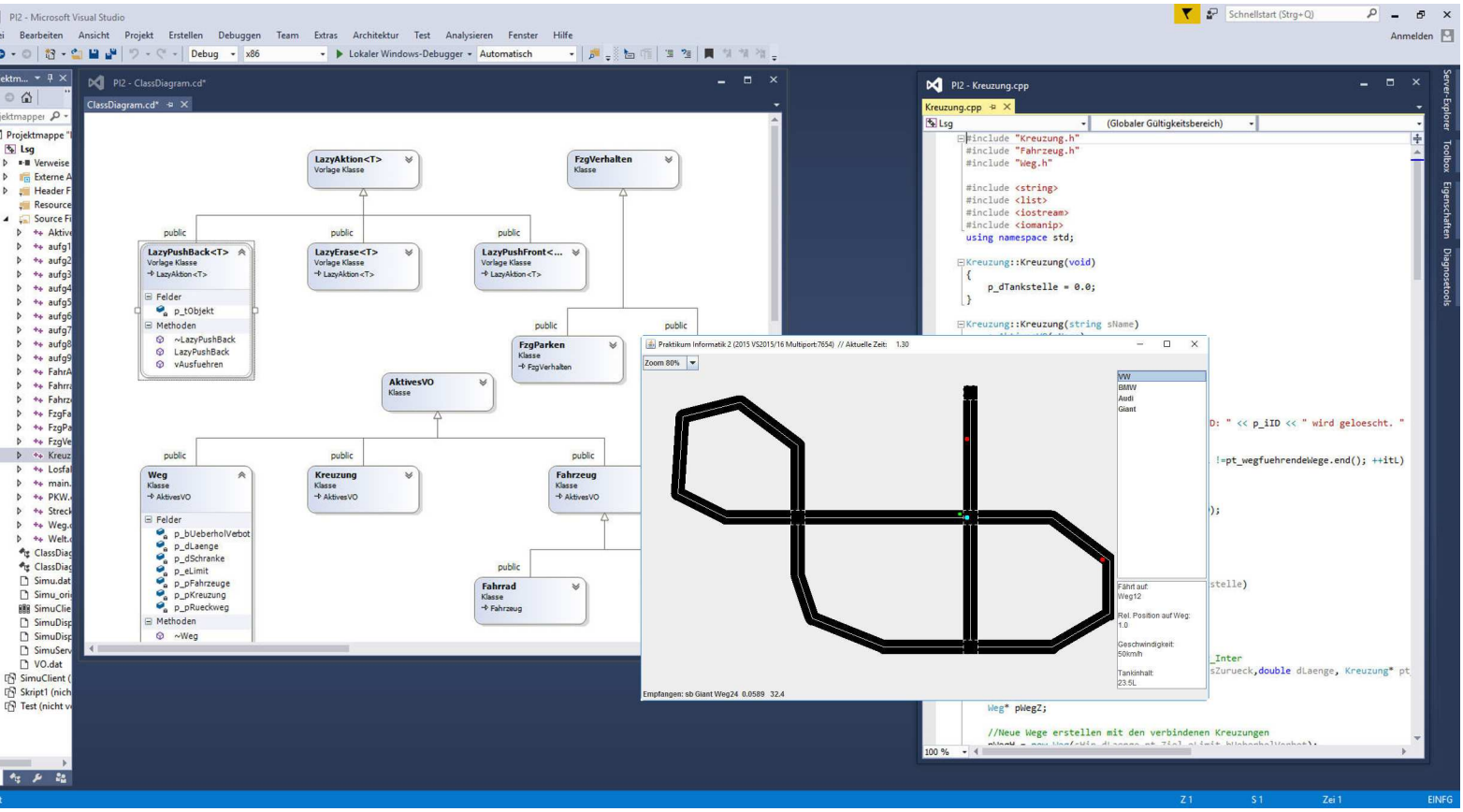
E.ON Energy Research Center
ACS | Institute for Automation of Complex Power Systems

Lehrstuhl für Integrierte digitale Systeme und Schaltungsentwurf

RWTH AACHEN UNIVERSITY

Praktikum Informatik 2
Skript und Aufgabenstellung

Wintersemester 2018/2019



Stand: 18. Oktober 2018

Inhaltsverzeichnis

L^{2}P

Zum Praktikum gehört ein Lehr- und Lernbereich (L^{2}P). Zum Zugriff darauf müssen Sie sich über RWTHOnline (zusätzlich zum Praktikum) für die Veranstaltung „Einführungsveranstaltung zum Praktikum Informatik 2“ anmelden (auch wenn Sie an dieser eventuell nicht teilgenommen haben). Die Anmeldung und der Zugriff ist bereits vor der Zuteilung zum Praktikum und bis Ende November möglich. Dort finden Sie:

1. Alle wichtigen organisatorischen Hinweise
2. Das Skript und die Aufgabenstellung als PDF
3. Die Vorgabedateien für die Bearbeitung von Aufgabenblock 2 und 3
4. Ein ausführliches Literaturverzeichnis
5. Links zu Online-Tutorien und Seiten mit grundlegenden und weiterführenden Informationen zu objektorientierter Programmierung unter C++.
6. Videos für die Basisschritte der Bedienung des Visual Studios
7. Ein Diskussionsforum mit (beantworteten) Fragen zum Praktikum aus den vergangenen Semestern. Dort haben Sie die Möglichkeit zur Problemlösung durch die Betreuer und Kommilitonen Fragen auch außerhalb der Praktikumszeiten zu stellen.

Ohne Nutzung des L^{2}P ist das Praktikum nicht sinnvoll zu bearbeiten.

Literaturverzeichnis

Im folgenden Abschnitt finden Sie einen kleinen Auszug aus der zur Verfügung stehenden Literatur zu C++ und objektorientierter Programmierung. Weitere Literaturhinweise und Links zu Online-Tutorien finden Sie im L^{2}P.

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

1. Kirch, Ulla, Prinz, Peter
   C++- Lernen und professionell anwenden
   mitp Professional 2015
   Mit Neuerungen zu C++ 11-14 (im Praktikum nicht benötigt)

2. Prinz, Peter, Kirch-Prinz Ulla
   C++- Das Übungsbuch.
   mitp Professional 2013

3. Schildt, Herbert
   C++- Die professionelle Referenz
   mitp Professional 2007

4. Stroustrup, Bjarne, Langenau, Frank (Übersetzer)
   Die C++-Programmiersprache
   Hanser 2015
   Mit Neuerungen zu C++ 11 (im Praktikum nicht benötigt)

   engl. Originalausgabe dazu:
   Stroustrup, Bjarne
   The C++ programming language
   Addison-Wesley 2013
   Vom Erfinder von C++

5. Stroustrup, Bjarne
   Principles and practice using C++
   Addison-Wesley 2014
   Mit Neuerungen zu C++ 11-14 (im Praktikum nicht benötigt)

1 Die Programmiersprache C++: Grundlagen

1.1 Lexikalische Konventionen

1.1.1 Token

Die Eingabe wird vom Compiler in sogenannte Token zerlegt, in C++ sind dies:

- Bezeichner (Identifier)
- Schlüsselworte (Keywords)
- Literale (Literals)
- Operatoren (Operators)
- Trennzeichen (Separators)

Leerzeichen, Tabulatoren, Zeilenenden und Kommentare werden ignoriert, sie trennen aber Token voneinander.

1.1.2 Kommentare

Ein Kommentar beginnt mit den Zeichen `/*` und endet mit `*/`.

```
/*
Diese Schreibweise fuer einen Kommentar ermoeglicht es den Text
ueber mehrere Zeilen zu verteilen.
*/
```

Solche Kommentare können nicht verschachtelt werden. Darüberhinaus lässt sich der Rest einer Zeile durch // als Kommentar kennzeichnen.

```
int i;     // Zähler
```

1.1.3 Bezeichner

Ein Bezeichner (engl: *identifier*) ist eine beliebig lange Folge von Buchstaben und Ziffern, wobei das erste Zeichen allerdings ein Buchstabe sein muss. Zu den Buchstaben wird auch das Underscore-Zeichen _ gerechnet. Bei Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

1.1.4 Schlüsselwörter

In C++ gibt es folgende Schlüsselwörter, die reserviert sind und nicht anderweitig benutzt werden dürfen:

```
asm     continue  float   new        signed    try
auto    default   for     operator   sizeof    typedef
break   delete    friend  private    static    union
case    do        goto    protected  struct    unsigned
catch   double    if      public     switch    virtual
char    else      inline  register   template  void
class   enum      int     return     this      volatile
const   extern    long    short      throw     while
```

Bei Schlüsselwörtern wird wie bei allen C++-Bezeichnern Groß- und Kleinschreibung unterschieden. Bei ASCII-Texten werden folgende Zeichen zur Interpunktion verwendet:

```
! % ^ & * ( ) - + = { } | ~ [ ] \ ; ' : " < > ? , . /
```

Außerdem folgende Zeichenkombinationen als Operatoren:

```
-> ++ -- .* ->* << >>  <=  >= == != &&
|| *= /= %= +=  -= <<= >>= %= ^= |= ::
```

1.1.5 Literale

Integer-Konstanten

Bei Integer (ganzzahligen)-Konstanten gibt es folgende Erscheinungsformen:

- Eine Zahl, die mit einer Ziffer ungleich 0 beginnt, ist eine Dezimalkonstante, z. B. `1234` oder `3990`.
- Eine Zahl, die mit `0` beginnt, ist eine Oktalkonstante, z. B. `015` oder `0377`.
- Eine Zahl, die mit `0x` beginnt, ist eine Hexadezimalkonstante, z. B. `0xffff` oder `0x5da7`.
- Der Typ einer Konstanten (siehe auch weiter hinten) richtet sich nach ihrem Wert und den entsprechenden Wertebereichen der Datentypen in der Implementation. Durch Angabe von `L` oder `U` hinter der Konstanten kann das aber auch explizit angegeben werden, z. B. `1999U (unsigned)` oder `0xabcdL (long)`.

Character-Konstanten

- Ein Zeichen, eingeschlossen in einfache Hochkommata ('x') ist eine Zeichenkonstante (Character-Konstante), nämlich der numerische Wert des Zeichens in der internen Codierung des Rechners (meistens `ASCII`). Beispiel: '0' hat den numerischen Wert 48 bei ASCII-Codierung.
- Folgende Sonderzeichen werden in Character-Konstanten verstanden:

`\0`	0-Byte (Stringende)
`\n`	Zeilenvorschub (Newline)
`\t`	Horizontaler Tabulator (Tab)
`\v`	Vertikaler Tabulator (Tab)
`\b`	Rückwärts löschen (Backspace)
`\r`	Zeilenende (Carriage Return)
`\f`	Seitenvorschub (Form Feed)
`\a`	Lärm (Bell)
`\?`	Fragezeichen (normalerweise einfach?)
`\'`	Hochkomma (Single Quote)
`\"`	Anführungszeichen (Double Quote)
`\`*nnn*	Oktaler Zeichencode
`\x`*hhh*	Hexadezimaler Zeichencode

Fließkomma-Konstanten

- Eine Zahl mit Vorkommastellen, dem Punkt ., Nachkommastellen und optional der Exponentenangabe mit e und Exponent ist eine Fließkommakonstante. Beispiel: `12.35` oder `1.602e-19` ($1.602 * 10^{-19}$).
- Der Typ einer solchen Konstanten ist double, durch Angabe von `F (float)` oder `L (long double)` kann der entsprechende Typ explizit angegeben werden, z. B. `0.25F`.

Bei Gleitkommazahlen bzw. Ausdrücken muss entweder die Exponentenschreibweise verwendet werden oder es muss ein Punkt enthalten sein, sonst kann es bei Ausdrücken zu unerwünschtem Verhalten kommen. So ergibt z. B. der Ausdruck 1/2 als Ergebnis `0`, da dies eine Integer-Division ist. Soll aber tatsächlich eine Fließkommadivision durchgeführt werden, so müssen die Operanden den entsprechenden Typ haben, im Beispiel führt $\frac{1.0}{2.0}$ zum gewünschten Ergebnis `0.5`.

C-String-Konstanten

- Eine Sequenz von Zeichen, eingeschlossen in Anführungszeichen, ist eine C-String-Konstante, z. B. `"abc"`.
- Der Typ einer C-String-Konstanten ist ein Array von `char`. Eine C-String-Konstante sollte nicht verändert werden.

Bei C-String-Konstanten können die gleichen Sonderzeichen mit \ eingegeben werden wie bei Character-Konstanten, z. B. `"Test.\n"`. Ein C-String wird vom Compiler durch ein abschließendes 0-Byte gekennzeichnet, `"abc"` wird im Speicher also als `'a' 'b' 'c' '\0'` abgelegt.

1.2 Grundlegende Konzepte

1.2.1 Fundamentale Datentypen

Character

`char`	„Normaler“ `char`-Typ
`signed char`	Vorzeichenbehaftet
`unsigned char`	Nicht vorzeichenbehaftet

Ob der Default-`char`-Typ vorzeichenbehaftet ist oder nicht, legt der Standard nicht fest. In sehr vielen Implementationen ist der Default aber `signed`.

Integer

`short`	Vorzeichenbehaftet
`int`	Vorzeichenbehaftet
`long`	Vorzeichenbehaftet

Alle diese Typen gibt es auch in einer `unsigned` Variante, z. B. `unsigned long`.

Fließkomma

`float`	Einfach genaue Fließkommazahl
`double`	Doppelt genaue Fließkommazahl
`long double`	Extra genaue Fließkommazahl

Der von den Datentypen belegte Speicherplatz und deren Wertebereich ist stark implementationsabhängig, für 32-Bit-Architekturen (z. B. SUN SPARC-Stations und PCs ab 386er) gelten im Regelfall die in Tabelle 1.1 angegebenen Werte.

Umwandlungen zwischen den Datentypen passieren normalerweise automatisch. Darüber hinaus existieren Cast-Operatoren, die eine explizite Typumwandlung erzwingen:

```
double x;
x = double(5);     // Passiert aber auch schon automatisch
x = (double) 5/6; // Erzwingt Fliesskommadivision
```

Typ	Größe (in Byte)	Wertebereich
Char/Integer mit Vorzeichen		
`char`	1	-128...127
`short`	2	-32768...32767
`int`	4	-2147483648...2147483647
`long`	4	-2147483648...2147483647
Char/Integer ohne Vorzeichen		
`unsigned char`	1	0...255
`unsigned short`	2	0...65535
`unsigned int`	4	0...4294967295
`unsigned long`	4	0...4294967295
Fließkomma		
`float`	4	$\approx 1.4 * 10^{-45}...3.4 * 10^{+38}$
`double`	8	$\approx 4.94 * 10^{-324}...1.8 * 10^{+308}$
Boolean		
`bool`	1	$true, false$

Tabelle 1.1: Typen in C++

Void

Der Datentyp `void` ist syntaktisch ein fundamentaler Datentyp. Er kann aber nur in Zusammenhang mit zusammengesetzten Datentypen verwendet werden, da es kein Objekt vom Typ `void` gibt. `void` wird benutzt um anzugeben, dass eine Funktion keinen Rückgabewert hat oder als Basistyp für einen Pointer auf ein Objekt mit unbekanntem Typ.

```
void f()     // Funktion liefert keinen Wert
void* p;     // Pointer auf Objekt mit unbekanntem Typ
```

Boolean

Logische Werte - sogenannte Wahrheitswerte - werden mit dem Typ bool dargestellt.

`bool` Wahrheitswert (true oder false)

```
bool bIsPos;         // Variable mit Vorbesetzung false
bool bIsVar = true;    // Variable mit Vorbesetzung true
bool bIsEmpty();     // boolesche Funktion

bIsPos = bIsEmpty();  // Zuweisung des Funktionswertes
bIsPos = (5 < 7);     // Zuweisung eines logischen Ausdrucks (hier true)
```

1.2.2 Zusammengesetzte Datentypen

C++ bietet folgende zusammengesetzte Datentypen, die aus den elementaren Typen gebildet werden können:

- Arrays (Felder) von Objekten
- Funktionen
- Pointer auf Objekte oder Funktionen
- Referenzen auf Objekte oder Funktionen
- Konstanten (siehe `const`)
- Klassen
- Strukturen
- Unions (Varianten)

Pointer (Zeiger)

Ein Pointer ist ein Datentyp, der die Adresse eines Objekts beinhaltet. Deklariert wird ein Pointer folgendermaßen:

```
TYPE* name;
```

`TYPE*` ist der Datentyp „Pointer auf `TYPE`“. `name` enthält die Adresse eines Objektes vom Typ `TYPE`. Es gibt zwei prinzipielle Operationen mit Pointern:

- *Dereferencing* (`*`)
  Zugriff auf das Objekt, auf das der Pointer zeigt.
- *Address of* (`&`)
  Referenzierung eines Objekts über einen Pointer.

Beispiel:

```
char c1 = 'A';
char* p = &c1;  // p enthält jetzt die Adresse von c1
char c2 = *p; // c2 enthält jetzt ebenfalls 'A'
```

Null-Pointer

Um zu kennzeichnen, dass ein Pointer eine ungültige Adresse hat, gibt es den speziellen Wert Null-Pointer. Der Null-Pointer wird in C++ durch das Symbol 0 angesprochen, 0 kann jedem Pointer zugewiesen werden.

```
char* p;
if (error)
  p = 0;     // Null-Pointer
else
  p = some_function();
```

Die Präprozessor-Konstante `NULL`, wie aus C bekannt, sollte aufgrund der engeren Typprüfung von C++ für diesen Zweck nicht benutzt werden.

Arrays

Arrays sind ein- oder mehrdimensionale Felder eines Datentyps, z. B. ein Array aus Integern. Felder werden über den Index-Operator `[]` indiziert.

```
double x;
double a[10]; // Ein Array mit 10 doubles, indiziert von 0..9
int k[5]; // Ein Array mit 5 ints,     indiziert von 0..4
a[0] = 0.0;   // Zuweisung an das Element
a[9] = 0.9;
k[3] = 125;
a[3] = k[3];
char matrix[10][20];  // char-Matrix aus 10 x 20 Elementen
matrix[0][0] = 'a'; // Zuweisung
matrix[0][10] = 'b';
matrix[9][0] = 'c';
```

Zu beachten: die Indizes eines mit n dimensionierten Arrays laufen immer von `0...n-1`. Die Indizierung wird nicht überprüft, so dass das Programm gegebenenfalls abstürzt. Je nach Entwicklungsumgebung kann eine Index-Überprüfung vom Debugger durchgeführt werden.

Pointer und Arrays

Der Zugriff auf ein Array kann auch über Pointer (Zeiger) erfolgen, tatsächlich ist nämlich `a[i]` nichts anderes als `*(a+i)`.

```
int* p;        // Pointer auf int
int a[10];     // Array aus 10 ints
int i;
p = &(a[3]);  // p zeigt jetzt auf das Element a[3]
i = *p;        // *p ist der Inhalt von a[3]
```

```
p++;           // p zeigt jetzt auf a[4]
*p = i;        // a[4] erhält jetzt über p den Wert von i
```

Dies kann z. B. ausgenutzt werden, um dynamisch Arrays anzulegen:

```
int n = 20;
int i;
char* s;
s = new char [n];   // Speicherbereich mit n char allokieren
for(i = 0; i < n; i++)
  s[i] = '';        // Speicherbereich initialisieren
delete [] s;        // Speicherbereich freigeben
```

Referenzen

Referenzen sind vor allem für Funktionsargumente und -rückgabewerte sowie überladene Operatoren sinnvoll. Die Syntax für Referenzen ist ähnlich der von Pointern:

```
TYPE& name;
```

Zu beachten ist allerdings, dass Referenzen immer initialisiert werden müssen:

```
int a;
int& b;      // Ungültig, da keine Initialisierung!
int& c = a; // Gültig.
```

Eine Referenz ist ein alternativer Name für ein Objekt. Über die Referenz c im Beispiel kann hier das Objekt a angesprochen werden.

Strukturen

Strukturen sind zusammengesetzte Datentypen aus verschiedenartigen Elementen. Der Gebrauch von Strukturen soll an folgenden Beispielen erläutert werden:

```
// Beispiel für eine Strukturdeklaration:
struct Geburtsdatum
{
  char* name;
  int   tag, monat, jahr;
};

// Definition eines Objektes:
Geburtsdatum mm;
mm.name  = "Max Mueller";
mm.tag   = 29;
mm.monat = 2;
mm.jahr  = 1999;
```

Strukturen sind in C++ tatsächlich Klassen mit dem einzigen Unterschied, dass in Strukturen alle Mitglieder standardmäßig public sind (siehe auch Beschreibung Klassen in 2.1).

1.2.3 Konstanten und Aufzählungen

Benutzerdefinierte Konstanten drücken aus, dass sich Werte nicht direkt ändern. Symbolische Konstanten führen zu besser wartbarem Code als direkt im Code eingesetzte Literale. Das Schlüsselwort `const` kann der Deklaration eines Objekts zugefügt werden, wobei dadurch der Typ des Elements verändert wird.

Da eine Konstante nicht verändert werden darf, muss sie bei ihrer Definition initialisiert werden. Folgendes Beispiel verdeutlicht die Verwendung von benutzerdefinierten Konstanten:

Beispiel:

```
const double pi = 3.14159265;
const int array[] = { 0, 1, 2 ,3 };
const int i;     // Fehler, keine Initialisierung
pi = 25.9;       // Fehler, da Zuweisung
```

Bei Zeigern sind zwei Objekte beteiligt, der Zeiger und das Element auf den der Zeiger zeigt. Um den Zeiger konstant zu machen muss anstatt des *-Deklarator-Operators `*const` benutzt werden. Ein `const` vor dem Basistyp oder vor dem * bezieht sich immer auf den Basistyp.

Beispiel:

```
char* const c1;    // Konstanter Zeiger auf char
char const* c2;    // Zeiger auf konstanten char
const char* c3;    // Zeiger auf konstanten char
```

Wenn möglich sollten Konstanten auch bei der Parameterübergabe in Funktionen benutzt werden, damit man sicherstellen kann, dass der übergebene Parameter in der Funktion nicht verändert werden kann.

```
void print(const int num)
{
  num += 1; // Fehler: num ist konstant
  cout << num << endl;
}
```

Aufzählungstyp

Mit `enum` wird ein neuer Datentyp (Aufzählungstyp) definiert. Dann können Variablen dieses Typs anlegt werden. Damit kann

- einer Variablen Elemente des Aufzählungstyp zugewiesen werden, über dessen Namen.

- sicherstellt werden, dass eine Variable nur bestimmte Werte annehmen soll

Beispiele:

```
enum Farbe {rot, gelb, blau}; // {rot=0, gelb=1, blau=2}
enum CentMuenzen {C01=1, C02, C05=5, C10=10, C20=20, C50=50};
```

Wenn man keine Werte angibt, beginnt die Aufzählung bei 0 und wird jeweils um 1 erhöht. Angegebene Werte müssen `int`-Konstanten sein. Fehlende Werte erhalten dann den nächsten freien Werte (z. B. `C02` erhält den Wert 2).

