Kapitel 3.3.2).

2. Testen Sie noch einmal die Bearbeitungsmethoden der Ausnahmeklassen: Ein Fahrzeug, das am Wegende ankommt, soll ausgegeben und aus der Liste gelöscht werden. Ein parkendes Fahrzeug soll bei Erreichen des Startzeitpunktes losfahren (Position innerhalb der Liste ändert sich!) und dabei seinen Namen, die Startzeit und den Startpunkt (Weg) ausgeben.

   Schreiben Sie dazu eine Hauptfunktion `vAufgabe_7()`, die zwei parkende Fahrzeuge (ein Fahrrad und ein PKW) auf einen Weg stellt und diesen abfertigt (der zweite Weg dient hier nur der grafischen Darstellung). Ihr Eventhandler sollte so lange abfertigen, dass beide Fahrzeuge losfahren und das Ende des Weges erreichen. Nach der Hälfte

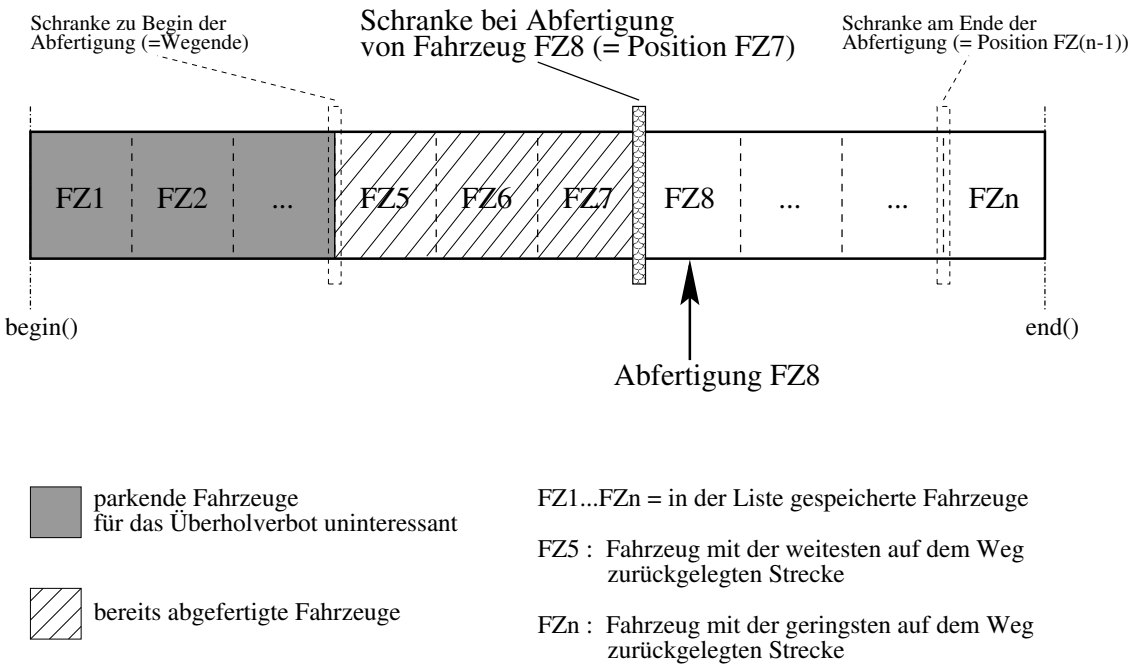


Abbildung 3.6: Schranke zum Zeitpunkt der Abfertigung von FZ8

der Zeit soll ein weiteres Fahrzeug vom Weg (parkend) angenommen werden. Die Ausgabe soll so implementiert werden, dass man die Umsortierung der Fahrzeuge in der Liste erkennt.

3. Da Überholen auf einigen, schlecht ausgebauten, Straßen unserer Verkehrswelt viel zu gefährlich ist, implementieren Sie in `FzgFahren::dStrecke()` ein Überholverbot. Das Nebeneinanderfahren (gleiche Position) kann zur Vereinfachung erlaubt werden. Das Überholverbot soll durch eine neue Boolesche Member-Variable gesteuert werden, `Weg::p_bUeberholverbot` (`true` $\widehat{=}$ schlecht ausgebauter Weg mit Überholverbot). Beim Erzeugen eines neuen Weges wird standardmäßig davon ausgegangen dass dieser schlecht ausgebaut ist.