Variablen werden wie folgt angelegt:

```
Farbe eineFarbe=rot;
CentMuenzen eineMuenze;
```

Mit diesen Variablen kann man alle Zuweisungen, Berechnungen und Abfragen durchführen, die mit `int`-Variablen möglich sind. Leider erfolgt bei Zuweisung von int-Werten keine automatische Zulässigkeitsprüfung durch den Compiler.

Präprozessorkonstanten

Schließlich können auch in C-Manier Konstanten mit dem Präprozessor definiert werden:

```
#define PI 3.14159265
#define false 0
#define true  1
```

Die Verwendung von `const` ist jedoch vorzuziehen.

1.2.4 Deklaration und Definition

Eine Deklaration macht einen Namen dem Programm bekannt. Eine Deklaration ist gleichzeitig auch eine Definition, es sei denn,

- sie deklariert eine Funktion ohne den Funktionsrumpf
- sie enthält die `extern`-Spezifikation und keine Initialisierung oder einen Funktionsrumpf
- sie ist eine Deklaration eines `static` Mitglieds einer Klasse
- sie ist eine Deklaration eines Klassennamens
- sie ist eine `typedef`-Deklaration

Beispiele für Definitionen:

```
int a;
int f(int x) { return x + a; }
struct S { int a; int b;};
enum { up, down };
```

Beispiele für reine Deklarationen:

```
extern int a;
int f(int);
struct S;
typedef int* pInteger;
```

1.2.5 Gültigkeitsbereich

In C++ gibt es vier Arten von Gültigkeitsbereichen:

- Lokal
  Ein Name, der lokal in einem Block deklariert wird, kann nur dort und nicht außerhalb benutzt werden. Formale Parameter einer Funktion werden so behandelt, als wären sie im äußeren Block der Funktion deklariert.

- Funktion
  Labels können überall innerhalb der Funktion benutzt werden, in der sie deklariert wurden. Den Funktionsgültigkeitsbereich gibt es nur für Labels.

- File
  Ein Name, deklariert außerhalb aller Blöcke und Klassen, hat den Gültigkeitsbereich des Files und kann überall nach seiner Deklaration benutzt werden. Diese Namen werden *global* genannt.

- Klasse
  Der Name eines Klassenmitglieds ist lokal in dieser Klasse und kann nur von einer Memberfunktion der Klasse, über den Punkt-Operator, den Pfeil-Operator, den Bereichsauflösungs-Operator (auch Scope-Operator genannt) oder von einer abgeleiteten Klasse benutzt werden.

Durch Deklaration eines gleichen Namens in einem eingeschlossenen Block oder einer Klasse wird der „äußere“ Name „versteckt“. Über den Scope-Operator kann man allerdings darauf auch zugreifen.

```
int x;          // Global

void some_function()
{
  int x;     // Versteckt globale Variable
  x = 1;     // Zuweisung an lokale Variable
  ::x = 1;   // Zuw. an globale Variable über Scope-Operator
}
```

1.2.6 Speicherklassen

Es gibt drei fundamentale Speicherklassen in C++:

- Automatischer Speicher
  Hier werden lokale Variablen und Funktionsargumente abgelegt. Sie werden automatisch bei der Definition angelegt und am Ende des Gültigkeitsbereichs wieder gelöscht.
- Statischer Speicher
  Im statischen Speicher werden globale Variablen, Variablen von Namensbereichen, statische Klassenelemente und statische Variablen in Funktionen abgelegt. Derartige Objekte existieren und behalten ihren Wert während der gesamten Ausführung des Programms.
- Freispeicher
  Diese Speicherklasse wird benutzt, wenn Speicherplatz explizit mit `new` angefordert wird. Diese Art Speicher wird auch *dynamischer Speicher* oder *Heap* genannt und muss vom Programm selbst wieder explizit mit `delete` freigegeben werden.

1.2.7 Dynamische Speicherverwaltung mit `new/delete`

Gewöhnlich haben Objekte eine Lebensdauer, die durch ihren Gültigkeitsbereich bestimmt wird. Manchmal ist es jedoch sinnvoll, Objekte zu erzeugen, deren Lebensdauer vom aktuellen Gültigkeitsbereich unabhängig sind bzw. deren Art und Anzahl erst zur Ausführung des Programms bekannt ist. Dafür gibt es die Operatoren new und delete. Objekte, die durch new angelegt wurden, befinden sich im Freispeicher. Bei dynamischer Programmierung ist der Programmierer selbst für die Speicherverwaltung verantwortlich, d. h. Objekte, die mit new explizit angelegt werden müssen auch wieder explizit mit delete zerstört werden.

In C++ gibt es im Allgemeinen keine automatische Speicherbereinigung (garbage collection). Vergisst also der Programmierer, nicht mehr benötigte Objekte zu zerstören, dann wird dieser Speicher erst zum Programmende vom Betriebssystem wieder freigegeben. Diese Art der Speicherverschwendung wird auch als Speicherleck bezeichnet.

Dynamische Speicherverwaltung führt aber noch zu anderen Problemen. Man denke nur an den Fall, dass viele Referenzen auf das gleiche Objekt zeigen. Das referenzierte Objekt darf erst dann zerstört werden, wenn es keine Referenzen mehr auf dieses Objekt gibt. Wird auf ein versehentlich zerstörtes Objekt zugegriffen, dann gibt es meist die gefürchtete Speicherschutzverletzung. Es gibt jedoch Hilfsklassen, mit denen die Referenzen auf ein Objekt verwaltet werden können.

Trotz der Gefahren, die dynamisches Programmieren mit sich bringt, kann man darauf nicht verzichten. Die Notwendigkeit (speziell in C++) und Eleganz (zumindest für einen C++ Programmierer) wird hoffentlich beim Bearbeiten der Aufgaben einsichtig.

Der Unterschied zwischen dem Anlegen/Zerstören von Objekten und Feldern ist unbedingt zu beachten. Wird ein `delete` auf ein Feld angewendet oder ein `delete[]` auf ein Objekt,

bedeutet dies undefiniertes Verhalten, was meist zu einer Speicherverletzung oder, noch schlimmer, einer anderen Programmsemantik führt.

Der Aufruf von `delete` auf einen NULL-Zeiger ist jedoch unproblematisch. Daher ist es sinnvoll, Zeiger immer mit 0 zu initialisieren. Dynamische Objekte sollten möglichst immer "parallelërzeugt und zerstört werden: Entweder Neuerstellen/Zerstören innerhalb einer Funktion oder bei Klassen Zerstören von Objekten, die im Konstruktor erstellt wurden, im Destruktor.

Beispiel für das Erzeugen/Zerstören eines Objekts:

```
int* p = 0;
delete p;      // Zerstören mit NULL-Zeiger erlaubt, aber ohne Wirkung;
p = new int;   // Neuen Integerwert erstellen
delete p;      // Zerstören des Integerwertes (Wert des Zeigers unverändert)
p = 0;         // Zeiger wieder auf 0 setzen
```

Beispiel für das Erzeugen/Zerstören eines Feldes von Objekten:

```
char* p = new char [10];
delete [] p;
```

1.3 Ausdrücke / Operatoren

C++ beinhaltet eine Vielzahl von Operatoren, die in Ausdrücken Werte verändern können. Tabelle 1.2 ist komplett aus dem Buch von Bjarne Stroustrup (siehe Literaturhinweise) übernommen und enthält eine Zusammenfassung aller Operatoren in C++. Für jeden Operator ist ein gebräuchlicher Name und ein Beispiel seiner Benutzung angegeben. Die hier vorgestellten Bedeutungen treffen für eingebaute Typen zu. Zusätzlich kann man Bedeutungen für Operatoren definieren, die auf benutzerdefinierte Typen angewendet werden (siehe Operatorüberladung).

`Klassenname` ist der Name einer Klasse, ein `Element` ein Elementname, ein `Objekt` ein Ausdruck, der ein Klassenobjekt ergibt, ein `Zeiger` ein Ausdruck, der einen Zeiger ergibt, ein `Ausdruck` ein Ausdruck und ein `Lvalue` ein Ausdruck, der ein nichtkonstantes Objekt bezeichnet.

Bereichsauflösung	`Klassenname :: Element`
Bereichsauflösung	`Namensbereichs-Name :: Element`
global	`:: Name`
global	`:: qualifizierter-Name`
Elementselektion	`Objekt.Element`
Elementselektion	`Zeiger->Element`
Indizierung	`Zeiger[Ausdruck]`
Funktionsaufruf	`Ausdruck(Ausdrucksliste)`
Werterzeugung	`Typ(Ausdrucksliste)`

continued on next page

continued form previous page	
Postinkrement	`Lvalue++`
Postdekrement	`Lvalue--`
Typidentifikation	`typeid(Typ)`
Laufzeit-Typinformation	`typeid(Ausdruck)`
zur Laufzeit geprüfte Konvertierung	`dynamic_cast<Typ>(Ausdruck)`
zur Übersetzungszeit geprüfte Konv.	`static_cast<Typ>(Ausdruck)`
ungeprüfte Konvertierung	`reinterpret_cast<Typ>(Ausdruck)`
const-Konvertierung	`const_cast<Typ>(Ausdruck)`
Objektgröße	`sizeof Objekt`
Typgröße	`sizeof(Typ)`
Präinkrement	`++Lvalue`
Prädekrement	`--Lvalue`
Komplement	`~Ausdruck`
Nicht	`!Ausdruck`
einstelliges Minus	`-Ausdruck`
einstelliges Plus	`+Ausdruck`
Adresse	`&Lvalue`
Dereferenzierung	`*Ausdruck`
Erzeugung (Belegung)	`new Typ`
Erzeugung (Belegung und Init.)	`new Typ(Ausdrucksliste)`
Erzeugung (Plazierung)	`new(Ausdrucksliste) Typ`
Erzeugung(Plazierung und Init.)	`new(Ausdrucksl.) Typ(Ausdrucksl.)`
Zerstörung (Freigabe)	`delete Zeiger`
Feldzerstörung	`delete [] Zeiger`
Cast (Typkonvertierung)	`(Typ) Ausdruck`
Elementselektion	`Objekt.*Zeiger-auf-Element`
Elementselektion	`Objekt->*Zeiger-auf-Element`
Multiplikation	`Ausdruck * Ausdruck`
Division	`Ausdruck / Ausdruck`
Modulo (Rest)	`Ausdruck % Ausdruck`
Addition	`Ausdruck + Ausdruck`
Subtraktion	`Ausdruck - Ausdruck`
Linksschieben	`Ausdruck « Ausdruck`
Rechtsschieben	`Ausdruck » Ausdruck`
Kleiner als	`Ausdruck < Ausdruck`
Kleiner gleich	`Ausdruck <= Ausdruck`
Größer als	`Ausdruck > Ausdruck`
Größer gleich	`Ausdruck >= Ausdruck`
Gleich	`Ausdruck == Ausdruck`
Ungleich	`Ausdruck != Ausdruck`
Bitweises Und	`Ausdruck & Ausdruck`

continued on next page

continued form previous page	
Bitweises Exklusiv-Oder	`Ausdruck ^ Ausdruck`
Bitweises Oder	`Ausdruck | Ausdruck`
Logisches Und	`Ausdruck && Ausdruck`
Logisches Oder	`Ausdruck || Ausdruck`
Bedingte Zuweisung	`Ausdruck ? Ausdruck : Ausdruck`
Einfache Zuweisung	`Lvalue = Ausdruck`
Multiplikation und Zuweisung	`Lvalue *= Ausdruck`
Division und Zuweisung	`Lvalue /= Ausdruck`
Modulo und Zuweisung	`Lvalue %= Ausdruck`
Addition und Zuweisung	`Lvalue += Ausdruck`
Subtraktion und Zuweisung	`Lvalue -= Ausdruck`
Linksschieben und Zuweisung	`Lvalue «= Ausdruck`
Rechtsschieben und Zuweisung	`Lvalue »= Ausdruck`
Und und Zuweisung	`Lvalue &= Ausdruck`
Oder und Zuweisung	`Lvalue |= Ausdruck`
Exklusiv-Oder und Zuweisung	`Lvalue ^= Ausdruck`
Ausnahme werfen	`throw Ausdruck`
Komma (Sequenzoperator)	`Ausdruck, Ausdruck`

Tabelle 1.2: Zusammenfassung der Operatoren

Hinweise

- Jeder Kasten enthält Operatoren gleicher Priorität. Die Operatoren in den oberen Kästen haben höhere Priorität als die in den unteren. Beispielsweise bedeutet `a+b*c` das gleiche wie `a+(b*c)`, da `*` eine höhere Priorität hat als `+`.
- Einstellige Operatoren und Zuweisungsoperatoren sind rechts-bindend, alle anderen links-bindend. Beispielsweise bedeutet `a=b=c` das gleiche wie `a=(b=c)` und `*p++` bedeutet `*(p++)`.
- Die Ergebnistypen von arithmetischen Operatoren werden nach einer Menge von Regeln bestimmt, die als die „üblichen arithmetischen Konvertierungen“ bekannt sind. Das generelle Ziel ist es, ein Resultat des „größten“ Operandentyps zu erzeugen. Beispielsweise ergibt `double+int` einen `double`-Wert oder `int*long` ergibt einen `long`-Wert.
- Die Reihenfolge der Auswertung von Teilausdrücken innerhalb eines Ausdrucks ist undefiniert. Speziell kann man nicht erwarten, dass ein Ausdruck von links nach rechts ausgewertet wird. Beispielsweise ist es undefiniert ob bei dem Code

  ```
  int x = f(2) + g(3);
  ```

  zuerst `f()` oder zuerst `g()` aufgerufen wird. Der Grund dafür ist, dass manche Compiler dadurch besser optimieren können.

1.4 Anweisungen

Es folgt eine Zusammenfassung aller Anweisungen, die es in C++ gibt. Sie ist aus dem Buch von Bjarne Stroustrup (siehe Literaturhinweise) übernommen.

```
Anweisung:
  Deklaration
  { Anweisungsliste_opt }
  try { Anweisungsliste_opt } Handler-Liste
  Ausdruck_opt;

  if(Bedingung) Anweisung
  if(Bedingung) Anweisung else Anweisung
  switch(Bedingung) Anweisung

  while(Bedingung) Anweisung
  do Anweisung while(Bedingung);
  for(for-Init-Anweisung; Bedingung_opt;Ausdruck_opt) Anweisung

  case Konstanter-Ausdruck: Anweisung
  default: Anweisung

  break;
  continue;
  return Ausdruck_opt

  goto Bezeichner;
  Bezeichner: Anweisung

Anweisungsliste:
  Anweisung Anweisungsliste_opt

Bedingung:
  Ausdruck
  Typangabe Deklarator=Ausdruck

Handler-Liste:
  catch(Ausnahme-Deklaration) { Anweisungsliste_opt }
  Handler-Liste Handler-Liste_opt
```

Man beachte, dass eine Deklaration eine Anweisung ist und dass Zuweisungen und Funktionsaufrufe Ausdrücke sind. Die Anweisung zur Handhabung von Ausnahmen, `try`-Blöcke, sind im entsprechenden Kapitel zur Ausnahmebehandlung beschrieben. Alle anderen Anweisungen werden in den folgenden Kapiteln beschrieben.

1.4.1 Deklarationen als Anweisungen

Eine Deklaration ist eine Anweisung. Der Grund, Deklarationen überall da zu erlauben, wo auch eine Anweisung erlaubt ist, liegt darin, es dem Programmierer zu ermöglichen, Fehler durch nicht initialisierte Variablen zu vermeiden und höhere Lokalität im Code

zu erzielen. Konstanten werden dadurch erst möglich und schließlich ist es oft effizienter, die Definition zu verzögern, bis ein passender Initialisierer verfügbar ist (vor allem bei benutzerdefinierten Typen).

1.4.2 `if-else`-Anweisung

Bedingte Anweisung:

`if (` *expression* `)` *statement*

bzw. mit `else`-Teil:

`if (` *expression* `)` *statement*$_1$ `else` *statement*$_2$

Bei dieser Anweisung wird zunächst die Bedingung (*expression*) bewertet. Ist das Ergebnis von Null verschieden, so wird *statement*$_1$ ausgeführt, ansonsten wird in der `if-else`-Variante *statement*$_2$ ausgeführt.

Beispiel:

```
int a = 1;
int b = 2;
int z = 0;
if (a < b) {     // Bedingung in runden Klammern
  z = b;         // Geschweifte Klammern bei nur einer
}                // Anweisung eigentlich nicht notwendig
else
  z = a;         // else-Teil kann auch entfallen

if (a < b)
  z = 1;       // a < b  => z = 1
else if (a == b)  // Alternative Bedingung
  z = 0;       // a = b  => z = 0
else
  z = -1;      // a > b  => z = -1
```

1.4.3 `switch`-Anweisung

Bedingte Anweisung mit Ausführung abhängig vom Wert eines Ausdrucks.

```
switch ( expression )
{
      case  const-expr1:      statements
      case  const-expr2:      statements
      default:                statements
}
```

Für *expression* sind nur int- oder char-Ausdrücke zulässig. Normalerweise wird eine `switch`-Anweisung aus einem Block und mehreren Labels und Anweisungen bestehen. Wichtig ist, dass es sich um einen konstanten Ausdruck (*const-expr*) hinter dem Schlüsselwort `case` handelt (keine Verwendung von Variablen usw.).

Beispiel:

```
char c;

switch (c)
{
  // x wird durch * ersetzt
  case 'x':
  c = '*';
  break;
  // Vokale werden durch ? ersetzt
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    c = '?';
    break;
  default:  // Alle anderen werden durch Leerzeichen ersetzt
    c = ' ';
    break;
}
```

Beachten Sie, dass es sich bei `case` um eine Einsprungmarke handelt, von der aus der Programmablauf fortgesetzt wird. Insbesondere werden auch die Anweisungen der nachfolgenden `case`-Labels ausgeführt, solange bis die Ausführung der `switch`-Struktur durch ein `break`-Statement abgebrochen wird (sog. *fall-through*). Im Regelfall wird daher jeder Block hinter einem `case`-Label durch ein `break`-Statement abgeschlossen.

1.4.4 `while`-Anweisung

Bei dieser Schleifenstruktur wird zunächst der Ausdruck bewertet. Ist dieser wahr (ungleich Null), so wird die abhängige Anweisung ausgeführt. Nach der Ausführung wird der Ausdruck erneut bewertet. Dieser Vorgang wird so lange durchgeführt, bis die Bewertung des Ausdrucks falsch (gleich Null) ist. Dann wird mit der Anweisung, die hinter *statement* steht fortgefahren.

`while (` *expression* `)` *statement*

1.4.5 `for`-Anweisung

Die `for`-Schleife ist wie die `while`-Schleife kopfgesteuert, d. h. die Bedingung (hier: $expression_2$) wird vor dem ersten Ausführen der bedingten Anweisung geprüft. Zusätzlich enthält die `for`-Schleife einen Initialisierungsausdruck ($expression_1$), der vor der Schleife einmalig vor

Beginn ausgeführt wird. Weiterhin einen Iterations-Ausdruck (*expression*$_3$), der zusätzlich am Ende jeder Ausführung der bedingten Anweisung durchgeführt wird.

`for (` *expression*$_1$*;* *expression*$_2$*;* *expression*$_3$ `)` *statement*

Eine `for`-Schleife entspricht also im Prinzip einer `while`-Schleife:

```
// Diese for-Schleife ist identisch ...
for (a; b; c)
  d;
// ... mit einer solchen while-Schleife
a;
while (b)
{
  d;
  c;
}
```

Beispiel für die Anwendung der for-Schleife:

```
double x[100];
int i;
for (i = 0; i < 100; i++)
  x[i] = 0.0; // Nullsetzen eines Feldes

int k = 5;
int fak = 1;
for (i = k; i > 1 ; i--)
    fak *= i;   // fak = Fakultät von k
```

1.4.6 `do-while`-Anweisung

Im Gegensatz zu den bisher genannten Schleifenkonstruktionen ist die `do-while`-Schleife fußgesteuert. Die bedingte Anweisung wird mindestens einmal beim Start der Schleife ausgeführt. Nach dieser Ausführung wird der Ausdruck *expression* ausgewertet. Ist das Ergebnis ungleich Null, so wird die bedingte Anweisung erneut durchgeführt.

`do` *statement* `while (` *expression* `);`

Man beachte das Semikolon am Ende der Konstruktion.

1.4.7 Sprunganweisungen

Wie bereits erwähnt, existieren in C++ mehrere Sprunganweisungen (engl: *jump-statements*), die vor allem bei den Schleifen und der `switch`-Anweisung zum Einsatz kommen. Durch den Einsatz der Sprunganweisungen wechselt der Programmablauf unbedingt.

- `break;`

  Dieser Sprung darf nur innerhalb einer Schleifenstruktur oder einer `switch`-Aweisung benutzt werden. In beiden Fällen wird die Struktur verlassen und mit der folgenden Anweisung fortgefahren. Sind mehrere solcher Strukturen ineinander verschachtelt, so bezieht sich die Sprunganweisung nur auf die kleinste umgebende Struktur.

- `continue;`

  `continue` darf nur in Schleifenstrukturen verwendet werden. In allen `while`-Schleifen wird die bedingte Anweisung verlassen und mit dem Ausdruck, der über Durchführung der Schleife entscheidet, fortgefahren. Wird die Sprunganweisung in einer `for`-Schleife verwendet, so wird ein Sprung zum Iterations-Ausdruck durchgeführt.

- `return` *opt-expression*`;`

  Die return-Anweisung wird benutzt, um die Ausführung einer Funktion zu beenden und zum Aufrufer zurückzukehren. Der optionale Ausdruck *opt-expression* wird dem Aufrufer zurückgeliefert. Erreicht der Programmablauf das Ende einer Funktion, so ist dies äquivalent zu einer `return`-Anweisung.

- `goto Bezeichner;`

  ```
  ...

  Bezeichner: Anweisung
  ```

  Sprung zu einer Anweisung, die durch eine Marke gekennzeichnet ist. Die Verwendung dieser unkontrollierten Sprünge sollte grundsätzlich vermieden werden. Wer also ein `goto` benutzt, sollte darüber mehrmals nachdenken. Im Praktikum soll dieser Sprungbefehl nicht verwendet werden.

1.4.8 `extern`-Anweisung

Die `extern`-Anweisung wird verwendet um eine bereits global definierte Variable in einer anderen Datei bekannt zu machen. Dies ist dann nur eine weitere Deklaration (wobei der Typ exakt übereinstimmen muss), und nicht eine neue Definition dieser Variablen.

```
Datei1.cpp:
  int x;
  double y = 10.0;

Datei2.cpp:
  double z;
  extern double y;

  z = ++y;     // z hat den Wert 11.
```

1.5 Funktionen

Eine Funktion ist eine Zusammenfassung von Anweisungen zu einer Einheit. Sie erhält Objekte eines bestimmten Typs als Argumente und liefert wiederum als Ergebnis ein Objekt eines bestimmten Typs.

Eine Funktionsdefinition in C++ spezifiziert den Namen der Funktion, den Typ des zurückgelieferten Objektes und die Typen und Namen der Argumente. Das Zurückliefern eines Wertes aus einer Funktion geschieht mit Hilfe der `return`-Anweisung. Die Funktion `main()` hat eine spezielle Bedeutung, sie wird beim Start des Programms aufgerufen.

```
long fak(int k)    // Returnwert long
{                  // Ein Argument int k
  ...
  return erg;      // Ergebnis
}

int main()         // Hauptprogramm
{
  long x;
  x = fak(5);      // Aufruf der Funktion
  return 0;        // Ende des Programms
}
```

Funktionsparameter werden in C++ normalerweise als Wert übergeben (*call by value*). Soll eine Referenz übergeben werden (*call by reference*), so muss dazu der Übergabe-Parameter als Pointer oder Referenz deklariert werden. Ausnahme bilden lediglich Arrays, die immer per *call by reference* übergeben werden.

Bei *call by value* wird eine Kopie des Werts an die Funktion übergeben. Dies hat zur Folge, dass Änderungen an den Parameter-Variablen nach dem Rücksprung aus der Funktion wirkungslos sind.

```
void test(int x, int y)
{
  x = 10;
  y = 20;      // x ist hier 10, y 20
}

int main()     // Hauptprogramm
{
  int wert1 = 15;
  int wert2 = 25;
  test(wert1, wert2); // Aufruf der Funktion
  // wert1, wert2 sind jetzt immer noch 15, 25
  return 0;
}
```

Möchte man Variablen per *call by reference* übergeben, so ist dies über Pointer möglich, aber das ist relativ umständlich:

```
void some_function(int* p) {
  *p = 99;
}

int main()
{
  int a;
  some_function(&a);
  // a hat jetzt den in some_function()
  // zugewiesenen Wert
}
```

In C++ kann dies einfacher über Referenzen realisiert werden. Das Beispiel mit der Funktion ließe sich mit Referenzen folgendermaßen umschreiben:

```
void some_function(int& i) {
  i = 99;
}

int main()
{
  int a;
  some_function(a);
  // a hat jetzt den in some_function()
  // zugewiesenen Wert
}
```

In C++ ist es außerdem möglich, mehrere Funktionen mit gleichlautendem Funktionsnamen zu versehen. Dieses „Überladen“ von Funktionsnamen unterscheidet der Compiler beim Aufruf und der Definition der Funktionen durch Anzahl, Typ und Reihenfolge ihrer Parameter.

```
double pos(double x) {       // Funktion für "double"-Argumente
  return (x < 0) ? -x : x;   // liefert Absolutbetrag
}

int pos(int x) {           // Funktion für "int"-Argumente
  return (x < 0) ? 0 : x;    // liefert für negative Werte 0
}

int main()
{
  pos(-1);     // pos(int)-Funktion wird aufgerufen(liefert 0)
  pos(-1.0);   // pos(double)-Funktion wird aufgerufen
               // (liefert 1.0)
}
```

Bei der Deklaration von Funktionen kann eine weitere Flexibilität durch *Defaultparameter* erreicht werden. Dabei wird einem Funktionsparameter standardmäßig ein Wert zugeordnet. Nur wenn die Funktion einen abweichenden Wert erhalten soll, muss dieser explizit angegeben werden.

```
void test(int x, int y=1);  // Deklaration: Standardwert y=1

int main()
{
  test(5);  // x=5; y=1
  test(7,4);  // x=7; y=4
}

void test(int x, int y) // Definition der Funktion
{
...
}
```

Die Definition der Defaultparameter erfolgt bei der Deklaration. Bei der Definition wird der Standardwert nicht mehr angegeben. Beachten Sie ggf. auftretende Mehrdeutigkeiten beim Überladen von Funktionen mit Defaultparametern. Defaultparameter können nur für hinten stehende Parameter benutzt werden.

Beispiele für unterschiedliche Funktionen bzgl. des Überladens:

```
double pos(double x, int y);

double pos(double x, double y);      // y unterschiedlicher Typ
double pos(double x, const int y);   // const unterscheidet
const double pos(double x, int y);   // const unterscheidet;
double pos(double x);                // unterschiedliche Anzahl
double pos(int x, double y);    // unterschiedliche Reihenfolge

double pos(double y, int x);         // gleiche Funktion !!!
double pos(double x, int y=1);       // gleiche Funktion !!!
                    // auch gleich zu double pos(double x);
```

1.6 Präprozessorkonzept

Vorteilhaft für die Programmentwicklung wirkt sich das Präprozessorkonzept der Sprache C/C++ aus. Zeilen, die im Sourcecode mit # eingeleitet werden, heißen Kontrollzeilen. Sie dienen der Kommunikation mit dem Präprozessor. Eine Kontrollzeile hat Auswirkungen auf die nach ihr stehenden Zeilen bis zum Ende der Datei. Sie werden üblicherweise am Anfang eines Sourcecodes eingefügt.

Eine Kontrollzeile der Form `#include <Dateiname>` bewirkt, dass der Präprozessor die Kontrollzeile durch eine Kopie des Inhalts der spezifizierten Datei ersetzt. Üblicherweise werden damit Headerdateien eingebunden. Gesucht wird in den voreingestellten Include-Verzeichnissen. Eine Kontrollzeile der Form `#include "Dateiname"` führt zu einer ersten Suche im aktuellen Verzeichnis.

Mit der Direktive `#define` können symbolische Konstanten und funktionsähnliche Makros definiert werden. Es gilt folgende Syntax:

```
#define <Makroname> <Ersetzungtext>
```

Der Präprozessor ersetzt vor der Compilierung jedes Auftreten von `<Makroname>` im Sourcecode durch den `<Ersetzungstext>`. Zur besseren Unterscheidung von Variablen und Funktionen sollten für den Makronamen nur Großbuchstaben verwendet werden. Einem Funktionsmakro können (ebenso wie einer Funktion) Parameter übergeben werden. Der Vorteil von Funktionsmakros besteht darin, dass keine Annahmen über den Typ der Parameter gemacht werden. Man kann also, im Gegensatz zu Funktionen, ein und dieselbe Definition für alle Typen verwenden.

Beispiel:

```
#include <iostream>
#include <iomanip>
/* Definition einer symbolischen Konstanten */
#define MFG "Mit freundlichen Gruessen"

/* Definition eines Funktionsmakros */
#define MAX(a,b) ((a > b) ? a : b)

void main()
{
  cout.setf(ios::fixed);
  cout.precision(2);

  /* Verwendung symbolischer Konstanten */
  cout << MFG << endl;

  /* Verwendung von Funktionsmakros */
  int xi = 10;
  int yi = 11;
  cout << "Maximum von " << xi << " und " << yi << " : "
       << MAX(xi, yi)) << endl;

  double xd = 10.0;
  double yd = 11.0;
  cout << "Maximum von " << xd << " und " << yd << " : "
       << MAX(xd, yd)) << endl;
}
```

Mit der Direktive `#undef` werden Makrodefinitionen wieder aufgehoben. Mit den bedingten Direktiven `#ifdef` und `#ifndef` lässt sich prüfen, ob ein Makro gegenwärtig definiert ist oder nicht. Hilfreich war dies z. B. in C zur Verhinderung der Mehrfacheinbindung von Headerdateien. In C++ kann dies einfacher durch die Direktive `#pragma once` erreicht werden.

Die Direktiven

```
#ifndef __MYFILE_H
  #include <myfile.h>
#endif
```

verhindern z. B. die Mehrfacheinbindung der Headerdatei myfile.h, da in dieser Headerdatei das Makro

`__MYFILE_H`

definiert wird:

```
#ifndef __MYFILE_H
  #define __MYFILE_H
  /* Deklarationen der Headerdatei */
#endif
```

Zusätzlich unterstützen die Direktiven wie `#if`, `#elif`, `#else` und `#endif` die bedingte Compilierung des Codes. Überprüft werden dabei Konstantenausdrücke (0 entspricht `FALSE` und alles andere entspricht `TRUE`).

2 Abstraktionsmechanismen in C++

Ein Problem bei der Entwicklung großer Programme besteht darin, dass die inneren Abhängigkeiten der Variablen im Programm nicht mehr überschaubar sind. In C versucht man dieses Problem durch Strukturieren der Programme in Module, die nur über ihre eindeutig definierten Schnittstellen kommunizieren, in den Griff zu bekommen.

C++ erweitert diese Möglichkeiten von C durch die Konzepte der objektorientierten Programmierung (OOP). Grundelement der objektorientierten Programmierung ist dabei die Zusammenfassung ggf. komplex strukturierter Daten und ihrer zugehörigen Methoden (Elementfunktionen) zu Klassen. Dabei kann dann gewöhnlich nur über Methoden auf die internen Datenstrukturen zugegriffen werden.

2.1 Klassen

Im Unterschied zu C können in C++ Datenstrukturen zusätzlich Elementfunktionen (sog. *Methoden*) zugeordnet werden. Diese dienen der Manipulation der in der Datenstruktur enthaltenen Variablen (sog. *data member*). Solche Datenstrukturen mit den zugehörigen Methoden nennt man Klassen. Statt des aus C bekannten Schlüsselworts `struct` für Datenstrukturen werden Klassen mit dem neuen Schlüsselwort `class` eingeleitet.

Die Verwendung von `struct` in C++ ist zwar abwärtskompatibel zu C, sollte aber in C++ nicht weiter benutzt werden. Die Erweiterung von C++ besteht darin, zusätzlich zu den Datenelementen noch Methoden zu definieren und den Zugriff genauer zu kontrollieren. Für die Zugriffskontrolle gibt es die Zugriffsspezifizierer `public` und `private`.

Beispiel:

```
class Beispiel
{
  public:
    int i;
  private:
    int j;
  public:
    void reset();
};
void Beispiel::reset()
{
  j = 0;
}
```

```
int main()
{
  Beispiel Objekt;
  Objekt.j=1;   // Fehler, da j private
  Objekt.reset(); // Zugriff auf j nur über Funktion möglich
}
```

Während auf öffentliche (`public`) Datenelemente von jeder Stelle des Programms aus zugegriffen werden darf (im Beispiel i), dürfen auf die privaten Datenelemente nur die Klassenmethoden zugreifen (im Beispiel j). Zugriffsspezifizierer können in beliebiger Reihenfolge und auch mehrfach innerhalb der Klasse verwendet werden und sind bis zur Angabe des nächsten Zugriffsspezifizierers gültig.

Eine Vereinbarung einer Klasse mit dem Schlüsselwort `class` unterscheidet sich von einer mit `struct` nur darin, dass bei `class` deren Elemente bei Fehlen eines Zugriffsspezifizierers `private` sind, bei `struct` aber `public`. Damit sind folgende Codesequenzen äquivalent:

```
struct <Klassenname>
{
  private:
  /* ... */
};
```

und:

```
class <Klassenname>
{
  /* ... */
};
```

auf der einen, sowie:

```
class <Klassenname>
{
  public:
  /* ... */
};
```

und:

```
struct <Klassenname>
{
  /* ... */
};
```

auf der anderen Seite.

In einer Klassendefinition sollten i. A. die Variablen `private` und die Methoden `public` definiert sein. Die Klassennamen sollten zur Unterscheidung von Variablennamen mit einem Großbuchstaben beginnen.

Die Erzeugung einer Variablen eines Klassentyps wird als *Instanziierung eines Objekts einer Klasse* bezeichnet. Ein Objekt (eine Instanz) ist also nichts anderes als eine Variable eines Klassentyps. Die Instanziierung von Objekten einer Klasse erfolgt analog zur Deklaration von Variablen der Standarddatentypen wie `int` oder `float` (vgl. Beispielprogramm). Zugriff auf die Mitglieder (Data Member und Methoden) eines Objektes erhält man über den *Punktoperator* (engl. *dot operator*) bei statisch und den *Pfeiloperator* (engl. *arrow operator*) bei dynamisch erzeugten Objekten. Über Getter- und Setter-Methoden wird festgelegt, auf welche Variablen lesend/schreibend zugegriffen werden kann.

Beispiel:

```
class Sum
{
public:
  int getJ(){return p_j;};
  void setI(int i);
  int getI(){return p_i;};
private:
  int p_j; // für j gibt es nur Getter
  int p_i; // für i gibt es Setter und Getter
  int p_s; // für s gibt es weder Setter noch Getter
};

void Sum::setI(int i)
{
  if (i > 0) // Abfrage zu Wertebereich möglich
  {
    p_i = i; // Instanzvariable setzen
    p_s = p_i + p_j; // ggf. abhängige Variable setzen
  }
}

int main()
{
  Sum tS;          // Statische Erzeugung eines Objektes vom Typ Sum
  tS.setI(1);      // Zugriff auf p_i nur über Funktion möglich.
  Sum* pS;
  pS = new Sum(); // Dynamische Erzeugung eines Objektes vom Typ Sum
  pS->setI(2);     // Zugriff auf p_i nur über Funktion möglich.
  int i;
  i = tS.getI();  // Zugriff über Punktoperator
  i = pS->getI(); // Zugriff über Pfeiloperator
}
```

Konstruktoren und Destruktoren

Methoden, die automatisch bei der Instanziierung eines neuen Objekts zu einer Klasse aufgerufen werden, heißen *Konstruktoren*. Diese sorgen dafür, dass bei der Instanziierung eines Objekts alle Datenelemente initialisiert werden. Im Sourcecode erkennt man die Konstruktoren daran, dass sie denselben Namen wie die Klasse tragen.

Methoden, die automatisch beim Löschen von Objekten einer Klasse aufgerufen werden, heißen *Destruktoren*. Sie geben den vom Objekt belegten Speicher frei. Sie tragen ebenfalls den gleichen Namen wie die Klasse, sind jedoch zusätzlich durch das Tildezeichen ~ gekennzeichnet.

Konstruktoren und Destruktoren haben keinen Rückgabetyp. Ein Destruktor hat auch keine Parameter. Einen Konstruktor ohne Parameter nennt man *Standardkonstruktor*. Er sollte zu jeder Klasse definiert sein. Auch der Destruktor sollte zu jeder Klasse definiert sein. Um ein korrektes Löschen von Unterklassen zu ermöglichen, sollte er immer `virtual` sein (genaues dazu s.u.).

Beispiel:

```
// Klassenbeispiel mit Konstruktor und Destruktor
#include <iostream>
class Sum
{
  public:
    Sum();              // Standardkonstruktor
    Sum(int i, int j); // Konstruktor mit Parameter
    virtual ~Sum();     // Destruktor
    int getJ(){return p_j;};
    void setI(int i);
    int getI(){return p_i;};
    int getS(){return *p_pS;};
  private:
    int p_j;
    int p_i;
    int* p_pS;
};
Sum::Sum()
{
  p_i = 0;
  p_j = 0;
  p_pS = new int;
  *p_pS = 0;
  std::cout << "Standardkonstruktor" << std::endl;
}
Sum::Sum(int i, int j)
{
  p_i = 0;
  p_j = 0;
  if (i > 0) // Abfrage zu Wertebereich möglich
  {
    p_i = i; // Instanzvariable setzen
  }
  if (j > 0) // Abfrage zu Wertebereich möglich
  {
    p_j = j; // Instanzvariable setzen
  }
  p_pS = new int;
  *p_pS = p_i + p_j; // ggf. abhängige Variable setzen
  std::cout << "Nicht-Standardkonstruktor" << std::endl;
```

```
}
Sum::~Sum()
{
  delete p_pS;
  std::cout << "Aufruf des Destruktors" << std::endl;
}
void Sum::setI(int i)
{
  if (i > 0) // Abfrage zu Wertebereich möglich
  {
    p_i = i; // Instanzvariable setzen
    *p_pS = p_i + p_j; // ggf. abhängige Variable setzen
  }
}

int main()
{
  std::cout << "Anfang" << std::endl;
  Sum tS;          // Statische Erzeugung mit Standardkonstruktor
  tS.setI(6);     // Zugriff auf p_i nur über Funktion möglich.
  Sum* pS1;
  pS1 = new Sum(); // Dynamische Erzeugung mit Standardkonstruktor
  pS1->setI(2);     // Zugriff auf p_i nur über Funktion möglich.
  Sum* pS2;
  pS2 = new Sum(1,3); // Dynamische Erzeugung mit Nicht-Standardkonstruktor
  pS2->setI(4);     // Änderung p_i nur über Funktion möglich.
  std::cout << tS.getS() << std::endl;
  std::cout << pS1->getS() << std::endl;
  std::cout << pS2->getS() << std::endl;
  delete pS1;
  delete pS2;
  std::cout << "Ende" << std::endl;
}
```

Anhand der Ausgabe ist zu erkennen, wann Konstruktor und Destruktor aufgerufen werden und welche Wirkung die aufgerufenen Memberfunktionen haben:

```
Anfang
Standardkonstruktor
Standardkonstruktor
Nicht-Standardkonstruktor
6
2
7
Aufruf des Destruktors
Aufruf des Destruktors
Ende
Aufruf des Destruktors
```

Im Konstruktor wird mit **new** Speicherplatz für das Summenelement reserviert und initialisiert (Hier als Beispiel ein Zeiger auf `int`, ansonsten für einfaches `int` nicht sinnvoll.).

Der Destruktor gibt den reservierten Speicherplatz wieder frei. Sein Aufruf erfolgt automatisch, wenn der Gültigkeitsbereich des Objektes verlassen wird oder bei dynamischen Objekten **delete** aufgerufen wird.

Eine Initialisierung von Werten einer Klasse kann anstatt mit Zuweisung im Konstruktorrumpf auch mit einer sogenannten Konstruktor-Initialisierungsliste geschehen. Ein Konstruktor der Form:

```
Klasse::Klasse(Typ1 a, Typ2 b)
{
  private_a = a;
  private_b = b;
  /* ... */
};
```

wird mittels Initialisierungsliste zu:

```
Klasse::Klasse(Type1 a, Typ2 b) : private_a(a), private_b(b)
{
  /* ... */
};
```

Die zweite Art der Initialisierung ermöglicht im Gegensatz zur ersten auch die Initialisierung von Konstanten. Datenelemente, denen dynamisch (also zur Laufzeit des Programms) Speicher zugewiesen wird, können allerdings nur im Rumpf des Konstruktors initialisiert werden. Es kann auch keine Wertebereichsprüfung durchgeführt werden. Bei mehreren Konstruktoren wird anhand von Anzahl und Typ der Parameter unterschieden, welcher Konstruktor aufgerufen wird (s. Kapitel 2.4).

Beispiel (als Ersatz für die Konstruktoren im vorigen Beispiel):

```
Sum::Sum():p_i(0),p_j(0)
{
  p_pS = new int;
  *p_pS = 0;
}
Sum::Sum(int i, int j):p_i(i),p_j(j)
{
  p_pS = new int;
  *p_pS = p_i + p_j; // ggf. abhängige Variable setzen
}
```

Neben dem *Standardkonstruktor* (ohne Parameter) gibt es noch einen anderen ausgezeichneten Typ eines Konstruktors: den *Copykonstruktor*. Dieser hat als Parameter eine (konstante) Referenz auf seine Klasse.

Beispiel:

```
class Klasse
{
  Klasse();          // Standardkonstruktor
  Klasse(const Klasse&);  // Copykonstruktor
};
```

Beim Copykonstruktor gilt aus historischen Gründen, dass er implizit existiert, wenn er nicht definiert wurde. Er kopiert dann Byte für Byte. Der Copykonstruktor wird automatisch immer dann aufgerufen, wenn eine Variable an eine Funktion/Methode nicht als Referenz übergeben wird. Dies kann unangenehme Folgen haben, wenn innerhalb der Klasse dynamische Datenstrukturen existieren. Stellen Sie sich hierzu einen Zeiger auf einen String vor. Wenn das Objekt byteweise kopiert wird, dann wird der Zeiger kopiert und nicht der Inhalt. Es existieren dann zwei Zeiger auf den gleichen String. Das Löschen eines der Objekte führt dann unter Umstände dazu, dass der String gelöscht wird und das verbleibende Objekt weiter auf den nicht mehr existierenden String verweist (*wilder Zeiger*). Dies ist gefährlich und sollte vermieden werden. Daher sollte man den Copykonstruktor immer dann selbst definieren, wenn die Klasse dynamische Elemente hat, oder den Aufruf eines Copykonstruktors ganz verbieten (d. h. Fehler zur Compilierzeit, wenn nicht mit Referenzen gearbeitet wird). Dazu kann man ihn in der Klasse als **private** deklarieren:

```
class Klasse
{
  private:
    Klasse(const Klasse&);
};
```

Der Parameter des Copykonstruktors muss eine Referenz sein, da ansonsten bei seinem Aufruf wieder kopiert würde, was zu einer unendlichen Rekursion führen würde.

Als Beispiel soll in einer einfachen Stringklasse der Copykonstruktor definiert werden:

```
class HString
{
  private:
    char* p_pString;
    int p_iSize;
  public:
    HString();
    HString(const HString&);
};

HString::HString():
  p_iSize(0), p_pString(0)
{}

HString::HString(const HString& aHString)
{
  p_iSize = aHString.p_iSize;
  p_pString = new char [iSize + 1];
  strcpy(p_pString, aHString.p_pString);
}
```

Konstante Elementfunktionen

Methoden können als konstant deklariert werden. Dazu muss hinter der Deklaration das Schlüsselwort `const` hinzugefügt werden.

Beispiel:

```
class Klasse
{
  public:
    int getElement() const;
  private:
    int iElement;
};

int Klasse::getElement() const
{
  ++iElement;   // Fehler, da Element verändert wird
  return iElement;
}
```

Konstante Methoden dürfen den Zustand des Objektes nicht verändern, weswegen es im Beispiel an gekennzeichneter Stelle zu einem Fehler beim Compilieren kommt. Dadurch wird es ermöglicht, eine semantische Beschränkung zu spezifizieren, die zur Compilierzeit geprüft wird. Der Aufwand konstante Methoden zu deklarieren ist meist sehr viel kleiner als Fehler zu finden, die durch ungewolltes Verändern eines Zustandes entstehen. Daher sollte man wann immer möglich Methoden als konstant deklarieren. Konstante Methoden dürfen nur ebensolche Methoden aufrufen.

Statische Klassenelemente

Data Member einer Klasse können mit Hilfe des Schlüsselworts `static` als statisch deklariert werden. Von statischen Klassenmitgliedern wird nur *ein* Exemplar pro Klasse erzeugt, unabhängig davon, wie viele Objekte der Klasse instanziiert werden. Eine statische Variable ist also der Klasse selbst zugeordnet und nicht den Objekten der Klasse. Statische Variablen können praktisch als globale Variablen im Kontext einer Klasse betrachtet werden.

Auch Funktionen können statisch deklariert werden, wenn sie Zugriff auf Elemente einer Klasse benötigen, jedoch nicht für ein bestimmtes Objekt aufgerufen werden sollen. Sie können über den vorangestellten Klassennamen aufgerufen werden.

Da statische Data Member existieren, ohne dass ein Objekt der Klasse existieren muss, erfolgt ihre Initialisierung außerhalb der Klasse und außerhalb jeder Funktion (auch außerhalb von main). Diese Definition muss erfolgen, da sonst kein Speicherplatz reserviert wird und der Linker bei der Nutzung der Variable eine Fehlermeldung ausgibt. Zur besseren Übersicht sollte der Code dazu im Sourcefile der jeweiligen Klasse stehen. Bei der

Initialisierung muss der Datentyp der Variablen wiederholt werden und es kann ein Wert zugewiesen werden.