   Tipp: Führen Sie zusätzlich zur Weglänge eine *„virtuelle Schranke“* ein, die jeweils auf die Position des aktuell abgefertigten Fahrzeugs gesetzt wird. Das nächste Fahrzeug darf dann nicht weiter als bis zu dieser Schranke fahren. Machen Sie sich klar, dass Sie wegen des speziellen Aufbaus der Liste (s. Abbildung 3.6) das Überholverbot so realisieren können. Andere Lösungen sind natürlich auch erlaubt. Ein liegengebliebener PKW (`p_dTankinhalt = 0.0`) soll kein Hindernis für nachfolgende Fahrzeuge darstellen.

Aufgabe 8: Aufbau des Verkehrssystems

1. Bisher besteht das Verkehrsnetz nur aus isolierten Wegen und darauf fahrenden Fahrzeugen. Die Wege sollen nun mittels Kreuzungen verbunden werden. Da die Infrastruktur gut ausgebaut ist, soll es keine Einbahnstraßen geben und eine Straße jeweils aus Hin- und Rückweg bestehen.

Erweitern Sie die Klassenhierarchie um die Klasse `Kreuzung`. Die Klasse `Kreuzung` speichert in einer Liste alle von ihr *wegführenden* Wege und bekommt eine Membervariable `p_dTankstelle`. Die Variable speichert das Volumen (Liter), dass einer Kreuzung zum Auftanken zur Verfügung steht. Überfährt ein PKW eine Kreuzung, wird er vollgetankt und `p_dTankstelle` um die entsprechende Menge reduziert, so oft, bis die Tankstelle leer ist (`p_dTankstelle = 0.0`). Auch hier gibt es zur Vereinfachung eine Reserve, so dass auch der letzte PKW voll tanken kann.

Schreiben Sie eine Methode `Kreuzung::vVerbinde(...)`, welche die Namen des Hin- und Rückweges, die Weglänge, einen Zeiger auf die zu verbindende Kreuzung sowie die gültige Geschwindigkeitsbegrenzung und das mögliche Überholverbot als Parameter übernimmt. Um die Kreuzungen verbinden zu können, müssen die Wege erzeugt und untereinander bekannt gemacht werden, d. h. ein Weg kennt seinen direkten Rückweg und er weiß, auf welche Kreuzung er führt. Passen Sie die Klasse `Weg` entsprechend an. Weiterhin bekommt `Kreuzung` die Funktion `vTanken(Fahrzeug*)`, die ggf. das angegebene Fahrzeug volltankt und den Inhalt der Tankstelle aktualisiert.

Implementieren Sie eine Methode `Kreuzung::vAnnahme(Fahrzeug*, double)`, die Fahrzeuge annimmt und diese parkend auf den ersten abgehenden Weg stellt. Die Fahrzeuge sollen dabei aufgetankt werden. Nun implementieren Sie noch eine Funktion `Kreuzung::vAbfertigung()`, die alle von dieser Kreuzung abgehenden Wege abfertigt.

2. Testen Sie die bisherige Klasse `Kreuzung` in `vAufgabe_8()`, indem Sie ein Verkehrsnetz entsprechend Abbildung 3.7 aufbauen und darin Fahrzeuge über die Kreuzung `Kr1` annehmen.

   Kontrollieren Sie zunächst nur den statischen Aufbau des Verkehrsnetzes, d.h. ob alle Kreuzungen die richtigen Wege gespeichert haben und ob alle Fahrzeuge auf den vorgesehenen Wegen stehen. Erweitern Sie dazu die entsprechenden Ausgabefunktionen.

   Setzen Sie dann die Tankkapazität für Kreuzung `Kr2` auf 1000 Liter und fertigen Sie die Kreuzungen ab. Alle anderen Kreuzungen haben keine Tankstelle. Passen Sie die Bearbeitungsfunktion von Streckenende so an, dass Fahrzeuge von der Zielkreuzung angenommen werden.

3. Beim Weiterleiten von Fahrzeugen sollen diese aus den wegführenden Wegen zufällig einen auswählen, aber nicht dieselbe Straße zurückfahren, die sie gekommen sind. Implementieren Sie dazu eine Funktion `Kreuzung::ptZufaelligerWeg(Weg*)`, die als Parameter einen Zeiger auf den Weg enthält, über den die Kreuzung erreicht wurde. Bei einer „Sackgasse“ muss natürlich der zurückführende Weg genommen werden.

   Bauen Sie diese Funktion nun in `Streckenende` ein, damit ein Fahrzeug, das am Ende des Weges angekommen ist, fahrend auf einen so gefundenen Weg umgesetzt wird. Auch dabei soll ggf. getankt werden. Um die Bewegungen der Fahrzeuge besser verfolgen zu können, soll beim Umsetzen folgende Ausgabe erfolgen:

   ```
   ZEIT       : [Zeitpunkt der Umsetzung]
   KREUZUNG   : [Name der Kreuzung] [Inhalt der Tankstelle]
   WECHSEL    : [Name alter Weg] -> [Name neuer Weg]
   FAHRZEUG   : [Daten des Fahrzeugs]
   ```

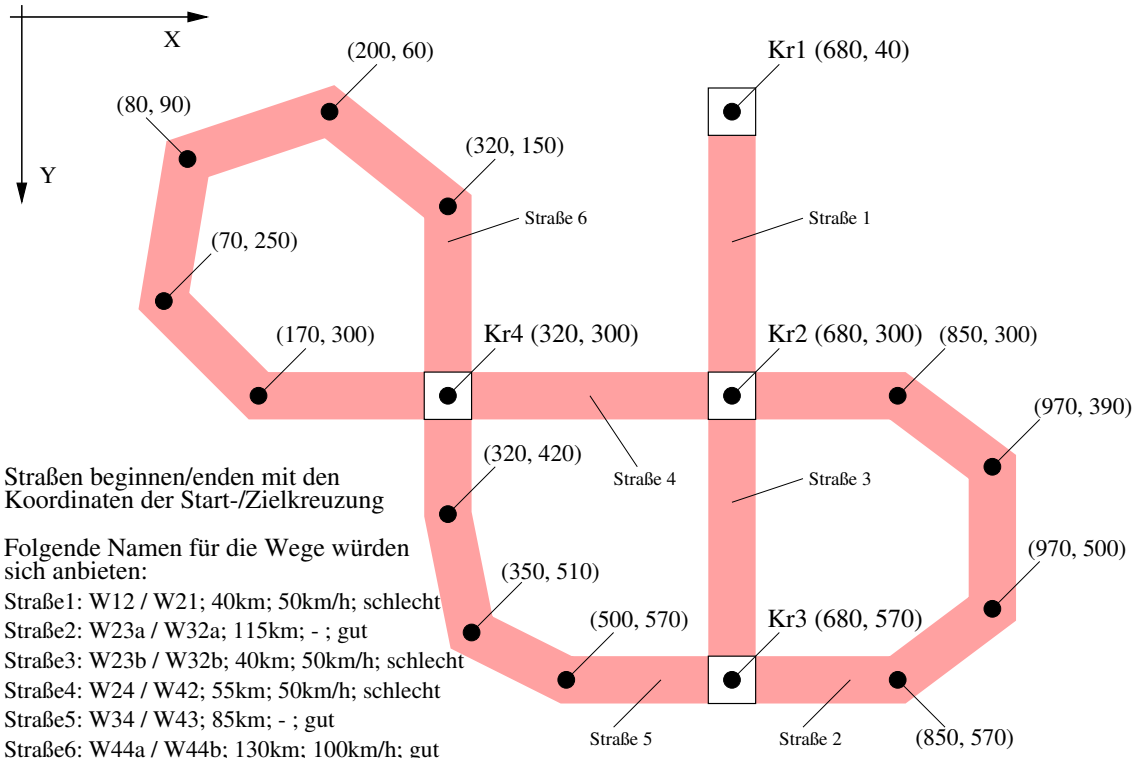


Abbildung 3.7: Koordinaten des Verkehrssystems

4. Für die grafische Darstellung der Kreuzung steht folgende Methode zur Verfügung:

   ```
   bool bZeichneKreuzung( int PositionX, int PositionY );
   ```

   Diese Funktion zeichnet eine Kreuzung an den Koordinaten `PositionX` und `PositionY`.