Das nächste Programmbeispiel zeigt den Unterschied von statischen und nicht-statischen Variablen sowie die Benutzung einer statischen Funktion:

```
#include <iostream>

class Beispiel_static
{
  public:
    Beispiel_static()
    {
      iAnzahl++;
      iNummer = iAnzahl;
    }

    static void vPrintAnzahl() // statische Methode
    {
      std::cout << "Anzahl erzeugter Objekte: " << iAnzahl;
    }

    void vPrintNummer()
    {
      std::cout << "Nr. des erzeugten Objekts: " << iNummer;
    }

    void vPrint()
    {
      vPrintAnzahl(); std::cout << " ... ";
      vPrintNummer(); std::cout << endl;
    }
  private:
    static int iAnzahl; // statische Variable
    int iNummer;
};

// Initialisierung der statischen Variablen
int Beispiel_static::iAnzahl = 0;

int main()
{
  Beispiel_static::vPrintAnzahl(); cout << endl;
  Beispiel_static a;
  a.vPrint();
  Beispiel_static b;
  a.vPrint();
  b.vPrint();
}
```

Die Ausgabe dieses Programmes lautet:

```
Anzahl erzeugter Objekte: 0
```

```
Anzahl erzeugter Objekte: 1 ... Nr. des erzeugten Objekts: 1
Anzahl erzeugter Objekte: 2 ... Nr. des erzeugten Objekts: 1
Anzahl erzeugter Objekte: 2 ... Nr. des erzeugten Objekts: 2
```

Der `this`-Zeiger

Werden mehrere Objekte einer Klasse erzeugt, so sind nur die Data Member in jedem Objekt vorhanden. Die Methoden hingegen werden für jede Klasse nur einmal erzeugt, da ihr Code für jedes Objekt gleich ist. Eine Mehrfacherzeugung wäre unnötige Speicherverschwendung. Bei Aufruf einer Methode muss der Compiler daher wissen, für welches Objekt der Klasse die Methode ausgeführt werden soll. Dies wird erreicht, indem der Methode als erstes Argument immer die Adresse des jeweiligen Objektes übergeben wird. Diese Adresse wird als `this`-Zeiger bezeichnet. Die Übergabe geschieht automatisch und unsichtbar für den Programmierer. Beispielsweise wird die Implementation der Methode

```
void myClass::printName()
{
  std::cout << "Der Name ist: " << name << std::endl;
}
```

intern in

```
void printName(myClass* this)
{
  std::cout << "Der Name ist: "<< this->name << std::endl;
}
```

konvertiert. Ebenso wird der Aufruf der Funktion von

```
classObject.printName();
```

in das Konstrukt

```
printName(&classObject);
```

konvertiert. Anwendungsbeispiele zur Verwendung des `this`-Zeigers finden Sie in Kapitel 2.4.

2.2 Vererbung / Inheritance

Die Vererbung (engl.: *Inheritance*) ist ein Mechanismus, der es erlaubt, Datenstrukturen und Methoden einer Klasse (Basisklasse) bei der Bildung neuer Klassen (Unterklasse, abgeleitete Klasse) wiederzuverwenden. In der Unterklasse kann die Datenstruktur erweitert werden und/oder die Methoden erweitert oder modifiziert werden. Weiterhin stellt die Vererbung ein Hilfsmittel zur Strukturierung von Anwendungsproblemen dar, die im allgemeinen viele ähnliche Elemente beinhalten.

```
class <KlassenName>:<Zugriffsspezifizierer> <BasisklassenName>
```

Mit der obigen Syntax wird eine Klasse vereinbart, die alle Datenelemente und Methoden einer bereits vorhandenen Klasse erbt. Der Zugriffsspezifizierer gibt an, wie Datenelemente und Methoden der Basisklasse innerhalb der abgeleiteten Klasse maximal zugänglich sind. Hat die Basisklasse alle Datenelemente z. B. `public` deklariert, und der Zugriffsspezifizierer ist `private`, so gelten alle Datenelemente der Basisklasse für ein Objekt der abgeleiteten Klasse als `private` deklariert.

Mit der Vererbung wird neben `public` und `private` die Einführung eines weiteren Schlüsselworts zur Kennzeichnung von Zugriffsrechten auf Datenelemente und Methoden notwendig. Das Schlüsselwort heißt `protected`. Genau wie auf `private`-Mitglieder einer Klasse kann auch auf `protected`-Mitglieder nicht von außerhalb der Klasse zugegriffen werden, jedoch werden `protected`-Mitglieder an eine abgeleitete Klasse vererbt, `private`-Mitglieder nicht.

Elementzugriff in der Basisklasse	Zugriffsspezifizierer der Klasse	Elementzugriff in der Ableitungsklasse
`private`	`public`	*nicht möglich*
`protected`	`public`	`protected`
`public`	`public`	`public`
`private`	`protected`	*nicht möglich*
`protected`	`protected`	`protected`
`public`	`protected`	`protected`
`private`	`private`	*nicht möglich*
`protected`	`private`	`private`
`public`	`private`	`private`

Tabelle 2.1: Zugriffsmöglichkeiten auf Basisklassenelemente

Tabelle 2.1 gibt einen Überblick über die Zugriffsmöglichkeiten auf Elemente von Basisklassen. In der Regel verwendet man für die Klasse den Zugriffsspezifizierer `public`, da dann der Zugriff auf Mitglieder der abgeleiteten Klasse in gleicher Weise möglich ist wie der Zugriff auf Mitglieder der Basisklasse. Beachte: Der Defaultwert ist `private`.

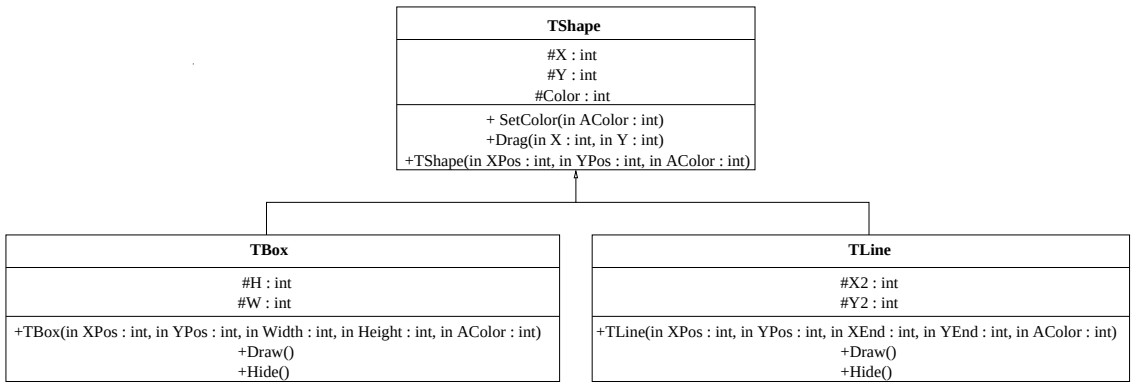


Abbildung 2.1: Darstellung einer Klassenhierarchie

Abbildung 2.1 zeigt ein Beispiel für die Verwendung einer Klassenhierarchie anhand von Zeichnungselementen, wie man sie bei grafischen Benutzeroberflächen findet.

```
// Klassenvereinbarungen
class TShape
{
  protected:
    int p_iColor;
    int p_iX, p_iY;
  public:
    TShape(); // Standardkonstruktor
    TShape(int iXPos, int iYPos, int iAColor);  // Konstruktor
    void setColor(int iAColor);
    void vDrag(int iX, int iY);
};

class TBox: public TShape
{
  private:
    int p_iW, p_iH;
  public:
    TBox(int iXPos, int iYPos, int iWidth, int iHeight, int iAColor);
    void vDraw();
    void vHide();
};

class TLine: public TShape
{
  private:
    int p_iX2, p_iY2;
  public:
    TLine(int iXPos, int iYPos, int iXEnd, int iYEnd, int iAColor);
    void vDraw();
    void vHide();
};

// Konstruktoren
TShape::TShape()
 : p_iColor(0),p_iX(0),p_iY(0)
{
}

TShape::TShape(int iXPos, int iYPos, int iAColor)
{
  p_iColor = iAColor;
  p_iX = iXPos;
  p_iY = iYPos;
}

TBox::TBox(int iXPos, int iYPos, int iWidth, int iHeight, int iAColor)
 : TShape(iXPos, iYPos, iAColor)  // Aufruf des Konstruktors der Basisklasse!
{
  p_iW = iWidth;
  p_iH = iHeight;
}

TLine::TLine(int iXPos, int iYPos, int iXEnd, int iYEnd, int iAColor)
  : TShape(iXPos, iYPos, iAColor) // Aufruf des Konstruktors der Basisklasse!
```

```
  , p_iX2(iXEnd), p_iY2(iYEnd) // Initialisierung hier über Liste
{
}


// übrige Methoden
void TShape::setColor(int iAColor)
{
  p_iColor = iAColor;
}

void TShape::vDrag(int iXPos, int iYPos)
{
  p_iX = iXPos;
  p_iY = iYPos;
}

void TBox::vDraw()
{
  // Rechteck zeichnen
}

void TBox::Hide()
{
  int iAColor = p_iColor;
  p_iColor = 0; // Schwarz zeichnen, hier über Variable
  vDraw();
  p_iColor= iAColor;
}

void TLine::Draw()
{
  // Linie zeichnen
}

void TLine::Hide()
{
  int iAColor = p_iColor;
  setColor(0);  // Schwarz zeichnen, hier über Methode
  vDraw();
  setColor(iAColor);
}

// Hauptprogramm
void main()
{
  TBox tBox(10,10,20,20,1);
  TLine tLine(10,10,30,30,1);

  // Objekte zeichnen
  tBox.vDraw();
  tLine.vDraw();

  // Objekte verschieben
  tBox.vHide();
  tBox.vDrag(0,0);
  tBox.vDraw();
```

```
  tLine.vHide();
  tLine.vDrag(0,0);
  tLine.vDraw();
}
```

Ein Objekt der abgeleiteten Klasse ist immer auch ein Objekt der Basisklasse. Die Umkehrung gilt nicht. Durch Casting kann man ggf. die Zugehörigkeit zu einer bestimmten Unterklasse erzwingen. Dies ist zum Beispiel erforderlich, wenn man das Ergebnis einer allgemeinen Methode in einer Unterklasse weiterverwenden will:

```
TShape* ptShape;
TLine* ptLine;
//..
ptShape = ptLine  // o.k.
ptLine = static_cast<TLine*>(ptShape)  // Casting erforderlich
```

Die Konstruktoren der Basisklasse können über einen Aufruf in der Initialisierungsliste ausgewählt werden. Wird dort kein Konstruktor angegeben, wird der Standardkonstruktor der Basisklasse aufgerufen.

```
TLine::TLine(int iXPos, int iYPos, int iXEnd, int iYEnd, int iAColor)
  : TShape(iXPos, iYPos, iAColor)
  , p_iX2(iXEnd), p_iY2(iYEnd)
{}
```

Die Initialisierung erfolgt in folgender Reihenfolge:

1. Initialisierung der Basisklasse
2. Initialisierungsliste
3. Ausführung des Codes im Konstruktor

2.2.1 Polymorphie

Besteht für verschiedene Klassen eine gemeinsame Funktionalität (z. B. `vDraw`), deren Realisierung für die Klassen aber unterschiedlich ist, so kann man in den verschiedenen Klassen Methoden mit gleicher Schnittstelle (Name, Typ, Parameter) einrichten, die dann die entsprechende angepasste Rolle übernehmen. Man spricht dann von Polymorphie (= Vielgestaltigkeit). Möchte man Polymorphie in einer Basisklasse und einer abgeleiteten Klasse nutzen, muss die Funktion in der Basisklasse als virtuell (Schlüsselwort `virtual`) deklariert werden. Virtuelle Funktionen wirken sich nur bei dynamischen Objekten aus.

Welche Funktion bei polymorpher Deklaration für ein dynamisches Objekt benutzt wird, wird erst zur Laufzeit entsprechend dem Typ des Objektes, für das die Methode aufgerufen wird, ermittelt. Man spricht hier auch von dynamischer bzw. später Bindung (engl.: *late binding*). Hierbei wird beginnend mit der höchsten Basisklasse geprüft, ob eine virtuell deklarierte Funktion in der abgeleiteten Klasse überschrieben wird. Ist dies der Fall, wird diese benutzt, ggf. bei virtueller Deklaration wieder mit Prüfung in der Unterklasse.

Möchte man gezielt die Methode einer bestimmten Klasse aufrufen, so kann dies explizit durch Angabe des Klassennamens (Vorfahren `Klasse::Methode(...);`) geschehen.

Betrachtet man in diesem Zusammenhang das Beispiel in Kapitel 2.2 genauer, wird man feststellen, dass die Methoden `vHide` bei `TBox` und bei `TLine` identisch sind, bis auf den Aufruf einer unterschiedlichen `vDraw`-Methode.

Wenn man in die Basisklasse (`TShape`) eine virtuelle Methode `vDraw` einfügt, wird erst zur Laufzeit entschieden, welche Methode (`TBox::vDraw()` oder `TLine::vDraw()`) tatsächlich aufgerufen wird. Aus diesem Grund brauchen `TBox` und `TLine` auch keine eigene `vHide`-Methode mehr, sondern diese kann bereits in `TShape` deklariert werden. Ebenfalls kann `vDrag` nun selbständig das Objekt auf dem Bildschirm löschen und neuzeichnen. Damit ergibt sich folgendes geänderte Codefragment für die Klassen:

```
class TShape
{ ...
  virtual void vDraw() {};  // inline Definition
                            // (leerer Anweisungsblock)
  void vHide();
  ...
};
...
void TShape::vHide()
{
  int iAColor = p_iColor;
  setColor(0);      // zum Löschen schwarz zeichnen
  vDraw();
  setColor(iAColor);
}

void TShape::vDrag(int iXPos, int iYPos)
{
  vHide();
  p_iX = iXPos;
  p_iY = iYPos;
  vDraw();
}
...
// TBox::vHide und TLine::vHide entfallen

// Hauptprogramm
void main()
{
  TShape* ptBox = new TBox(10,10,20,20,1);
  TShape* ptLine = new TLine(10,10,30,30,1);

  // Objekte zeichnen
  ptBox->vDraw();
  ptLine->vDraw();

  // Objekte verschieben
  ptBox->vDrag(0,0);
  ptLine->vDrag(0,0);
```

```
}
```

In diesem Fall wird jeweils die Methode `vDraw` aus TBox bzw TLine aufgerufen. Würde man hier das `virtual` in `TShape` weglassen, würde die Funktion aus der Basisklasse des Zeigers (`TShape`) aufgerufen. Es ist nicht sinnvoll, alle Methoden einer Klasse ohne Grund `virtual` zu definieren, da die Auswertung der späten Bindung zur Laufzeit einen höheren Aufwand erfordert.

Angemerkt sei noch, dass Destruktoren immer `virtual` erklärt werden sollten. Ein virtueller Destruktor sorgt dafür, dass Nachfahren korrekt gelöscht werden können, auch wenn man nur einen Zeiger hat, der vom Typ des Vorfahren ist. Ein Nachfahrendestruktor ruft zudem automatisch den Vorfahrendestruktor auf, um sicherzustellen, dass der durch den Vorfahren belegte Speicherplatz freigegeben wird.

Beispiel:

```
class BasisKlasse
{
  public:
    BasisKlasse(){};              // Konstruktor
    virtual ~BasisKlasse() {};    // inline Def. Destruktor
};

class Nachfahre : public BasisKlasse
{
  private:
    int* p_pInhalt;
  public:
    Nachfahre(int* pData, int iSize); // Konstruktor
    virtual ~Nachfahre(); // Destruktor
};

Nachfahre::Nachfahre(int* pData, int iSize) : BasisKlasse()
{
  p_pInhalt = new int [iSize];
  for (int i=0;i<iSize;i++)
  {
     p_pInhalt[i] = pData[i];
  }
}

Nachfahre::~Nachfahre()
{
  delete [] p_pInhalt;
}

void main()
{
  BasisKlasse* AObject = new Nachfahre({1,2,3},3); // Zuweisung OK!
  ...
  delete AObject;  //hier wird ~Nachfahre() aufgerufen!
}
```

Konstruktoren dürfen jedoch nicht virtuell erklärt werden. Wäre ein Konstruktor eine virtuelle Methode, so würde seine Aufrufadresse erst während der Laufzeit ermittelt. Diese müsste dann aber in der Liste stehen, die der Konstruktor bei der Erzeugung eines Objekts anlegt. Die Adresse des Konstruktors wäre also erst dann verfügbar, wenn er bereits aufgerufen wurde („Henne und Ei“-Problem).

2.2.2 Abstrakte Klassen

Zum Aufbau von Klassenstrukturen ist es oft sinnvoll, gemeinsame Eigenschaften mehrerer Klassen in einer Oberklasse zusammenzufassen, obwohl eigentlich keine Objekte dieser Oberklasse existieren. So könnte z. B. eine Basisklasse Fahrzeuge bereits alle notwendigen Funktionen zur Beschreibung von PKW und LKW enthalten. Eine solche Klasse nennt man „Abstrakte Klasse“. Es können keine Objekte einer abstrakten Klasse erzeugt werden. In C++ realisiert man eine abstrakte Klasse durch Definition mindestens einer „rein virtuellen“ Funktion. Eine rein-virtuelle Methode wird dadurch deklariert, dass sie den Wert 0 zugewiesen bekommt. Es ergibt sich folgende Syntax für eine solche Methode:

```
virtual <Typ> <Funktionsname> (<Parameterliste>) = 0;
```

Eine rein-virtuelle Funktion braucht in der abstrakten Klasse nicht implementiert zu werden, kann dies aber. Die rein-virtuellen Funktionen müssen in allen abgeleiteten Klassen überschrieben werden.

2.2.3 Mehrfachvererbung

In C++ ist es möglich, eine Klasse von mehreren Basisklassen abzuleiten. Sie erbt dann die Datenelemente und Methoden aller Oberklassen. Mehrfachvererbung sollte jedoch nicht ersatzweise zur Darstellung einer has-a-Beziehung verwendet werden: Ein Personenzug kann z. B. nicht als Unterklasse von Zug (enthält Lokomotive) und Personenwaggons dargestellt werden, sondern ist Unterklasse von Zug und enthält Personenwaggons. (Fast) alle Probleme lassen sich ohne Mehrfachvererbung darstellen.

Eine Mehrfachvererbung wird einfach durch Angabe aller Basisklassen hinter dem Ableitungsoperator implementiert. Häufig entstehen Vererbungsunklarheiten (oder es werden Konstruktoren mehrfach aufgerufen), wenn die Basisklassen selber von einer gemeinsamen Oberklasse erben. Dann müssen gezielt einzelne Basisklassen mit dem Schlüsselwort `virtual` virtuell erklärt werden.

Im folgenden Beispiel vererbt die Grundklasse an zwei abgeleitete Klassen und diese wiederum an eine Klasse mit dem Namen `ZweifachabgeleiteteKlasse`:

```
class Grundklasse
{
  ...
};

class AbgeleiteteKlasse1: virtual public Grundklasse {...};
class AbgeleiteteKlasse2: virtual public Grundklasse {...};
```

```
class ZweifachabgeleiteteKlasse:
  public AbgeleiteteKlasse1, public AbgeleiteteKlasse2
  {...};
```

Ein Mehrfachaufruf des Konstruktors der Grundklasse bei Instanziierung eines Objekts der Klasse ZweifachabgeleiteteKlasse wird hier vermieden.

2.3 Freundschaften

Ein grundlegendes Konzept von C++ ist die Datenkapselung, d. h. dass nur die Methoden einer Klasse Zugriff auf die privaten Mitglieder besitzen. Hierdurch kann das Objekt selbst kontrollieren, welche Werte in seine Data Member geschrieben werden und Fehleingaben abfangen. In Ausnahmefällen kann es jedoch sinnvoll sein, dass auch Funktionen, die nicht Mitglied der Klasse sind, Zugriff auf deren private Mitglieder erhalten. Dies lässt sich durch `friend`-Funktionen erreichen. Eine als `friend` deklarierte Funktion ist nicht Mitglied der Klasse, hat aber Zugriff auf die privaten Mitglieder dieser Klasse.

Es sei betont, dass die Klasse selbst die „Freundschaft“ anbieten muss, d. h. sie muss der betreffenden Funktion den Zugriff auf ihre Mitglieder erlauben. Da durch `friend`-Funktionen das Konzept der Datenkapselung unerwünscht umgangen wird, sollte man diese Funktionen so sparsam wie möglich einsetzen. Viele Programmierer sind der Ansicht, dass ihre Verwendung nicht nötig ist. Das Beispiel zeigt die Verwendung von Freundschaften:

```
#include <string>
#include <iostream>

class Person
{
  private:
    int p_iAlter;
    std::string p_sName;
  public:
    Person(int i, std::string sWort); // Konstruktor
    ~Person();  // Destruktor
    void vAnzeigen();
    // hier wird der Funktion "vGeburtstag" die
    // Freundschaft angeboten
    friend void vGeburtstag(Person a);
};

Person::Person(int i, std::string sWort)
  : p_iAlter(i)
{
  p_sName = p_sWort;
}

Person::~Person()
{
}
```

```
void Person::vAnzeigen(void)
{
  std::cout << p_sName << " ist "
    << p_iAlter << " Jahre alt." << std::endl;
}


void vGeburstag(Person &aPers) // globale Methode, nicht in Klasse
{
  // Zugriff auf privates Mitglied von a
  aPers.p_iAlter++;
}

// Hauptprogramm
int main()
{
  Person hans(42, "Hans");
  hans.anzeigen();
  geburtstag(hans);
  hans.anzeigen();
}
```

Die Ausgabe des obigen Programmes lautet:

```
Hans ist 42 Jahre alt.
Hans ist 43 Jahre alt.
```

Es sei noch erwähnt, dass eine Klasse nicht nur Funktionen, sondern auch ganzen Klassen die Freundschaft anbieten kann. Alle Mitgliedsfunktionen der befreundeten Klasse erhalten unbeschränkten Zugriff auf alle Mitglieder der eigenen Klasse.

Man sollte die Verwendung von Freundschaften soweit wie möglich vermeiden und z. B. durch entsprechende Zugriffsfunktionen ersetzen, da bei jeder Änderung der Daten sonst ggf. auch in allen befreundeten Klassen der Zugriff geändert werden muss.

2.4 Überladen

Normalerweise erlauben es Programmiersprachen, lediglich eine einzige Definition für ein und denselben Funktionsnamen anzugeben, damit der Compiler eindeutig entscheiden kann, welcher Code verwendet werden soll. In C++ gilt diese Einschränkung nicht. Es können mehrere Funktionen gleichen Namens definiert werden. Diese Technik wird als *Überladen* (engl. *overloading*) bezeichnet. Überladen ist dabei nicht nur für Funktionen sondern auch für Operatoren möglich.