   Um das ganze etwas anschaulicher zu machen, soll das Abfertigen des Verkehrssystems auch grafisch dargestellt werden. Alle nötigen Grafikfunktionen sind bereits beschrieben worden. Die fehlenden Koordinaten für die Straßen und Kreuzungen finden Sie in Abbildung 3.7. Damit wird das Verkehrssystem etwas vereinfacht dargestellt.

   Für jede Straße legen Sie ein `int`-Feld mit den Koordinaten an. Dieses Feld wird dann der Funktion `bZeichneStrasse()` zum Zeichnen der Straße übergeben. Die Koordinaten der Kreuzungen werden direkt angegeben.

Aufgabe 9: Verkehrssystem als Datei (File Streams, map)

1. Überladen Sie für alle Verkehrsobjekte den Eingabeoperator `operator>>()`. Implementieren Sie, ähnlich der Lösung beim Ausgabeoperator `operator<<()`, hierfür zuerst eine Methode `istreamEingabe()`. Die einzelnen Klassen sollen Daten wie folgt einlesen:

`AktivesVO`:	[Name]
`Kreuzung`:	[AktivesVO] [Tankstelle]
`Fahrzeug`:	[AktivesVO] [MaxGeschwindigkeit]
`PKW`:	[Fahrzeug] [Verbrauch] [Tankvolumen]
`Fahrrad`:	wie `Fahrzeug`

Implementieren Sie `AktivesVO::istreamEingabe()` so, dass nur in bisher unspezifizierte Objekte (`p_sName = ""`) eingelesen werden kann und werfen Sie im Fehlerfalle einen String als Fehlermeldung. Diese Exception soll im Hauptprogramm gefangen und ausgegeben werden.

Um den neuen Operator zu testen, benutzen Sie die Datei *VO.dat*, die einen PKW, ein Fahrrad und eine Kreuzung enthält. Öffnen Sie in `vAufgabe_9()` die Datei *VO.dat* als `ifstream` (testen Sie den Erfolg des Öffnens und verlassen Sie ggf. das Programm mit einer Fehlermeldung). Erzeugen Sie drei entsprechende Objekte, lesen Sie sie aus der Datei ein und geben Sie sie am Bildschirm wieder aus.

2. Da Verkehrsobjekte teilweise automatisch erzeugt werden, ist in einigen Fällen nur der Name (`string`) bekannt. Für die Operationen werden jedoch Zeiger auf Objekte benötigt (z.B. benötigen Sie einen Zeiger auf einen Weg, um diesen abzufertigen, haben aber nur den Namen beim Verbinden der Kreuzungen angegeben).

   Speichern Sie deshalb für **alle** Verkehrsobjekte den Namen und den dazugehörigen Zeiger in einer map der STL. Überlegen Sie, an welchen Stellen im Programmcode Sie Objekte in die `map` einfügen, so dass erzeugte **und** eingelesene Objekte berücksichtigt werden. Ist ein Objekt unter dem gewünschten Namen bereits abgelegt, werfen Sie einen entsprechenden `string` als Fehlermeldung.

   Mit der Methode `AktivesVO* AktivesVO::ptObjekt(string)` soll dann über den Namen auf den entsprechenden Zeiger zugegriffen werden. Da es sich bei `AktivesVO` um eine abstrakte Klasse handelt, implementieren Sie `ptObjekt()` als statische Funktion. Sollte unter dem angegebenen Namen kein Objekt vorhanden sein, werfen Sie auch hier einen String als Fehlermeldung.

   Testen Sie Ihre `map` und kontrollieren Sie, ob alle Fehleingaben abgefangen werden.