2.4.1 Funktionen / Methoden

Für überladene Funktionen stellt sich die Frage, anhand welchen Kriteriums der Compiler entscheidet, welcher Code auszuführen ist. Beim Aufruf einer Funktion müssen die übergebenen Argumente, die bei der Funktionsdefinition angegebenen Datentypen besitzen. Der

Compiler trifft die Entscheidung über den auszuführenden Code anhand der übergebenen Argumente. Das bedeutet, dass sich überladene Funktionen in der Anzahl und/oder dem Typ der übergebenen Argumente unterscheiden müssen. Bitte beachten Sie, dass ein veränderter Rückgabetyp einer Funktion kein Kriterium darstellt. Der Compiler meldet einen Fehler, wenn sich zwei Funktionen lediglich im Typ ihres Rückgabewertes unterscheiden. Das folgende Beispiel zeigt eine überladene Funktion, die das Maximum zweier `int`- oder `char`-Werte liefert:

```
int max(int a, int b) { return (a > b) ? a : b; }
char max(char a, char b) { return (a > b) ? a : b; }
```

Der Versuch, die Funktion ein weiteres Mal zu überladen, so dass der größere zweier `int`-Werte gedruckt wird, führt zu einem Fehler:

```
void max(int a, int b)
{
  if (a > b)
    cout << a << " ist groesser als " << b << endl;
  else if (b > a)
    cout << b << " ist groesser als " << a << endl;
  else
    cout << a << " ist gleich " << b << endl;
}
```

Diese Funktion unterscheidet sich von `int max(int a, int b)` nur im Typ des Rückgabewerts (`void` statt `int`). Daher kann der Compiler beim Funktionsaufruf nicht entscheiden, welcher Code ausgeführt werden soll.

Ähnlich der außerhalb einer Klasse definierten Funktionen können auch klasseneigene Methoden überladen werden. Den verschiedenen Versionen einer überladenen Methode können verschiedene Zugriffsrechte gegeben werden. Eine sehr häufig überladene Methode ist der Konstruktor. Folgendes Beispiel hat drei Konstruktoren, einen Standardkonstruktor, einen parametrisierten Konstruktor und einen privaten Copykonstruktor.

```
#include <string>

class Student
{
  private:
    std::string p_sName;
    int p_iSemester;
    Student(const Student& aStud);  // Copykonstruktor
  public:
    Student();                      // Standard-Konstruktor
    Student(std::string s, int n);  // param. Konstruktor
};

Student::Student{ ... }
Student::Student(std::string s, int n) { ... }

int main()
```

```
{
  Student student;                // Standardkonstruktor
  Student arthur("Arthur", 42); // param. Konstruktor
  Student s3(s1);                 // Fehler, Copykonstruktor private
}
```

Man beachte die Unterschiede zwischen

1. Vererbung (Methodenname aus Basisklasse)
2. Polymorphie (Gleicher Methodenname und gleiche Parameter in verschiedenen Klassen)
3. Überladung (Gleicher Methodenname und verschiedene Parameter in einer Klasse)

2.4.2 Operatoren

In C++ sind Operatoren prinzipiell als Funktionen definiert und können somit wie Funktionen benutzt und auch überladen werden. Der Name einer Operatorfunktion besteht aus dem festen Bestandteil `operator` und seinem Zeichen. Der Funktionsname für den Operator + lautet also `operator+` und die Anweisung `int a = 4 + 3;` wird vom Compiler für den Programmierer unsichtbar in `int a = operator+(4, 3);` übersetzt.

Da es unsinnig wäre, das Verhalten der Operatoren auf in C++ eingebauten Typen (z. B. `int`) zu verändern, muss bei der Operatorüberladung mindestens ein Operand ein Objekt sein. Operatoren auf komplexeren Datenstrukturen können individuell und problemangepasst definiert werden. So stellt z. B. die Implementation eines Plus-Operators zur Stringverkettung, wie er in anderen Programmiersprachen standardmäßig implementiert ist, in C++ kein Problem dar. Folgende Operatoren können in C++ überladen werden:

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

In C++ werden Klassenmethoden wie normale Funktionen behandelt, die implizit als ersten Parameter den `this`-Zeiger der zugehörigen Instanz bekommen. Gleiches gilt selbstverständlich auch für Operatoren, die somit innerhalb (als Klassenmethode) oder auch außerhalb (als Funktion) einer Klasse definiert werden können. Ein zweistelliger Operator kann durch eine Klassenmethode mit einem Parameter oder durch eine globale Funktion mit zwei Parametern definiert werden. Ein einstelliger Operator kann durch eine Klassenmethode ohne Parameter oder durch eine globale Funktion mit einem Parameter definiert werden.

Was ist besser, Operatordefinition innerhalb oder außerhalb der Klasse? Leider lässt sich diese Frage nicht allgemeingültig beantworten. Als Regel schlägt Bjarne Stroustrup (der

Entwickler von C++) vor: Eine Funktion sollte Klassenmethode sein, soweit es keinen guten Grund dagegen gibt. Manchmal ist eine Definition innerhalb der Klasse unmöglich, wenn z. B. der erste Parameter ein nicht-Klassenobjekt sein soll, da ja bei Methoden der `this`-Zeiger immer implizit der erste Parameter ist. Beispiel hierfür sind der Ein- und Ausgabeoperator (`<<, >>`), bei denen der erste Operand ein istream bzw. ostream ist, oder auch bei arithmetischen Operatoren auf gemischten Typen, also z. B. `operator+(int, X)`.

Als Beispiel folgt die Deklaration der am häufigsten überladenen Operatoren für eine Typklasse X:

```
class X
{
  public:
    const X operator++(int);     // Postfix-Inkrement
    X& operator++();             // Praefix-Inkrement
      // analog fuer --
    X& operator+=(const X&);     // X+=X
      // analog fuer alle op=
    X& operator+(const X&);      // arithm. +
      // analog fuer arith., logische und bit Operatoren
    bool operator==(const X&);   // Vergleich
      // analog fuer !=,<,>,<=,>=
    X& operator[](int);          // Subskript
    void operator()();           // Funktionsaufruf
    operator int() const;        // Konvertieroperator
};

istream& operator>>(istream&, X&);
ostream& operator<<(ostream&, const X&);
```

Anmerkungen:

- Wie bei den meisten Methoden sollten konstante Referenzen übergeben werden, damit Objekte nicht beim Funktionsaufruf kopiert werden müssen, was unter Umständen sehr lange dauern kann. Man kann sich die Übergabe konstanter Referenzen als ein schnelles call-by-value vorstellen. Schnell, weil nur eine Referenz übergeben wird und call-by-value weil auf den Wert nur lesend zugegriffen werden darf.

- Die Zuweisungsoperatoren (`=, +=, -=, ...`) sollten immer eine nicht konstante Referenz auf this zurückgeben, da dies bei den C++ Standardtypen (`int, double, ...`) auch so üblich ist. Dadurch sind Zuweisungsketten wie `a=(b=c);` oder auch `(a=b)=c;` möglich. Über den Sinn solcher Zuweisungsketten lässt sich streiten, aber warum sollte man von bestehenden Konventionen abweichen.

- Damit der Operator zum Index-Zugriff (`operator[]`) auch auf der linken Seite einer Zuweisung vorkommen kann (z. B. `a[i] = 5;`), sollte er eine Referenz zurückgeben. Weiterhin kann der Zugriffsoperator anstatt int auch einen anderen Parameter bekommen, z. B. ist bei der STL-map der Parameter ein `const string &`.

- Um über Zeiger auf Elemente eines Objektes zugreifen zu können wurde der Operator `->` definiert, d. h. der Zugriff erfolgt dermaßen: `Zeiger->Element`. Dies ist die verkürzte Schreibweise für `(*Zeiger).Element`.

- Syntaktisch wird die Definition von Präfix bzw. Postfix durch den übergebenen int unterschieden, er hat ansonsten keine weitere Bedeutung und wird auch meistens deswegen nicht benannt. Um den semantischen Unterschied zwischen den Präfix- und Postfix- Varianten der Inkrement- bzw. Dekrement-Funktion zu verstehen, merkt man sich am besten folgende Regel. Präfix bedeutet „erhöhen und holen“, während Postfix „holen und erhöhen“ bedeutet. Die Postfix-Variante kann keine Referenz zurückgeben, denn hier muss ein temporäres Objekt zurückgegeben werden, da ja der alte Wert zurückgeliefert werden muss, während der aktuelle Wert inkrementiert wird. Für Integer würden die beiden Inkrementversionen folgendermaßen aussehen:

```
//Praefix-Inkrement
MyInt& MyInt::operator++() // Referenz-Rückgabe
{
  *this += 1;
  return *this;
}

//Postfix-Inkrement
const MyInt MyInt::operator++(int) // Kopie-Rückgabe
{
  MyInt oldValue = *this;
  ++(*this);
  return oldValue;
}
```

  Die Postfix-Variante sollte einen konstanten Wert zurückliefern, da dies zum einen Standard in C++ ist und weiterhin Anweisungen wie `i++++;` zurückgewiesen werden, da man ansonsten ein nichtintuitives Verhalten hat (Das zweite ++ erhöht nur eine temporäre Variable). Allerdings macht die Anweisung `++++i;` durchaus Sinn und ist durch die Referenzrückgabe auch möglich.

- Alle Operatoren, bei denen als Operand (das, was links vom Operator steht) ein Element der Klasse definiert ist, können (sollen) innerhalb der Klasse definiert werden, damit im Operator auf die privaten Variablen der Klasse zugegriffen werden kann. Ist der Operand kein Element der Klasse (wie bei den Ein-/Ausgabe-Operatoren, wo es der entsprechende Stream ist), so muss der Operator außerhalb der Klasse definiert werden. Dies führt dazu, dass auf die privaten Variablen nicht zugegriffen werden kann. Dort ist es meist sinnvoll, eine entsprechende Memberfunktion zu implementieren und diese dann aufzurufen oder getter-Funktionen zu benutzen.

  Der Ausgabeoperator `operator<<()` sollte eine Referenz auf `ostream` zurückliefern, damit Anweisungen wie `cout << a << b;` möglich sind. Dies ist einsichtig, wenn man sich diese Anweisung in einer Notation mit Operatorfunktion betrachtet:

```
(cout.operator<<(a)).operator<<(b);
```

Gleiches gilt für den Eingabeoperator `operator>>()`.

Für eine Klasse `Mitarbeiter` mit Name und Vorname könnte man sich folgenden Ausgabeoperator vorstellen:
Prototyp in der .h-Datei:

```
class Mitarbeiter {
  public:
    /* ... */
    string getName() { return p_sName; }
    string getVorname() { return p_sVorname; }
  private:
    string p_sName;
    string p_sVorname;
};
// Prototyp des Ausgabeoperators (ausserhalb der class-Def.)
ostream& operator <<(ostream& out, Mitarbeiter& x);
```

Implementierung in der .cpp-Datei:

```
ostream& operator <<(ostream& out, Mitarbeiter& x)
{
  out << x.getName() << ", " << x.getVorname();
  return out;
}
```

- Mit dem Konvertierungsoperator wird eine implizite Konvertierung von einem Benutzerdefinierten Typ zu einem schon definierten Typ, der auch fundamental sein kann, definiert. Im Beispiel wird eine implizite Konvertierung von `X` nach `int` definiert. Man beachte, dass der Typ, zu dem konvertiert wird, Teil des Namens ist und nicht als Rückgabetyp der Konvertierungsfunktion anzugeben ist. Ein Konvertierungsoperator gleicht damit eher einem Konstruktor. Konstruktoren mit einem Parameter werden implizit auch wie ein Konvertierungsoperator benutzt. Konvertierungsoperatoren sollte allerdings mit Vorsicht eingesetzt werden, da schnell Mehrdeutigkeit entstehen können, die der Compiler nicht auflösen kann.

2.5 Typumwandlung

In Ausdrücken und Zuweisungen können fundamentale Datentypen (`char, int, float, double, ...`) beliebig gemischt werden. Um die gewünschte Aktion durchführen zu können, müssen die verschiedenen Werte auf einen gemeinsamen Nenner gebracht werden. Dazu werden die Werte automatisch (wenn möglich) so umgewandelt, dass keine Informationen verloren gehen. Es besteht aber für den Programmierer durch Angabe des Typs in Klammern auch die Möglichkeit, eine Typumwandlung zu erzwingen. Dabei sollten mögliche Datenverluste beachtet werden.

```
int i; float f; double d;

d = i + f;              // i->float, (i+f)->double
i = (int) f + (int) d;  // f->int, d->int, evtl. Datenverlust!
          // Nachkommastellen von f und d werden abgeschnitten
```

In C++ kann derselbe Effekt auch für benutzerdefinierte Typen (Klassen) erreicht werden. Hierbei unterscheidet man zwischen der impliziten (automatischen) und der expliziten (erzwungenen) Umwandlung.

2.5.1 Implizite (automatische) Typumwandlung

Eine automatische Typumwandlung kann auf zwei Arten ermöglicht werden: Durch Definition eines bestimmten Konstruktors oder eines speziellen Umwandlungsoperators.

Konstruktor zur Typumwandlung

Jeder Konstruktor, der mit genau einem Argument aufgerufen werden kann, definiert implizit eine Typumwandlung. Dazu zählen auch Konstruktoren mit mehr als einem Parameter, die Default-Argumente besitzen.

```
class Bruch {
  public:
    Bruch(int x = 0, int y = 1){ p_x = x; p_y = y; }
    const Bruch operator*(const Bruch& b)
    {
      Bruch h(b);
      h.p_x *= p_x;
      h.p_y *= p_y;
      return h;
    }
  private:
    int p_x, p_y;   // Zähler und Nenner
}

int main()
{
  Bruch x(5, 2); Bruch y;
  y = x * 15;       // 15->Bruch(15)
  return 0;
}
```

Zur Umwandlung von `int` nach `Bruch` wird der Konstruktor benutzt, der eine `int`-Zahl als Parameter akzeptiert. Die Umwandlung ist so aber nur in diese Richtung möglich. Der

Ausdruck `15 * x` ist nicht möglich, da für einen fundamentalen Datentyp keine Umwandlungsfunktionen definiert werden können. In diesem Fall müsste man eine Umwandlung selbst durchführen, z. B. in der Form `Bruch(15) * x`.

Manchmal können bei dieser Form der Typumwandlung Probleme auftreten. Man definiert einen entsprechenden Konstruktor, möchte aber keine automatische Typumwandlung festlegen. Dies kann man verhindern, indem man dem Konstruktor das Schlüsselwort `explicit` voranstellt.

Typumwandlung durch Operator

Alternativ zum Konstruktor erfolgt eine automatische Typumwandlung auch durch die Definition eines entsprechenden Umwandlungsoperators. Dieser wird definiert durch das Schlüsselwort `operator`, gefolgt von dem Typ, in den man umwandeln möchte. Dabei muss es sich nicht um einen fundamentalen Datentyp handeln.

```
class Bruch {
  public:
    explicit Bruch(int x = 0, int y = 1){ p_x = x; p_y = y; }
    operator double() const {
      return (double) (p_x / p_y);
    }
  private:
    int p_x, p_y;   // Zähler und Nenner

int main()
{
  Bruch x(5, 2);
  double y;

  y = x * 15;       // x->double
  y = 10 * x;       // hier auch möglich x->double
  return 0;
}
```

Hier wird `x` mittels des Operators von `Bruch` nach `double` umgewandelt. Die Anwendung von `explicit` beim Konstruktor ist notwendig, da die Umwandlung sonst zweideutig ist.

2.5.2 Explizite (erzwungene) Typumwandlung

In der ursprünglichen Spezifikation von C++ waren für die explizite Typumwandlung zwei Alternativen vorgesehen:

Funktionale Notation

Diese ermöglicht eine Typumwandlung durch Aufruf eines entsprechenden Konstruktors. Dabei kann auch eine Liste von Argumenten übergeben werden.

```
Bruch y = x * Bruch(25, 11);
Bruch y = x * Bruch(15);
```

Cast-Notation

Hierbei wird der Datentyp, in den umgewandelt werden soll, in Klammern dem umzuwandelnden Objekt vorangestellt. Die Umwandlung erfolgt entsprechend der automatischen Typumwandlung durch den entsprechenden Konstruktor mit einem Parameter oder durch einen passenden Umwandlungsoperator.

```
Bruch y = x * (Bruch) 10;
double z = (double) Bruch(10, 7);
```

Diese Schreibweise stellt eine Kompatibilität zu C dar und ist im Gegensatz zur funktionalen Notation auch in der Lage, zusammengesetzte Typennamen (Zeiger) zu verwenden.

```
(char*) p_sName
```

Da Typumwandlungen verschiedenen Zwecken dienen können, wurden in der neuesten C++-Spezifikation vier Operatoren zur genauer definierten Typumwandlung eingeführt:

`static_cast`, `dynamic_cast`, `const_cast`, `reinterpret_cast`.

Die gängigsten sind die ersten beiden Operatoren und werden daher hier kurz erläutert.

- `static_cast<typ> (parameter)`

  Der static_cast-Operator konvertiert den `parameter` in einen Typ `typ`. Dies geschieht zwischen verwandten Typen, wie etwa verschiedenen Zeigertypen, zwischen einer Aufzählung und einem integralen Typ oder einem Gleitkommatyp und einem integralen Typ. Man kann auch Zeiger innerhalb einer Klassenhierarchie bewegen, d. h. die Umwandlung von einer Basisklasse in eine abgeleitete Klasse (Downcast) oder umgekehrt (Upcast). Im Gegensatz zum dynamic_cast findet keine Überprüfung zur Laufzeit statt, womit der Programmierer für die Richtigkeit der Umwandlung verantwortlich ist.

```
class B {/*...*/};
class D : public B {/*...*/};

void f(B* pb, D* pd) {
  D* pd2 = static_cast<D*> (pb);
  // unsicher, pb zeigt evtl. nicht auf D sondern auf B

  B* pb2 = static_cast<B*> (pd);  //sicher
}
```

- `dynamic_cast<typ> (parameter);`

  Der dynamic_cast-Operator konvertiert den parameter in einen Typ `typ`. `typ` muss ein Zeiger oder eine Referenz auf eine zuvor definierte Klasse sein oder ein Zeiger auf void. Als `parameter` wird entsprechend ein Zeiger oder ein Objekt einer Klasse übergeben. Der Zweck des dynamic_cast ist die Behandlung von Fällen, in denen die Korrektheit der Umwandlung nicht durch den Compiler ermittelt werden kann. Der Operator schaut dabei zur Laufzeit auf den Typ des zu wandelnden Objekts (`parameter`) und entscheidet, ob die Umwandlung sinnvoll ist. Falls nicht, wird eine 0 zurückgegeben.

```
class B {/*...*/};
class D : public B {/*...*/};
class E : public B {/*...*/};

void f() {
  B* pb1 = new D();
  B* pb2 = new E();

  D* pd1 = dynamic_cast<D*> (pb1);
  // ok, pb1 zeigt auf D

  D* pd2 = dynamic_cast<D*> (pb2);
  // pd2 == 0, da pb2 nicht auf D zeigt
}
```

Einen Sonderfall stellen Variablen dar, die von einem `enum`-Datentyp abgeleitet wurden. Leider erfolgt dann beim Casten keine Typprüfung. `static_cast` ist nicht möglich, da `enum` ein Relikt aus C ist.

Man kann die Typdefinition auch auf eine Klasse beziehen. Dann muss man diese Definition `static` machen, um auf die Werte über Klassenname:: zugreifen zu können.

Beispiel:

```
class Geld {
  public:
    static enum CentMuenzen {C01=1, C02, C05=5, C10=10, C20=20, C50=50};
  private:
    CentMuenzen p_Einwurf;
}
```

Der Zugriff ist dann mit `Geld::CentMuenzen` oder `Geld::C01` möglich.

Zur ausführlichen Beschreibung der Operatoren ziehen Sie bitte weitere Literatur zurate.

2.6 Templates

Oftmals unterscheiden sich Klassen oder Funktionen zur Lösung verschiedenster Probleme nur in den Datentypen, auf denen sie arbeiten, nicht jedoch in ihrer Funktionalität. Zum

Beispiel wird eine Klasse, die eine Warteschlange implementiert, immer Funktionen zum Anhängen und Entfernen von Objekten bereitstellen müssen. Lediglich die Datentypen der Objekte können sich von Fall zu Fall unterscheiden. Genauso ist der Ablauf einer Sortierroutine für unterschiedliche Datentypen immer gleich und beruht nur auf dem Vergleich und dem Vertauschen von zwei Elementen.

In C++ stellen *Templates* ein Hilfsmittel zur Generierung des entsprechenden Codes dar. Ein Template ist ähnlich einer Schablone. Erst durch das Ausfüllen der Schablone wird es zu eigenständigen Code.

Ein Template ist eine Klasse oder Funktion mit (noch) nicht definierten Datentypen. Die eigentlichen Datentypen werden erst bei der Instanziierung eines Objekts einer Templateklasse oder beim Aufruf einer Templatefunktion als Parameter übergeben. Die Vereinbarung erfolgt mit dem Schlüsselwort `template`. Erst bei der Übersetzung wird für jeden benutzten Datentyp der entsprechende Code generiert. Das gesamte Template muss sich in einer Datei befinden und in den Quellcode der benutzenden Funktion eingebunden werden. Bei Templates steht also sowohl Deklaration als auch Methodencodierung in der *.h-Datei.

Syntaktisch beginnt ein Template immer mit einer Zeile der Form:

```
template <class T>
```

Die folgenden Regeln sind bei der Definition der `template-Zeile` zu beachten:

- `<class T>` steht als Platzhalter für einen benutzten Datentyp bzw. eine Klasse. Überall dort, wo im Template ein entsprechender Typ verwendet werden soll, benutzt man diesen Platzhalter. Der Buchstabe T ist lediglich ein (häufig benutztes) Beispiel, es kann auch jeder andere Bezeichner verwendet werden.
- Die `<class T>`-Vereinbarung ist eine Liste. Es können auch mehrere Datentypen benutzt werden. Alle folgenden Vereinbarungen sind erlaubt:

  ```
  <class T>
  <class A, class B>
  <class T1, class T2>
  ```

- Auf den `template`-Kopf folgt die normale Definition der Klasse (Datenstrukturen, Methoden) oder der Funktion (Rückgabetyp, Name, Parameterliste). Hängt der Rückgabetyp und/oder der Typ eines Parameters von der Funktion ab, die aus dem Template generiert wird, so verwendet man `T` als Datentyp. Dabei kann `T` auch für lokale Variablen verwendet werden.

2.6.1 Funktionentemplates

Funktionentemplates werden immer dann sinnvoll eingesetzt, wenn Algorithmen unabhängig vom Datentyp auf bestimmte Eigenschaften (Methoden) dieser Datentypen zurückgeführt werden können. Am einfachsten erlernt man die Verwendung von Funktionentemplates anhand eines Beispiels:

Die Berechnung von Zinsen aus Zinssatz und Kapital ist immer dann möglich, wenn zwischen den verwendeten Datentypen der `*-operator` definiert ist. Für Standardtypen ist die Operation natürlich auch ohne Template möglich, aber für Kapital ist ja auch eine spezielle Klasse (z. B. als BCD-Code oder über verschiedene Währungen) vorstellbar. Das Template sieht dann so aus:

```
// Funktionstemplate zur Verzinsung:
template <class T>
float zinsen(T kapital, float zinssatz)
{
  return (kapital * zinssatz / 100);
}
```

Der erste Parameter der Funktion kann ein beliebiger Datentyp sein (angezeigt durch das `T`) vorausgesetzt die *-Operation zwischen diesem Datentyp und `float` ist definiert und liefert als Ergebnis einen `float`-Wert. Aufgrund dieser Funktionsschablone sind z. B. folgende Aufrufe möglich:

```
int geld1 = 1000;
float geld2 = 1000.0;
float prozent = 0.03;
float ertrag;
ertrag = zinsen(geld1, prozent);  // 1. Parameter ist int
ertrag = zinsen(geld2, prozent);  // 1. Parameter ist float
```

Sinnvoller ist es, wenn diese Funktion auch einen Wert vom selben Typ wie Kapital zurückliefern würde. Das ginge mit folgender Templatedefinition:

```
template <class T>
T zinsen(T kapital, float zinssatz)
{
  T hilf;
  double q = zinssatz / 100.0;
  hilf = kapital * q;
  return hilf;
}
```

Hier muss der `*-operator` zwischen dem verwendeten Datentyp und `double` definiert sein und ein Objekt dieses Typs zurückliefern. Die Hilfsvariable `q` ist erforderlich, da sonst auch der `/-operator` für `T` definiert sein müsste. Die Hilfsvariable `hilf` ist nicht notwendig, sondern soll nur die Definition von Hilfsvariablen des Schablonentyps darstellen.

Die Funktionsschablone kann für alle Variablentypen genutzt werden, für welche die innerhalb der Funktion verwendeten Operatoren definiert sind. Der Compiler erzeugt zur Übersetzungszeit den benötigten Code für alle Funktionsaufrufe. In diesem Fall ist die Funktion also tatsächlich mehrmals vorhanden.

2.6.2 Klassentemplates

Klassentemplates werden immer dann eingesetzt, wenn Datenstrukturen unabhängig vom Typ der gespeicherten Elemente dargestellt werden können und immer die gleichen Eigenschaften und Methoden anbieten. Alle abstrakten Datentypen (Listen, Bäume, Stack etc.) sind typische Beispiele. Für diese gibt es daher auch die bereits vorgefertigten Templates in der STL (s. Kapitel 3.4).

Den Aufbau eines Klassentemplates soll das folgende Beispiel eines Stacks verdeutlichen:

```
// Klassentemplate
template <class T> // Def.zeile eines Templates für Datentyp T
class Stack
{
  private:
    T theElem;         // Speicher für das Datenelement
    Stack* ptNext;     // Zeiger auf den Nachfolger
  public:
    Stack() { ptNext = 0 } // K'tor: unterstes Element des Stacks=0
    Stack(T aElem, Stack *ptS = 0) : theElem(aElem), ptNext(ptS) { }

    Stack* push(T aElem); // Fügt Element oben hinzu
    Stack* pop();         // Löscht oberstes Element
    bool empty() { return (ptNext == 0); }  // Stack leer ?
    T top() { return theElem; }
};

template <class T>        // Templatefunktion
Stack<T>* Stack<T>::push(T aElem)
{
  Stack<T>* p = new Stack<T>(aElem, this);
  return p;
}

template <class T>
Stack<T>* Stack<T>::pop()
{
  if (empty()) return this;
  Stack<T>* p = ptNext;
  delete this;
  return p;
}
```

Der Name einer Klasse, die aus solch einem Template generiert wird, ist der Name des Klassentemplates (hier: `Stack`), gefolgt von den konkreten Datentypen in spitzen Klammern. Folgender Code erzeugt z. B. zwei Stacks, einen zur Verwaltung von Integerwerten und einen zur Verwaltung von Gleitkommawerten:

```
Stack<int> stInt;
Stack<float> stFloat;
```

Hinweis: Man kann die Implementierung der Methoden direkt innerhalb der Klassendefinition durchführen (wie bei `empty` oder `top`) oder wie sonst getrennt als Templatefunktion. Bei der Verwendung des Scope-Operators:: muss dem Compiler mitgeteilt werden, dass es sich lediglich um ein Template und nicht um eine fertige Klasse handelt. Dies geschieht durch das angehängte Platzhalteelement in spitzen Klammern (`<T>`) an den Templatedatentyp (wie z. B. oben in `Stack<T>`).

Template-Parameter dürfen von beliebigem Typ sein, vorausgesetzt die Argumente können vom Compiler ausgewertet werden. Eine Definition für eine Templateklasse eines Vektors könnte also z. B. mit der Vektorgröße als Parameter so aussehen:

```
template <class TElem, int TSize>
class Vector
{
  private:
    TElem array[TSize];
    /*...*/
};
```

Dann wäre folgende Definition erlaubt:

```
Vector<double, 100> ds // Vektor mit 100 double-Werten
```

die Definition

```
int n = 100;
Vector<double, n> ds
```

aber nicht, da `n` nicht konstant ist und die Größe von `n` beim Compilieren nicht feststeht.

2.7 Exception Handling (Ausnahmebehandlung)

Unter Exceptions (Ausnahmen) versteht man besondere Programmsituationen, die zur Laufzeit eines Programms auftreten können, insbesondere die sogenannten Laufzeitfehler. Beispiele sind falsche Eingaben, mathematische Fehler (z. B. Divisionen durch Null), Speicherfehler oder Bereichsüberschreitungen bei Vektoren, aber auch andere Situationen, die eine normale Fortsetzung des Programms verbieten. Diese Situationen können bisher nur synchron, das heißt direkt an der Stelle des Auftretens abgehandelt werden. Beim Entwurf von Klassen oder Bibliotheken ist aber oft an der Stelle, an der die Ausnahmesituation entdeckt wird, nicht klar, wie sie geeignet zu behandeln ist. Dann muss durch Setzen eines Fehlercodes und Rücksprung über mehrere Ebenen eine Fehlerbehandlung vorgesehen werden. C++ stellt mit dem «Exception Handling» ein elegantes Konzept zur asynchronen Fehler- (Ausnahme-) Behandlung bereit: Die Bearbeitung der Ausnahmesituation wird vom Ort des Entdeckens direkt an den Ort des Aufrufs weitergereicht.

Eine Exception ist ein Objekt eines beliebigen Typs, das am Entdeckungsort geworfen (`throw`) und am Behandlungsort durch den zugehörigen Exception Handler gefangen (`catch`) wird. Das Objekt enthält alle für die Behandlung wichtigen Informationen. Meist werden für diesen Zweck eigens definierte Klassen verwendet. Der zu überwachende Code

steht in einem `try`- Block, der ggf. von mehreren Exception Handlern überwacht wird. `throw` und `catch` stehen normalerweise nicht innerhalb derselben Funktion (dann könnte man auch `if-then-else` zur synchronen Fehlerbehandlung benutzen). Zur Verdeutlichung diene zunächst folgendes Beispiel:

Die Stackklasse aus vorigem Kapitel soll bei Aufruf von `top()` auf leerem Stack eine Fehlermeldung liefern.

```
template <class T>
T Stack<T>::top()
{
  if (empty()) {    // Werfen der Exception als char*
    throw "Error: Stack empty.\n";
  }
  return theElem;
}
```

Hier wird als einfachste Form direkt ein C-String (`char*`) geworfen, der die Meldung enthält. Um diese Fehlersituation zu berücksichtigen, muss der Aufruf der Funktion `top()` in einem try- Block stehen, der einen Exception-Handler für `char*` enthält.

```
try {
  // Dieser Block wird von den Exception Handlern überwacht.
  /*...*/
  x = aStack.top();
  /*...*/
}
catch (const char* s) { cout << s; }  // Handler für Typ char*
```

Beim Wurf der Exception wird mit dem Code des entsprechenden Handlers fortgesetzt, hier wird also der Text auf `cout` ausgegeben. Der Code hinter dem `throw`-Statement wird nicht mehr ausgeführt, ebenso wird der entsprechende `try`-Block ohne Bearbeitung des nach dem Aufruf stehenden Codes verlassen. Das geworfene Fehlerelement (hier der String) wird an die Variable des `catch`-Blocks (hier s) übergeben und kann so referenziert werden.

Bei größeren Projekten oder Klassen, die von anderen benutzt werden, sollte man eigene Exception-Klassen (oder sogar eine Klassenhierarchie) einführen, um eine Kollision mit anderen Programmteilen zu vermeiden. Die entsprechenden Exception-Klassen müssen dann Elementvariablen zur Aufnahme aller Daten haben, die die Ausnahme beschreiben. Diese werden beim `throw` durch Aufruf entsprechender Konstruktoren gefüllt. Außerdem empfiehlt sich die Definition einer (polymorphen) Bearbeitungsfunktion (z. B. `behandleException()`), die dann im `catch`-Block aufgerufen werden kann.

Ein Teil einer mathematischen Fehlerhierarchie könnte dann so aussehen:

```
class MathError
{
    // Variable zum Speichern der Fehlerbeschreibung
  protected:
```

```
    char p_szErrString[50];
    // Konstruktor mit Fehlerbeschreibung
  public:
    MathError(const char* s)
    {
      strcpy(p_szErrString, s);
      return;
    }
    // Fehlerbehandlung: Ausgabe der Beschreibung
    virtual void behandleException()
    {
      cout << p_szErrString;
    }
};

class NegWurzel: public MathError
{
  // Variable zum Speichern des (negativen) Arguments
  protected:
    double p_dArg;
  public:
    // Konstruktor mit Fehlerbeschreibung und Argument
    NegWurzel(const char* s, double a) : MathError(s), p_dArg(a)
    {}
    // Fehlerbehandlung: Ausgabe der Beschreibung mit Argument
    virtual void behandleException()
    {
      MathError::behandleException();
      cout << " Argument:" << p_dArg;
    }
};
```

Ein Exception Handler dazu könnte dann die Fehler über eine Referenz zu MathError abfangen und behandeln:

```
try {
    /*...*/
}
catch (MathError& e) {
  // Handler für MathError und alle Ableitungen davon
  e.behandleException();
}
```

Ebenso wären separate catch- Blöcke möglich (z. B. wenn keine polymorphe Behandlung des Fehlers existiert):

```
try {
    /*...*/
```

```
}
catch (NegWurzel& e) {
  // behandle NegWurzel
}
catch (MathError& e) {
  // behandle alle anderen MathError
}
```

Eine Möglichkeit für eine Ausnahme könnte im Programm folgendermaßen geworfen werden:

```
  throw NegWurzel("Negative Wurzel", -9.0);
```

Zusammenfassend kann das Exception Handling in C++ so dargestellt werden:

An der Stelle des Codes, wo die Ausnahmesituation auftritt (meistens in einer Bibliothek oder allgemein benutzten Klasse) wird ein Objekt einer Klasse geworfen, welches das Ausnahmeereignis repräsentiert. Die Syntax dazu lautet:

`throw` <*Ausnahmeobjekt*>;

Um auf dieses Ausnahmeobjekt reagieren zu können, muss der Aufruf der zu überwachenden Methode innerhalb eines `try`-Blockes stehen, der einen zum Ausnahmeobjekt passenden Exception Handler (`catch`-Block) enthält. Die Syntax eines `try-catch`-Blocks lautet:

```
try {
  // Dieser Teil des Codes wird überwacht
  // hier Aufruf der Bibliotheksfunktion
}
catch (<Ausnahmeklasse1>& aExc) {
  // hier werden Ausnahmen der Ausnahmeklasse1
  // und aller Unterklassen bearbeitet
}
[ catch (<Ausnahmeklasse2>& aExc) {
  // hier werden Ausnahmen der Ausnahmeklasse2
  // und aller Unterklassen bearbeitet
} ]
/*...*/
[ catch (...) {
  // hier werden alle übrigen Ausnahmen behandelt
} ]
```

Hinweise:

1. Es können beliebig viele `catch`-Blöcke zu einem `try`-Block angegeben werden. Sie werden in der angegebenen Reihenfolge berücksichtigt. Man beachte, dass auch Unterklassen gefangen werden. Daher müssen Spezialfälle zuerst abgefangen werden.

2. Das konkrete Ausnahmeobjekt kann über `aExc` im catch-Block referenziert werden.

3. `try-catch`-Blöcke können geschachtelt werden und ihrerseits wieder `throw` Anweisungen enthalten. Die Auflösung der `try-catch`-Blöcke erfolgt von innen nach aussen.

4. Ein Ausnahmeobjekt kann innerhalb eines `catch`-Blocks zur weiteren Verarbeitung an höhere `try`-Verschachtelungen weitergeworfen werden. Dies erfolgt durch `throw;` (ohne Objekt).

5. Der `(...)-catch`-Block muss, wenn gewünscht, als letzter angegeben werden. Man beachte, dass dieser Block alle Exceptions (auch vom System ausgelöste) fängt und kein Referenzobjekt zur Verfügung stellt. Dieser Exception-Handler sollte nur sehr vorsichtig benutzt werden. Es empfiehlt sich hier meist, die Ausnahme nach Ausführung der spezifischen Behandlung an höhere (System-) Exception-Handler weiterzuwerfen.

6. Wird eine Exception ohne Überwachung durch einen entsprechenden Exception Handler geworfen, erfolgt eine Systemfehlermeldung mit Programmabbruch.

7. Beim Werfen der Ausnahme wird direkt auf den entsprechenden `catch`-Block verzweigt. Die restlichen Anweisungen der werfenden Funktion und des `try`-Blocks werden nicht mehr ausgeführt.

Das Werfen einer Ausnahme beeinflusst die Schnittstelle einer Funktion, da die aufrufende Funktion einen `try`-Block und entsprechende Exception Handler bereitstellen muss. Daher ist es sinnvoll, die Ausnahmen, die geworfen werden können als Teil der Funktionsdeklaration zu spezifizieren. Die Syntax für eine erweiterte Funktionsdeklaration lautet dann:

<*Rückgabetyp*> <*Funktionsname*> (<*Parameterliste*>) `throw` (<*Ausnahmeliste*>);

also z. B.

```
void fkt(int i) throw (exc1, exc2);
```

Dies garantiert dem Aufrufer, dass die Funktion `fkt` nur Ausnahmen aus den Klassen (Typen) `exc1` und `exc2` und davon abgeleiteten Klassen wirft. Der Versuch, eine andere Ausnahme zu werfen, führt dann zu einem Übersetzungsfehler. Die erweiterte Funktionsdeklaration muss sowohl bei der Definition als auch beim Prototyp benutzt werden. Eine Funktionsdeklaration ohne `throw` erlaubt das Werfen beliebiger Ausnahmen. Eine Funktion, die keine Ausnahmen wirft, kann dies durch den Zusatz `throw()` explizit in die Schnittstelle aufnehmen.

2.8 Namensbereiche

Ein Namensbereich (engl. *namespace*) ist ein Mechanismus zum Gruppieren logisch zusammengehöriger Deklarationen. Diese Technik wird vorwiegend bei größeren Projekten eingesetzt, bei denen durch Namensbereiche repräsentierte Module oft Hunderte von Funktionen, Klassen und Templates beinhalten. In diesem Praktikum werden zwar keine eigenen Namensbereiche definiert, da jedoch die C++ Standardbibliothek diese Technik einsetzt

(alles, was zu ihr gehört, liegt im Namensbereich std), werden sie in diesem Kapitel kurz beschrieben.

Ein Namensbereich ist ein Gültigkeitsbereich mit Namen, und somit einer Klasse sehr ähnlich. Im Prinzip ist eine Klasse ein Namensbereich mit einigen zusätzlichen Sprachmitteln. Folglich treffen die üblichen Regeln für Gültigkeitsbereiche auch für Namensbereiche zu.

Beispiel:

```
#include <iostream>
#include <string>

namespace name1
{
  double f();
  int g(int);
}

namespace name2
{
  void h(std::string);
  int zahl;
}

double name1::f() { /*...*/ }
int name1::g(int i) { /*...*/ }

void name2::h(std::string s)
{
  std::cout << s << name1::f() << " "
            << name1::g(zahl)<< std::endl;
}
```

Anmerkungen:

- Der Qualifizierer `name2` bei der Implementierung von `h` ist notwendig, um festzuhalten, dass die Funktion `h` zu `name2` gehört und nicht eine globalen Funktion ist.
- Die Variable `zahl` braucht nicht qualifiziert zu werden, da sie zum selben Namensbereich (`name2`) gehört wie die Funktion `h` selbst.
- Die Funktionen `f` und `g` gehören zum Namensbereich `name1` und müssen somit qualifiziert werden.
- `string`, `cout` und `endl` gehören zur C++ Standardbibliothek und somit zum Namensbereich `std`.

Der Sinn von Namensbereichen ist an dieser Stelle vielleicht nicht ersichtlich, aber wenn man sich ein großes Projekt mit vielen Hunderten von Klassen und Funktionen vorstellt, wird einem schnell klar, dass durch Namensbereiche Namenskollisionen vermieden werden, die gerade bei großen Projekten nur schwer zu finden sind.

Wird ein Name häufig außerhalb seines Namensbereichs benutzt, wird es schnell lästig und auch unübersichtlich ihn immer wieder zu qualifizieren. Dies kann mit Hilfe der `using`-Deklaration vermieden werden, die ein lokales Synonym erzeugt.Es können auch alle Namen eines Namensbereichs mit der `using`-Direktive verfügbar gemacht werden. Man betrachte für die beiden Varianten von `using` (Direktive bzw. Deklaration) folgendes Beispiel:

```
#include <iostream>
#include <string>

using namespace std;  // using Direktive

namespace name1
{
  double f();
  int g(int);
}

namespace name2
{
  void h(std::string);
  int zahl;
  using name1::f;     // using Deklaration
}

double name1::f() { /*...*/ }
int name1::g(int i) { /*...*/ }

void name2::h(string s)
{
  using name1::g;     // using Deklaration
  cout << s << f() << " " << g(zahl)<< endl;
}
```

Anmerkungen:

- Aufgrund der `using`-Direktive können alle Namen der C++ Standardbibliothek in diesem Beispiel ohne Qualifizierer benutzt werden. Da diese Form der Veröffentlichung von Namen den Mechanismus der Namensbereiche zunichte macht, sollte eine globale `using`-Direktive eher sparsam eingesetzt werden. Vor allem sollte sie wenn möglich in Headerdateien vermieden werden, weil automatisch alle Dateien, die diese Datei einbinden, sie auch beinhalten.

- Es ist meistens eine gute Idee, eine `using`-Deklaration so lokal wie möglich zu halten. Speziell sollte man sich bei `using`-Deklarationen innerhalb einer Namensbereichsdefinition überlegen, ob alle Funktionen einen Bezug zu dem Synonym haben.

- Da im Praktikum nur mit dem Namensraum `std` der Standardbibliothek gearbeitet wird, kann dort zur Vereinfachung von dieser Regel abgewichen werden und die Anweisung

  ```
  using namespace std;
  ```

  in allen .cpp- und .h-Dateien nach den Includes der Standardbibliothek eingesetzt werden.

3 Die C++ Standardbibliothek

In C++ können sämtliche Standard-C-Funktionen benutzt werden. Dazu muss jeweils das entsprechende Headerfile vorher mit

```
#include <file.h>
```

eingebunden werden, z.B. `#include <math.h>` für die Mathematikfunktionen. Die Headerfiles enthalten jeweils die Deklarationen.

Wo es möglich ist, sollten aber in C++ die entsprechenden alternativen Klassenbibliotheken benutzt werden. Diese können durch Einbinden des entsprechenden Headerfiles (ohne .h) bereitgestellt werden:

```
#include <file1>
#include <file2>
#include <file3>
using namespace std;
```

Da alle Standardbibliotheken den Namensraum `std` benutzen, kann durch die Benutzung von `using namespace std;` der Default auf diesen eingestellt werden. Für Ein- und Ausgabe sollen **nicht** die `stdio`-Funktionen wie `putchar(), printf()` etc. benutzt werden, sondern die C++ Klassen-Bibliothek `iostream`.

3.1 Ein- und Ausgabe

Für Ein- und Ausgabe bietet C++ die sogenannten Streams. Dies sind Klassen bzw. vorgefertigte Templates, die diese Funktionen beinhalten. Ein großer Vorteil der Streams ist, dass die Ein- und Ausgabe über die überladenen Operatoren `>>` und `<<` realisiert wird, womit eine übersichtliche Programmierung und eine einfache Erweiterung um benutzerdefinierte Datentypen ermöglicht wird. Im einfachsten Fall werden Sie die Ausgabe auf den Bildschirm und die Eingabe von der Tastatur benötigen. Dazu stellt C++ die Klassen `istream` und `ostream` sowie die Objekte

`cin`	Standardeingabe (Tastatur)
`cout`	Standardausgabe (Bildschirm)
`cerr`	Fehlerausgabe (Bildschirm)

zur Verfügung . Um die Streams zu benutzen, müssen Sie

```
#include <iostream>
```

einbinden. (Beachte: Ohne Extension .h)

3.1.1 Ausgabe

Eine Ausgabe erfolgt durch die Anweisung:

```
cout << object;
```

Mehrere Ausgaben können auch direkt miteinander verbunden werden:

```
cout << object << object << ... << object;
```

Der Operator << ist bereits für alle eingebauten Datentypen definiert, die Ausgabe erfolgt immer entsprechend des jeweiligen Objekts.

Beispiel:

```
double x = 25.391;
int i = 10;
char s[] = "Test";

cout << "Dies ist die Zahl x: " << x << endl;
cout << "Dies ist ein " << s << " mit i = " << i << endl;
```

`endl` steht für neue Zeile, stattdessen kann auch \n in einem String verwendet werden. Die Ausgabe des Beispiels ist:

```
Dies ist die Zahl x: 25.391
Dies ist ein Test mit i = 10
```

3.1.2 Formatierte Ausgabe

Die Stream-Klassen bieten in der Streams-2.0-Implementation folgende Methoden zur Manipulation der Ausgabe:

`width()`	Lesen/Setzen der Feldbreite für Ausgabe
`fill()`	Lesen/Setzen des Füllzeichens für Ausgabe
`flags()`	Lesen/Setzen der Flags für Ausgabe
`setf()`	Setzen von einzelnen Flags
`unsetf()`	Löschen von einzelnen Flags
`precision()`	Nachkommastellen bei Fließkommaausgabe

Das Setzen der Feldbreite hat nur Einfluss auf die nächste Ausgabeoperation!!!

Ohne Angabe eines Parameters liefern `width()`, `fill()`, `flags()` und `precision()` jeweils den aktuellen Wert.