3. Um unterschiedliche Simulationen komfortabel durchführen zu können, implementieren Sie eine Klasse `Welt`, die zwei Methoden `vEinlesen()` und `vSimulation()` anbietet. `vEinlesen()` bekommt einen Eingabestrom und erzeugt aus diesem das komplette Verkehrsnetz. Der Eingabestrom besteht aus Zeilen folgender Syntax:

   `KREUZUNG` <Kreuzungsdaten>
   `STRASSE` <Name Quellkreuzung> <Name Zielkreuzung> < Name Weg1> <Name Weg2> <Länge> <Geschwindigkeitsbegrenzung> <Überholverbot>
   `PKW` <PKW-Daten> <Name Startkreuzung> <Zeitpunkt des Losfahrens>
   `FAHRRAD` <Fahrraddaten> <Name Startkreuzung> <Zeitpunkt des Losfahrens>

   Dabei gelten folgende Wertekonventionen:

   ```
   Überholverbot: bool: 0(falsch) oder 1(wahr)
   Geschwindigkeitsbegrenzung: enum: 1(innerorts) 2(Landstraße) 3(Autobahn)
   ```

Führen Sie für jede Zeile des Eingabestroms eine entsprechende Aktion durch. Dazu erzeugen Sie ein dem Schlüsselwort (`KREUZUNG, PKW, FAHRRAD`) entsprechendes Objekt und lesen die vorgegebenen Daten für dieses Objekt ein oder Sie verbinden zwei Kreuzungen (`STRASSE`). Zum Einlesen der objektspezifischen Daten soll der überladene Eingabeoperator benutzt werden. Im Falle eines fehlerhaften Schlüsselwortes werfen Sie wieder einen String als Fehlermeldung.

Beachte:

- Stellen Sie durch entsprechendes Casting (und Fehlerauswertung) sicher, dass nur zulässige Objekte benutzt werden. Es sollen z.B. nicht versehentlich Fahrzeuge als Kreuzungen benutzt werden.
- Um eine Simulation durchführen zu können, muss Welt alle existierenden Kreuzungen kennen.

4. Die Methode `Welt::vSimulation()` fertigt alle ihr bekannten Kreuzungen ab und stellt somit einen Simulationsschritt dar.

5. Implementieren Sie das Hauptprogramm, welches eine Eingabedatei öffnet, die Welt mit diesem Eingabestrom erzeugt und eine gewisse Zeitspanne Simulationsschritte durchführt. Benutzen Sie als Eingabedatei die Datei *Simu.dat*. Die vorgegebene Datei enthält Fehler. Testen Sie, ob entsprechende Ausnahmen geworfen werden und korrigieren Sie dann die Fehler in Ihrer Kopie von *Simu.dat*.

6. Als letzte Aufgabe führen wir die Simulation mit der grafischen Darstellung zusammen. Schreiben Sie zum Erzeugen einer „grafischen Welt“ eine neue Funktion `vEinlesenMitGrafik()`. Erweitern Sie dazu eine Kopie von `vEinlesen()`, so dass alle zusätzlichen Werte pro Schlüsselwort eingelesen werden. Benutzen Sie als Eingabedatei für die Simulation *SimuDisplay.dat*.

   Ergänzen Sie die Syntax `KREUZUNG` um die beiden Koordinatenwerte und rufen Sie die Funktion `bZeichneKreuzung()` nach dem Erzeugen der Kreuzung auf.

   Ergänzen Sie die Syntax für `STRASSE` um die Anzahl der Koordinaten und das Array der X/Y-Werte und rufen Sie die Funktion `bZeichneStrasse()` nach dem Erzeugen der Straße auf.

Abbildungsverzeichnis

Index

RWTH Aachen
Lehrstuhl für Betriebssysteme

N.N.
Kopernikusstr. 16
52056 Aachen
Germany

Tel.: +(49)-241-8027634
Fax: +(49)-241-80627634

WWW: http://lfbs.rwth-aachen.de

RWTH Aachen
Lehrstuhl für Allgemeine Elektrotechnik
und Datenverarbeitungssysteme

Univ.-Prof. Dr.-Ing. Tobias G. Noll
Schinkelstr. 2
52062 Aachen
Germany

+(49)-241-8097600
+(49)-241-8092282

http://eecs.rwth-aachen.de

E-Mail: PI2Betreuung@cip1.eecs.rwth-aachen.de