Für die Ausgabeformatierung stehen folgende Flags zu Verfügung:

`ios::skipws`	führende Spaces beim Einlesen ignorieren (default)
`ios::noskipws`	führende Spaces beim Einlesen nicht ignorieren
`ios::left`	Linksbündige Ausgabe
`ios::right`	Rechtsbündige Ausgabe
`ios::internal`	Vorzeichen linksbündig, Wert rechtsbündig
`ios::dec`	Dezimale Ausgabe
`ios::oct`	Oktale Ausgabe
`ios::hex`	Hexadezimale Ausgabe
`ios::showbase`	Ausgabe der Zahlenbasis
`ios::showpoint`	Ausgabe aller Nachkommazahlen, auch wenn 0
`ios::showpos`	Ausgabe von + vor positiven Werten
`ios::uppercase`	„E“ und „X“ statt „e“ und „x“
`ios::scientific`	Darstellung: `d.ddddedd`
`ios::fixed`	Darstellung: `dddd.dd`
`ios::unitbuf`	Ausgabe nach jeder Operation
`ios::stdio`	Ausgabe nach jedem Zeichen

Als Masken für `setf()` stehen außerdem zur Verfügung:

`ios::basefield`	Alle Zahlenbasis-Flags
`ios::adjustfield`	Alle Ausrichtungs-Flags
`ios::floatfield`	Alle Fließkomma-Flags

Beispiele für die Anwendung der Methoden:

```
double x = 19.275;
int i = 125;
cout.width(6);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout << x << endl;
cout.width(10);
cout.precision(5);
cout.setf(ios::fixed, ios::floatfield);
cout << x << endl;
cout.width(10);
cout << i << endl;
cout.width(6);
cout << i << endl;
```

Ausgabe:

```
19.275
  19.27500
       125
   125
```

Um das ganze etwas zu vereinfachen, existieren die sogenannten IO-Manipulatoren, die das Setzen der Ausgabeparameter direkt als Ausgabeelemente über `<<` ermöglichen. Um sie zu verwenden, muss

```
#include <iomanip>
```

eingebunden werden. Die zur Verfügung stehenden IO-Manipulatoren sind:

kill	`setbase()`	Setzen der Zahlenbasis
	`setfill()`	Setzen des Füllzeichens für die Ausgabe
	`setprecision()`	Nachkommastellen bei Fließkommaausgabe
	`setw()`	Setzen der Feldbreite für die Ausgabe
	`setiosflags()`	Setzen der `ios`-Flags
	`resetiosflags()`	Rücksetzen der `ios`-Flags

Die Parameter entsprechen denen der o.g. Methoden. Am Beispiel ist nochmals die obige Ausgabe realisiert:

```
double x = 19.275;
int i = 125;
cout << setw(6) << setprecision(3)
     << setiosflags(ios::fixed) << x << endl;
cout << setw(10) << setprecision(5)
     << setiosflags(ios::fixed) << x << endl;
cout << setw(10) << i << endl;
cout << setw(6) << i << endl;
```

Alle Parameter sind unabhängig voneinander. Das bedeutet zum Beispiel, dass man für einen Wechsel von links- auf rechtsbündige Ausgabe `left` zurücksetzen **und** `right` setzen muss. Ansonsten ist das Resultat nicht vorhersehbar.

3.1.3 Eingabe

Möchte man umgekehrt den Wert einer Variablen von der Tastatur einlesen, so existiert dafür der `>>`-Operator. Dieser funktioniert analog zur Ausgabe:

```
cin >> variable;
```

Die Eingabe muss mit der Taste Enter abgeschlossen werden. Ausgewertet wird die Eingabe für jede Variable soweit, wie die gelesenen Zeichen zum Typ der Variable passen: z.B. endet die Eingabe für `int` bei allen nicht Ziffern, bei `char[]` beim Leerzeichen. Alle Leerzeichen innerhalb der Eingabe werden als Trennzeichen behandelt, soweit nicht das `skipws`-Flag gesetzt ist.

Ein als `enum` definierter Datentyp kann nicht einfach mit dem entsprechenden Operator eingelesen oder ausgegeben werden. Stattdessen muss eine Variable (z.B. ein `int`) eingelesen

und casting oder einer if-Abfrage oder switch dem entsprechenden `enum`-Wert zugewiesen werden. Für die Ausgabe gilt das ebenso.

Beispiel1:

```
enum Farbe{Rot,Blau};
Farbe eF=Rot;
int iF;
cout << "Gib Farbe (1=Rot, 2=Blau, sonst=Rot)" << endl;
cin >> iF;
if (iF == 1)
   eF = Rot;
else if (iF == 2)
   eF = Blau;
cout << "Farbe: " << (int) eF << endl;
```

Ausgegeben wir hier nicht “Rot“ oder “Blau“ sondern 1 oder 2;

Beispiel2:

```
CentMuenzen eG=C50;
int iG;
cout << "Gib Cent-Muenze:" << endl;
cin >> iG;
eG = (CentMuenzen) iG;
cout << "Cent-Muenze: " << (int) eG << endl;
```

3.2 File I/O

C++ unterstützt zwei Arten von Files: sequentielle Files und Files mit wahlfreiem Zugriff. In diesem Praktikum werden nur erstere behandelt.

Es gibt lediglich drei mögliche Aktionen, die man auf einem sequentiellen File ausführen kann: Daten aus dem File einlesen, Daten in ein neues File schreiben und Daten an ein existierendes File anhängen. Bei einem sequentiellen File können die Daten nicht umgeordnet werden, ohne das ganze File zu löschen.

3.2.1 Schreiben von Daten in ein File

Die Ausgabe von Daten in ein File erfolgt ganz analog zur Ausgabe von Daten auf dem Bildschirm. Betrachten Sie zunächst folgendes Beispielprogramm:

```
#include <fstream>
#include <iostream>

using namespace std;
```

```
void main(void)
{
  // Testvariablen
  char name[] = "Arthur";
  int semester = 42;
  double schnitt = 4.2;

  // Öffnen des Files für die Ausgabe
  ofstream outfile("Ausgabedatei.dat");

  // Schreibe die Daten in das File
  outfile << "Name: " << name << endl;
  outfile << "Semester: " << semester << endl;
  outfile << "Notenschnitt: " << schnitt << endl;
}
```

Nach der Initialisierung einiger Testvariablen erzeugt das Programm ein Objekt mit Namen `outfile`, welches den Datentyp `ofstream` (für *output file stream*) hat. `ofstream` ist eine Unterklasse von `ostream`, der allgemeinen Klasse, für die der `<<`-Operator definiert wird. `cout` ist ein Objekt von `ostream`. Dieser Typ ist im Headerfile `fstream` deklariert, welches daher zu Beginn eingebunden werden muss. Als Parameter erhält der Konstruktor eines `ofstream`-Objektes den Namen des zu erzeugenden Files (hier: *Ausgabedatei.dat*). Das Objekt `outfile` repräsentiert nun das File, in das die Ausgabe erfolgen soll. Selbstverständlich können Sie auch einen beliebigen anderen Namen für das Objekt verwenden. Ganz analog zur Ausgabe auf dem Bildschirm können die Daten nun mit Hilfe des Ausgabeoperators `<<` in das File geschrieben werden.

Beachten Sie, dass das File weder explizit geöffnet noch geschlossen werden muss. Der Konstruktor öffnet das File beim Erzeugen eines `ofstream`-Objektes automatisch und der Destruktor schließt das File wieder, wenn der Gültigkeitsbereich des Objektes verlassen wird.

Das Programm erzeugt bei seiner Ausführung die Datei *Ausgabedatei.dat* mit folgendem Inhalt:

```
Name: Arthur
Semester: 42
Notenschnitt: 4.2
```

3.2.2 Lesen von Daten aus einem File

indexDateieingabe Genauso einfach wie das Schreiben in ein File ist mit C++ auch das Lesen aus einem File. Das folgende Programm liest die Daten aus der Datei *Ausgabedatei.dat* wieder ein und gibt sie auf dem Bildschirm aus:

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

```
void main(void)
{
  // Testvariablen
  char name[7];
  int semester;
  double schnitt;
  char infoString[3][255];
  // Öffnen des Files zum Lesen
  ifstream infile("Ausgabedatei.dat");
  // Lese die Daten aus dem File
  infile >> infoString[0] >> name;
         // "Name:" -> infoString[0], "Arthur" -> name
  infile >> infoString[1] >> semester;
         // "Semester:" -> infoString[1], 42 -> semester
  infile >> infoString[2] >> schnitt;
         // "Notenschnitt:" -> infoString[2], 4.2 -> schnitt
  // Schreiben auf den Bildschirm
  cout << infoString[0] << name << endl;
  cout << infoString[1] << semester << endl;
  cout << infoString[2] << schnitt << endl;
}
```

Nach der Definition der Variablen, welche die einzulesenden Daten aufnehmen sollen, wird ein Objekt vom Typ `ifstream` (für *input file stream*) erzeugt, welches automatisch das gewünschte File zum Lesen öffnet. `ifstream` ist eine Unterklasse von `istream`, der allgemeinen Klasse, für die der `>>`-Operator definiert wird. `cin` ist Objekt von `istream`. Der überladene Eingabeoperator `>>` wird dann verwendet um die Daten aus dem File in die Variablen einzulesen. Zu beachten ist hierbei, dass Leerzeichen, Tabulatoren und Zeilenumbrüche als Trennzeichen dienen um die einzelnen Daten im File voneinander abzugrenzen. Ein in der Datei gespeicherter String darf also keine Leerzeichen enthalten, da er beim Lesen sonst als zwei (oder mehr) Strings interpretiert wird. Dieses Verhalten kann durch spezielle Funktionen geändert werden. Auch ist darauf zu achten, dass die Datentypen der Variablen mit den Datentypen der gelesenen Werte übereinstimmen, da es ansonsten zu einem Lesefehler kommt. Das Programm gibt folgendes auf dem Bildschirm aus:

```
Name:Arthur
Semester:42
Notenschnitt:4.2
```

Ist die Menge der einzulesenden Daten nicht von vornherein bekannt, so muss es eine Möglichkeit geben zu erkennen, wann das Dateiende erreicht ist. Dies kann mittels der Funktion `eof()` getestet werden. Folgendes Programmfragment liest solange `int`-Werte aus einer Datei ein, bis das Dateiende erreicht ist:

```
// Variable zum Einlesen
int zahl;
// Öffnen der Datei
```

```
ifstream fin("Testdatei.dat");
// Lies, bis Dateiende erreicht
fin >> zahl;
while (!fin.eof())
{
  cout << "Gelesene Zahl : " << zahl << endl;
  fin >> zahl; // Versuch, nächste Zahl zu lesen
}
```

Beachten Sie, dass das Dateiende nicht beim Lesen der letzten Zeile, sondern erst beim darauffolgenden (erfolglosen) Versuch, eine Zeile zu lesen, erkannt wird. Bevor auf `eof()` getestet werden kann, muss also eingelesen werden.

Zur Verdeutlichung der Zusammenhänge sei in Abbildung 3.1 nochmal die Klassenstruktur der IO dargestellt.

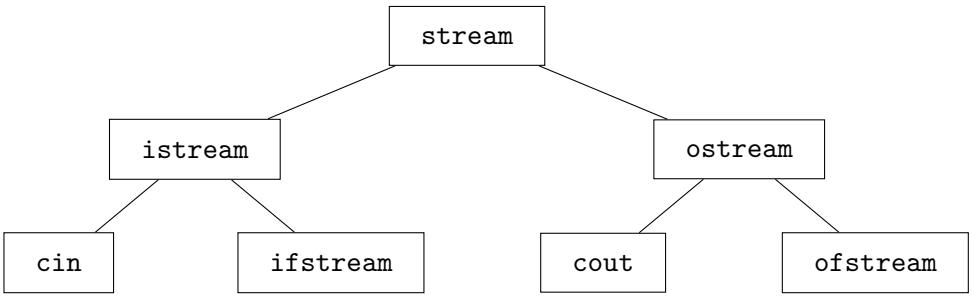


Abbildung 3.1: Klassenhierarchie stream

3.2.3 Abfangen von Fehlern

Beim Lesen von Files und Schreiben in Files können Fehler auftreten, die abgefangen werden müssen. Beispielsweise könnte man versuchen, aus einem nicht existierenden File zu lesen, das USB-Laufwerk könnte nicht verfügbar oder man für die Datei nicht genügend Rechte haben. Ein `ifstream`- oder `ofstream`-Objekt besitzt vier member functions die zum Abfangen solcher Fehler verwendet werden können: `good()`, `eof()`, `fail()` und `bad()`.

Wir konzentrieren uns hier auf die Verwendung von `good()`. Diese Memberfunktion liefert `true` zurück, wenn kein Fehler auftrat. Sobald es bei einer Ein-/Ausgabefunktion (Öffnen, Lesen, Schreiben) zu einem Fehler kommt, liefert die Funktion `false` zurück.

Die folgende Funktion `liesFile` liest 100 int-Werte aus einer Datei ein. Konnte die Datei mit dem als Parameter übergebenen Namen nicht geöffnet werden oder kam es während des Lesens zu einem Fehler, so liefert die Funktion `false` zurück, ansonsten `true`.

```
bool liesFile(const char* dateiname, int zahl[]) {
  // Öffne die Datei zum Lesen
  // (weitere Flags siehe Datei-Flags)
  ifstream fin(dateiname);
```

```
    // Prüfe, ob Öffnen erfolgreich war
    if (!fin.good())
      return false;

    // Lies die Werte ein
    for (int i = 0; i < 100; i++)
      fin >> zahl[i];
    // Liefere Fehlercode zurück
    // (true, falls alles OK war, sonst false)
    return fin.good();
  }
```

3.2.4 Datei-Flags

Für den genauen Modus der Dateibearbeitung existieren einige Flags, die in der Klasse `ios` wie folgt definiert werden:

`ios::in`	Lesen (Default bei `ifstream`)
`ios::out`	Schreiben (Default bei `ofstream`)
`ios::app`	Anhängen
`ios::ate`	ans Ende Positionieren (`"at end"`)
`ios::trunc`	alten Dateiinhalt löschen
`ios::nocreate`	Datei muss existieren
`ios::noreplace`	Datei darf nicht existieren

Die Flags können mit dem `|`-Operator verknüpft werden. Übergeben werden sie dann dem Konstruktor als optionales zweites Argument. Die folgende Anweisung öffnet z.B. eine Datei, die bereits existieren muss, zum Schreiben und sorgt dafür, dass der geschriebene Text an das Ende der Datei angehängt wird:

```
fstream datei("xyz.out", ios::out | ios::app | ios::nocreate);
```

3.2.5 Fehlerzustände, Stream-Status

Für den prinzipiellen Zustand eines Streams werden in der Klasse ios Flags definiert:

`ios::goodbit`	alles in Ordnung, kein (anderes) Bit gesetzt
`ios::eofbit`	End-Of-File
`ios::failbit`	Fehler: letzter Vorgang nicht korrekt abgeschlossen
`ios::badbit`	fataler Fehler: Zustand nicht definiert

Bei `ios::goodbit` von einem Flag zu reden, ist etwas verwirrend, da es typischerweise den Wert 0 besitzt und damit automatisch anzeigt, ob ein anderes Flag gesetzt ist.

Der Unterschied zwischen `ios::failbit` und `ios::badbit` ist klein, aber durchaus von Interesse:

- `ios::failbit` wird i.A. gesetzt, wenn ein Vorgang nicht korrekt durchgeführt werden konnte, der Stream aber prinzipiell in Ordnung ist. Typisch dafür sind Formatfehler beim Einlesen. Wenn z.B. ein Integer eingelesen werden soll, das nächste Zeichen aber ein Buchstabe ist, wird dieses Flag gesetzt.
- `ios::badbit` wird i.A. gesetzt, wenn der Stream prinzipiell nicht mehr in Ordnung ist oder Zeichen verlorengegangen sind. Dieses Flag wird z.B. beim Positionieren vor einen Dateianfang gesetzt.

3.3 Strings

Mit einem String der Standardbibliothek von C++ wird die Bearbeitung von Zeichenketten gegenüber der „klassischen“ Form über `char`-Felder wesentlich erleichtert. Voraussetzung für die Benutzung ist die Einbindung der Bibliothek `<string>` durch

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

Es können dann Elemente der neuen Klasse `string` mit verschiedenen Konstruktoren erstellt werden:

1. `string myString`

   erstellt einen leeren String.

2. `string myString(10, 'A')`

   erstellt einen String mit 10 Zeichen und dem Inhalt `"AAAAAAAAAA"`. Üblicher als eine Initialisierung mit `'A'` ist natürlich eine mit Blank.

3. `string myString("Dies ist ein String")`

   erstellt einen String mit 19 Zeichen und dem Inhalt *"Dies ist ein String"*.

Die Ein-/Ausgabe der Strings erfolgt über die überlagerten `<<` bzw. `>>`-Operatoren. Beim Eingabeoperator wird dabei ggf. der String auf die benötigte Größe verlängert.

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

void main()
{
  string s1(10, 'A');
  string s2("Dies ist ein String.");
```

```
  string s3;

  cout << s1 << endl;
  cout << s2 << endl;
  cout << "Geben Sie einen String ein:" << endl;
  cin >> s3;
  cout << s3 << endl;
}
```

Die Ausgabe dieses Programms ist:

```
AAAAAAAAAA
Dies ist ein String.
Geben Sie einen String ein:
(Eingabe:abcd) abcd
```

Hinweis: Sollten bei Benutzung einer alten Version des Visual Studios der Einsatz der Bibliothek `<string>` sowie im weiterer aller STL-Klassen Fehlermeldungen mit der Nummer 4786 erzeugen, können diese mit `#pragma warning(disable:4786)` unterbunden werden. Damit verhindern Sie die Ausgabe von Warnungen bei zulangen Variablennamen. In diesen Bibliotheken werden zur Vermeidung von Überschneidungen lange Variablennamen benutzt.

Auch die Vergleichsoperatoren, sowie der `+` - und der `+=` -Operator sind in intuitiver Weise für Strings überlagert. Die Vergleichsoperatoren liefern das Ergebnis des zeichenweisen Vergleichs anhand der zugrundeliegenden Codierung (`ASCII`), die `+` -Operatoren dienen zur Aneinanderreihung (Konkatenation) zweier Strings.

Beispiel:

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

void main()
{
  string s1("ABC");
  string s2;
  string s3;

  s2 = "AB";
  s3 = s1 + s2;
  cout << s3 << endl << "Maximum: ";
  if (s1 > s2)
    cout << s1;
  else
    cout << s2;
}
```

Die Ausgabe dieses Programms ist:

```
ABCAB
Maximum: ABC
```

Der Zugriff auf die einzelnen Zeichen eines Strings erfolgt mit dem `[ ]`-Operator, wobei auch hier mit dem Index 0 auf das erste Zeichen zugegriffen wird. Im Gegensatz zu den C-Strings wird der String nicht durch `\0` beendet, sondern das Ende wird durch die aktuelle Länge des Strings bestimmt. Die Länge kann durch die Methode `length()` bestimmt werden.

Beispiel:

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

void main()
{
  string s1("ABC");
  int i;

  for (i = 0; i < s1.length(); i++)
    cout << s1[i];
}
```

Neben diesem direkten Zugriff auf die einzelnen Elemente kann auch sequentiell auf die Zeichen eines Strings zugegriffen werden. Dies geschieht über die Inkrement-/ Dekrementoperatoren `++` und `--` auf einen Iterator auf die Elemente von String. Einen entsprechenden Iterator deklariert man durch `string::iterator`. Spezielle Methoden (`begin()` und `end()`) liefern jeweils Iteratoren auf den Anfang und hinter das Ende des Strings. Weitere Informationen zu Iteratoren finden Sie im Kapitel 3.4.2. Das folgende Beispiel erzeugt die Ausgabe:

```
A B C D
```

Beispiel:

```
#include <string>
#include <iostream>

using namespace std;

void main()
{
  string s1("ABCD");
  string::iterator it;

  for (it = s1.begin(); it < s1.end(); it++)
    cout << *it << ' ';
```

```
}
```

Für die komfortable Bearbeitung von Stringobjekten stehen diverse Methoden zur Verfügung. Die in Tabelle 3.1 aufgeführte Auswahl ist nicht vollständig und für einige Methoden gibt es neben den dargestellten auch noch weitere Aufrufvarianten mit anderen Parametertypen (siehe dazu die Literatur oder Onlinehilfe).

Methode	Funktionalität
`lenght(), size()`	Liefert die aktuelle Länge des Strings
`empty()`	Liefert `true`, falls leerer String
`insert(pos, str)`	Fügt `str` vor `pos` im String ein
`erase(pos, anzahl)`	Löscht im String ab `pos` `anzahl` Zeichen
`replace(pos, anzahl, str)`	Ersetzt ab `pos` bis zu `anzahl` Zeichen durch die Zeichen von `str`
`find(str, pos)`	Liefert die Position des **ersten** Auftretens von `str` im String ab `pos` (-1 falls nicht vorhanden)
`rfind(str, pos)`	Liefert die Position des **letzten** Auftretens von `str` im String bei oder vor `pos` beginnend (-1 falls nicht vorhanden)
`find_first_of(str, pos)`	Liefert die Position des **ersten** Auftretens eines Zeichens von `str` im String ab `pos` (-1 falls nicht vorhanden)
`find_last_of(str, pos)`	Liefert die Position des **letzten** Auftretens eines Zeichens von `str` im String bei oder vor `pos` (-1 falls nicht vorhanden)
`find_first_not_of(str, pos)` `find_last_not_of(str, pos)`	Liefert analog die Position des Zeichens im String, das nicht in `str` vorkommt (-1 falls alle Zeichen im String in `str` vorkommen)
`substr(pos, laenge)`	Liefert einen Substring ab `pos` im String mit der angegebenen `laenge`

`compare(str)`
`compare(pos, anz, str)`
`compare(pos, anz, str, pos1, anz1)`
	Vergleicht den String zeichenweise mit `str` und liefert als Ergebnis:
	-1 falls `str` < aktuellem String
	0 falls `str` = aktuellem String
	1 falls `str` > aktuellem String
	Durch die übrigen Parameter kann der Vergleich auf einen Teilstring des aktuellen Strings (`pos, anz`) bzw. des Vergleichstrings (`pos1, anz1`) eingeschränkt werden.

Tabelle 3.1: Bearbeitungsfunktionen für Strings

Es ist zu beachten, dass Stringobjekte und C-Strings nicht verwechselt oder gemischt werden dürfen, auch wenn für die meisten obigen Stringmethoden auch entsprechende überlagerte Methoden für C-Strings als Parameter existieren. Für Funktionen, die C-Strings erwarten (z.B. Funktionen der Windowsbibliothek) existieren keine entsprechenden Aufrufe für Stringobjekte. Allerdings können sie in beide Richtungen umgewandelt werden:

- C-Strings in Stringobjekte durch Aufruf des entsprechenden Konstruktors
- Stringobjekte in C-Strings durch die Methode `c_str()`

3.3.1 String-Streams

Ein Stream kann einem String zugeordnet werden. Damit kann man unter Verwendung der Formatierungsmöglichkeiten aus einem String lesen (wie z.B. von der Standardeingabe `cin`) oder in einen String schreiben (wie auf die Standardausgabe `cout`). Solche Streams werden String-Streams (Klasse `stringstream`) genannt. Sie werden in `<sstream>` definiert.

Wie bei den Stream-Klassen für Dateien gibt es analog Stream-Klassen für Strings:

- `istringstream` - für Strings, aus denen gelsen wird (input string stream)
- `ostringstream` - für Strings, in die geschrieben wird (output string stream)
- `stringstream` - für Strings, die zum Lesen und Schreiben verwendet werden

Die Klassen erben von `iostream` und übernehmen daher auch den Funktionsumfang der Oberklasse, wie z.B. die Ein-/Ausgabeoperatoren. Weiterhin stellt `stringstream` folgende Funktionen zur Verfügung:

- `str()` - liefert den Ausgabestrom als String zurück
- `rdbuf()` - liefert die Adresse des internen String-Buffers

Beispiel:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
  string namen[3];
  for (int i = 0; i < 3; i++) {
    stringstream s;          // Beachte: stringstream lokal
    // Ausgabe "Objekt" + Nummer auf Stringstream, z.B. "Objekt1"
    s << "Objekt" << i + 1;
    namen[i] = s.str();
  }

  for (int i = 0; i < 3; i++)      // Kontrollausgabe
    cout << namen[i] << endl;
}
```

Die Ausgabe dieses Programms ist:

```
Objekt1
Objekt2
Objekt3
```

3.4 STL (Standard Template Library)

Die „Standard Template Library“ (STL) ist eine C++-Bibliothek, die ursprünglich im HP-Entwicklungslabor in Palo Alto entwickelt wurde, mittlerweile aber von fast allen C++-Entwicklungsumgebungen angeboten wird und 1994 in den erweiterten C++-Standard aufgenommen wurde. Sie ermöglicht dem Programmierer, einfachen, effizienten und wiederverwendbaren Code auf der Basis einer intensiven Nutzung von Templates (bzw. daraus generierten Klassen) zu erstellen. Für die wichtigsten Datenstrukturen (Vektoren, Verzeichnisse, Listen, Bäume, ...) und allgemeinen Algorithmen (Suchen, Sortieren, ...) werden entsprechende Implementierungen angeboten. Im Folgenden sollen einige STL-Elemente vorgestellt werden. Diese Beschreibung ist jedoch nicht vollständig und es sei hier auf die Literatur- bzw. Online-Linkliste hingewiesen.

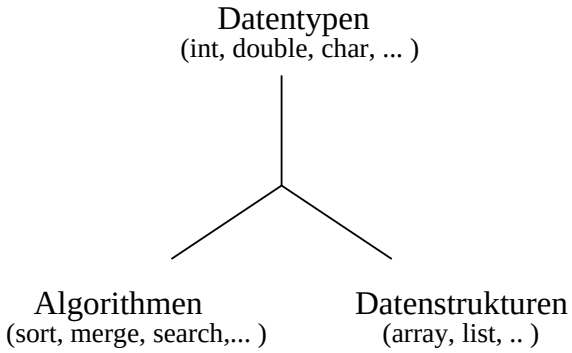


Abbildung 3.2: Softwarekomponenten

Die Idee, die hinter STL steckt, soll an folgender Überlegung deutlich werden. Jede Software setzt sich aus verschiedenen Komponenten des in Abbildung 3.2 dargestellten dreidimensionalen Raumes zusammen. Damit müssen auch entsprechend viele verschiedene Codevarianten entworfen werden: z.B. je ein Array für `int`, für `double`, für `char` usw., ebenfalls je eine Liste für alle Datentypen, dann ein Sortieralgorithmus für `int`-Arrays, für `int`-Listen, für `double`-Arrays usw. Dies lässt sich vereinfachen, in dem man zunächst Datenstrukturen entwickelt, die beliebige Datentypen enthalten können und dann auf diesen Datenstrukturen Algorithmen, die wieder mit beliebigen Datenstrukturen arbeiten, die bestimmte grundlegende Funktionen verstehen. Diese komponentenbasierte Softwarebibliothek ist die Idee von STL.

STL kennt folgende Komponentenarten:

Container	Klasse zur Beschreibung einer Datenstruktur mit Methoden zum Aufnehmen, Löschen und Verwalten von Objekten (z.B. doppeltverkettete Listen)
Iteratoren	Abstrahierter Zeigerzugriff auf Behälter, um allgemeine Algorithmen entwerfen zu können (z.B. <- oder ++ -Operator)
Algorithmen	Funktionsvorschrift, die auf verschiedenen Datenstrukturen arbeiten kann (z.B. sortieren)
Funktionsobjekte	Funktionen, die in Objekte gekapselt werden

3.4.1 Container

Container sind Klassen zur Aufnahme von Objekten beliebigen Typs mit Methoden zum Einfügen, Löschen und Verwalten dieser Objekte. Verschiedene Container unterscheiden sich in der Art und Weise, wie die Objekte zueinander arrangiert sind, ob die Elemente sortiert sind und wie der Zugriff auf die einzelnen Elemente erfolgt. Entsprechend unterscheidet die STL *Sequence Container* und *Associative Container*.

Sequence Container ist die Organisation einer endlichen Menge von Objekten (alle desselben Typs) in einer strikt linearen Ordnung. Der Zugriff auf die Elemente ist nur von einem Element zum nächsten bzw. vorigen, bzw. über die genaue Position eines Elementes möglich. STL stellt dabei die Typen Feld (`vector`), verkettete Liste (`list`), und Schlangen (`deque`) zur Verfügung.

Zu den gerade beschriebenen Containern gibt es noch sogenannte *Adapter*. Diese Container-Adapter stellen zu einem Basis-Container eine eingeschränkte Schnittstelle zur Verfügung. Dabei handelt es sich um den `stack` und die `queue` bzw. `priority_queue`.

Associative Container bieten im Gegensatz dazu (zusätzlich) die Möglichkeit des schnellen Zugriffs auf einzelne Elemente über einen Schlüssel. STL kennt dabei die Typen `set` und `map`, deren Elemente einen eindeutigen Schlüssel besitzen sowie `multiset` und `multimap`, bei denen auch mehrere Elemente denselben Schlüssel besitzen dürfen. Set (Multiset) und Map (Multimap) unterscheiden sich dadurch, dass bei „Set“ der Schlüssel Bestandteil der Daten ist, während bei „Map“ Daten und Schlüssel unabhängig voneinander sind.

In Tabelle 3.2 finden Sie nochmal eine Zusammenfassung aller Container der STL. Im Praktikum werden wir nur die Containertypen `vector, list` und `map` verwenden, auf die wir nun näher eingehen wollen.

vector

Um in C++ (ohne STL) mehrere gleichartige Objekte zusammenzufassen, definieren wir ein entsprechendes Feld, z.B. für 5 `int`-Zahlen:

```
int feld[max_size];
```

Wenn wir dies codieren, müssen wir die maximale Anzahl der Elemente kennen (spätestens beim dynamischen Allokieren des Speichers). Mit Hilfe der STL können wir statt

Container	Beschreibung
Sequence Container	
`vector`	lineare, benachbarte Speicherung; schnelles Einfügen nur am Ende
`deque`	lineare, nicht benachbarte Speicherung; schnelles Einfügen an Außenstellen
`list`	beidseitig verkettete Liste; schnelles Einfügen an beliebigen Stellen
Associative Container	
`multiset`	Datenmenge, schneller assoziativer Zugriff, Duplikate zulässig
`set`	wie `multiset`, jedoch keine Duplikate zulässig
`multimap`	Menge von Schlüssel/Wert-Paaren, schneller Zugriff über Schlüssel, Duplikate für Schlüssel zulässig
`map`	wie `multimap`, jedoch keine Duplikate für Schlüssel zulässig
Adapter	
`stack`	strikte FILO-Datenstruktur (Kellerspeicher)
`queue`	strikte FIFO-Datenstruktur (Warteschlange)
`priority_queue`	queue, die Elemente mit Priorität speichert und somit die Reihenfolge des Zugriffs bestimmt

Tabelle 3.2: Containerklassen in der Übersicht

eines (festdimensionierten) Feldes einen entsprechenden Vektor definieren. Definiert ist der Vector in `<vector>` und wird dort wie folgt deklariert:

```
template<class T> class vector {};
```

Wie oben schon beschrieben handelt es sich hierbei um einen sequenziellen Container, dessen Größe sich nach Bedarf anpassen lässt:

```
vector<int> feld;
```

oder um eine Anfangsgröße des Vektors festzulegen:

```
vector<int> feld(5);
```

Auch ein Initialwert kann bei der Definition direkt mitgegeben werden, z.B. 3 `float`-Elemente mit dem Wert `-1`:

```
vector<float> feld(3, -1.0);
```

Auf die einzelnen Elemente des Vektors kann mit dem `[]`-Operator zugegriffen werden, beginnend für das erste Element mit 0 (Beispiel: `feld[4]`). Der Zugriff auf nicht definierte Elemente führt auch hier (wie bei C++-Feldern) zu unvorhersehbaren Ergebnissen. Wo nötig, sollte also der Index zuvor auf Gültigkeit getestet werden bzw. eine eigene (sichere) Vektorklasse implementiert werden.

Tabelle 3.3 enthält einige grundlegende Funktionen des Vectors. Für eine vollständige Übersicht sei auch hier wieder auf die empfohlene Literatur verwiesen.

Methoden	Funktionalität
`size_type size() const`	Liefert die aktuelle Anzahl der Elemente
`bool empty() const`	Liefert `true`, falls Vektor leer
`void push_back(const T& val)`	Fügt `val` am Ende des Vektors ein
`void pop_back()`	Löscht das letzte Element im Vektor
`void swap(vector<T>& vec)`	Tauscht den Inhalt mit dem von `vec`

Tabelle 3.3: Einige Funktionen von vector

Beispiel:

```
#include <vector>
#include <iostream>

using namespace std;

void main() {
  vector<int> v1(3);
  int i;

  for (i = 0; i < 3; i++)
      v1[i] = i;
  for (i =0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;

  v1.pop_back();
  for (i = 0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;

  v1.push_back(5);
  v1.push_back(6);
  for (i = 0; i < v1.size(); i++)
    cout << v1[i] << ' ' ;
  cout << endl;
  return;
}
```

Die zugehörige Ausgabe ist dann:

```
0 1 2
0 1
0 1 5 6
```

Es ist möglich, einen Vektor direkt einem anderen zuzuweisen (=-Operator) und zwei Vektoren auf Gleichheit zu testen (==-Operator). Für Gleichheit müssen die beiden Vektoren gleich groß sein und alle Elemente übereinstimmen.

Methoden	Funktionalität
`size_type size() const`	Liefert die aktuelle Anzahl der Elemente
`bool empty() const`	Liefert true, falls Liste leer
`iterator begin()`	Liefert Iterator, der auf das erste Element zeigt
`iterator end()`	Liefert Iterator, der hinter das letzte Element zeigt
`void push_back(const T& val)`	Fügt val am Ende der Liste ein
`void push_front(const T& val)`	Fügt val am Anfang der Liste ein
`void pop_back()`	Löscht das letzte Element
`iterator insert(iterator pos, const T& val)`	Fügt val an Position pos ein und gibt einen Iterator auf das neue Element zurück.
`void erase(iterator pos)`	Löscht das Element an Position pos.

Tabelle 3.4: Einige Funktionen von list

list

Eine Liste ist ein sequentieller Container, der sich optimal für das Einfügen und Löschen an beliebigen Stellen seiner Struktur eignet. Er ist in `<list>` definiert und weist folgende Deklaration auf:

```
template<class T> class list {};
```

Betrachten Sie Abbildung 3.3. Anders als bei einem Vector bietet eine Liste keinen Zugriff auf Indexbasis. Die Funktion `begin()` liefert einen Iterator auf das 1. Element der Liste. Dieser Iterator kann mit `++` und `--` vor- und rückwärts bewegt werden. Das Ende der Liste kann durch die Funktion `end()` abgefragt werden. Die Beschreibung für Iteratoren finden Sie in Kapitel 3.4.2.

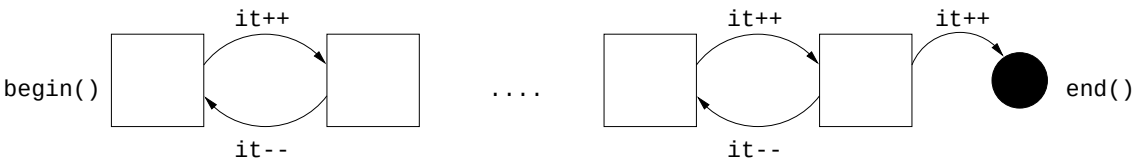


Abbildung 3.3: Durchlaufen einer Liste

Die Definition einer Liste erfolgt z.B. für eine Liste mit `string`-Elementen durch:

```
list<string> meineListe;
```

Eine Liste besitzt dieselben Memberfunktionen zum Erweitern, Löschen und Abfragen der Größe wie ein Vektor. Um Elemente in die Liste einzufügen oder zu löschen, stehen zusätzlich die Funktionen `insert()` und `erase()` zur Verfügung (s. Tabelle 3.4). Diese Funktionen existieren zwar auch für Vektoren, führen dort aber zu einer Reallokation des gesamten Vektors und machen alle Iteratoren, die sich auf Elemente hinter der Einfüge-/Löschposition beziehen, ungültig. Sie sind also dort nur in Sonderfällen sinnvoll einsetzbar.

Zum umgekehrten Durchlaufen der Liste kann man einen reverse-iterator mit den zugehörigen Funktionen `rbegin()` (=letztes Element) und `rend()` (**vor** dem 1. Element) benutzen. Entsprechend gibt es auch die umgekehrten push-/pop-Funktionen: `push_front(val)` und `pop_front()`. Mit der Funktion `reverse()` kann die Reihenfolge der Elemente in der Liste umgekehrt werden.

Es ist möglich, eine Liste oder einen Vektor direkt einer anderen Liste zuzuweisen (=-Operator) und zwei Listen auf Gleichheit zu testen (==-Operator). Für Gleichheit müssen die beiden Listen gleich groß sein und alle (in der richtigen Reihenfolge) Elemente übereinstimmen.

Beispiel:

```
#include <iostream>
#include <iomanip>
#include <list>

using namespace std;
typedef list<int> ListInt;

void main()
{
  ListInt v;
  ListInt::iterator itL;
  ListInt::reverse_iterator itR;

  v.push_back(7);            // v: 7
  v.insert(v.begin(), 3);    // v: 3 7
  v.insert(v.end(), 8);      // v: 3 7 8
  itL = v.begin();
  itL++;
  v.erase(itL);              // v: 3 8

  for (int i = 0; i < 3; i++)
    v.insert(v.begin(), i); // v: 2 1 0 3 8

  //Rückwärtsausgabe
  for (itR = v.rbegin(); itR != v.rend(); itR++)
    cout << setw(2) << *itR;

  return;
}
```

Ausgabe dieses Programms ist: `8 3 0 1 2`

Statt ein einzelnes Element in die Liste einzufügen, kann man auch alle Elemente des Intervalls [first, last) einfügen. Der notwendige Funktionsaufruf ist dann:

```
void insert(iterator pos, const T* first, const T* last)
```

Der durch `first` und `last` bestimmte Teil einer Liste (oder eines Vektors bzw. Feldes) wird an der Position `pos` eingefügt.

Beispiel:

```
#include <iostream>
#include <list>
#include <string>

using namespace std;
typedef list<string> ListString;

void main()
{
  ListString aSList1, aSList2;
  ListString::iterator itL;

  aSList1.push_back("gelb");
  aSList1.push_back("blau");
  aSList1.push_back("rot");

  aSList2.push_front("weiss");
  aSList2.push_front("lila");
  aSList2.push_front("schwarz");

  aSList1.insert(aSList1.begin(), aSList2.begin(), aSList2.end());

  for (itL = aSList1.begin(); itL != aSList1.end(); itL++)
    cout << *itL << ' ';

  return;
}
```

Ausgabe dieses Programms: `schwarz lila weiss gelb blau rot`

map

Maps sind assoziative Container zur Speicherung von Schlüssel/Werte-Paare. Dies ermöglicht die sortierte Speicherung der Daten und den schnellen Zugriff auf die Daten über den Schlüssel. Maps sind in `<map>` definiert und enthalten dort folgende Deklaration:

```
template<class Key, class Value, class Compare>
class map {};
```

Im Folgenden werden nur einige elementare Eigenschaften der Maps beschrieben. Für weitergehende Information sei wieder auf die Literatur verwiesen. Im einfachsten Fall erfolgt die Deklaration einer Map über:

```
map([Key], [Value])
```

Methoden	Funktionalität
`size_type count(const Key& key) const`	Liefert die Anzahl der Paare, deren Schlüssel mit `key` übereinstimmt
`iterator find(const Key& key) const`	Liefert Iterator auf ein Paar, falls key mit Schlüssel übereinstimmt. Sonst Iterator auf `end()`
`Value& operator[](const Key& key)`	Verknüpft `key` mit einem Standardwert, falls noch kein Wert für `key` vorhanden und gibt Zeiger auf diesen neuen Wert zurück. Ist Wert für `key` vorhanden, wird dieser Zeiger geliefert

Tabelle 3.5: Einige Funktionen von map

also z.B.

```
map<string, int> MapStringInt;
```

zur Definition einer Map, die int-Zahlen enthält, auf die über einen String zugegriffen wird, oder

```
map<long int, Student> MapStd;
```

zur Definition einer Map, die Studentendaten enthält, auf die über eine `long int`-Zahl (z.B. Matrikelnummer) zugegriffen wird.

Über den Subskript-Operator `operator[]()` erfolgt dann der Zugriff auf die einzelnen Datenpaare. Die Funktionen `begin, end, rbegin, rend, empty` und `size` sind analog zu den übrigen Containern auch für Maps definiert. In Tabelle 3.5 finden Sie noch einige hilfreiche Funktionen für Maps.

Maps unterstützen Random-Access-Iteratoren und den =-Operator. Die Dereferenzierung eines Iterators liefert jeweils ein `pair`-Objekt aus Schlüssel und Wert. Mit `first` kann dann auf den Schlüssel, mit `second` auf den Wert zugegriffen werden. Man beachte, dass dies keine Memberfunktionen sind und sie daher ohne `()` geschrieben werden.

Beispiel:

```
#include <iostream>
#include <string>
#include <map>
#include "Student.h"

using namespace std;
typedef map<long int, Student> MapStd;

void main()
{
  MapStd StudentMap;
  MapStd::iterator itM;
```

```
  StudentMap[124252] = Student("Meier","ET");
  StudentMap[242521] = Student("Mueller","ET");
  StudentMap[254115] = Student("Schulze","ET");

  for (itM = StudentMap.begin(); itM != StudentMap.end(); itM++)
  {
    cout <<  itM->first << ' ';
    cout << (itM->second).getName() << endl;
  }
  return;
}
```

mit folgender Definition für Student:

```
#include <string>

using namespace std;

class Student
{
  public:
    string getName();
    Student();
    Student(char*, char*);
    virtual ~Student();
  private:
    string p_sFach;
    string p_sName;
};

Student::Student() {}
Student::Student(char* aName, char* aFach) :
    p_sName(aName), p_sFach(aFach) {}
Student::~Student() {}
string Student::getName() { return p_sName; }
```

Die Ausgabe ist:

```
124252 Meier
242521 Mueller
254115 Schulze
```

Der Vorteil der Nutzung einer Map statt eines Vektors bei einem `int`-Schlüssel ist, dass man hier nur die wirklich benutzten Elemente anlegt und nicht so viele wie der maximale Schlüsselwert.

Eine sehr häufige Anwendung der Maps ist die Zuordnung zwischen externen Textschlüsseln auf interne Werte, wie im folgenden Beispiel die Zuordnung der Wochentagesnamen

zu dem entsprechenden Wochentag. Man beachte: Die Reihenfolge der Elemente in der sequentiellen Auflistung entspricht der Sortierfolge (<-Operator) des Schlüssels. Falls ein Schlüssel beim Zugriff nicht gefunden wird, wird 0 zurückgegeben.

Beispiel:

```
#include <iostream>
#include <string>
#include <map>

using namespace std;
typedef map<string, int> MapStrInt;

void main()
{
  MapStrInt Wochentage;
  MapStrInt::iterator itM;
  string sTag;
  Wochentage["Montag"] = 1;
  Wochentage["Dienstag"] = 2;
  Wochentage["Mittwoch"] = 3;
  Wochentage["Donnerstag"] = 4;
  Wochentage["Freitag"] = 5;
  Wochentage["Samstag"] = 6;
  Wochentage["Sonntag"] = 7;

  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout << itM->first << ' ' << itM->second << endl;

  do {
    cout << endl << "Gib Wochentag: ";
    cin >> sTag;
    cout << endl << sTag;
    cout << " ist der " << Wochentage[sTag];
    cout << ". Wochentag.";
  }
  while(sTag != "");

  return;
}
```

Die Ausgabe dieses Programms lautet dann:

```
Dienstag 2
Donnerstag 4
Freitag 5
Mittwoch 3
Montag 1
```

```
Samstag 6
Sonntag 7

Gib Wochentag: (Eingabe: Montag)

Montag ist der 1. Wochentag.
Gib Wochentag: (Eingabe: Dienstag)

Dienstag ist der 2. Wochentag.
Gib Wochentag: (Eingabe: Freitag)

Freitag ist der 5. Wochentag.
Gib Wochentag: (Eingabe: Ende)

Ende ist der 0. Wochentag.
```

Die Funktionen `erase()` und `insert()` funktionieren analog zu den Listen. Um einen Iterator auf ein bestimmtes Element zu erhalten, existiert die Memberfunktion

```
find([schlüssel])
```

die einen entsprechenden Iterator liefert.

Beispiel:

```
#include <iostream>
#include <string>
#include <map>

using namespace std;
typedef map<string, int> MapStrInt;

void main()
{
  MapStrInt Wochentage;
  MapStrInt::iterator itM;
  string sTag;

  Wochentage["Montag"] = 1;
  Wochentage["Dienstag"] = 2;
  Wochentage["Mittwoch"] = 3;
  Wochentage["Donnerstag"] = 4;
  Wochentage["Freitag"] = 5;
  Wochentage["Samstag"] = 6;
  Wochentage["Sonntag"] = 7;

  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout << itM->first << ' ' << itM->second << endl;
```

```
  itM = Wochentage.find("Donnerstag");
  Wochentage.erase(itM);
  itM = Wochentage.find("Montag");
  Wochentage.erase(itM,Wochentage.end());

  cout << endl;
  for (itM = Wochentage.begin(); itM != Wochentage.end(); itM++)
    cout <<  itM->first  << ' ' <<  itM->second << endl;

  return;
}
```

Ausgabe:

```
Dienstag 2
Donnerstag 4
Freitag 5
Mittwoch 3
Montag 1
Samstag 6
Sonntag 7

Dienstag 2
Freitag 5
Mittwoch 3
```

Vollständigkeitshalber ist in Abbildung 3.4 noch eine Übersicht aller Containerschnittstellen dargestellt.

3.4.2 Iteratoren

Iteratoren sind eine Verallgemeinerung der C++-Pointer, die es gestatten, die Elemente eines Containers (oder C-Arrays, C++-iostreams, ...) zu durchlaufen. Ein Iterator hat zu jedem Zeitpunkt eine bestimmte Position innerhalb eines Containers, die er so lange beibehält, bis er durch einen neuen Befehlsaufruf wieder versetzt wird.

Die STL definiert fünf Kategorien von Iteratoren, die hierarchisch angeordnet sind:

Input	kann in einem Schritt ein Element in Vorwärtsrichtung lesen
Output	kann in einem Schritt ein Element in Vorwärtsrichtung schreiben
Forward	Kombination von Input- und Output-Iterator
Bidirectional	wie Forward, kann sich aber auch rückwärts bewegen
Random-Access	wie Bidirectional, kann jedoch auch springen

Die grundlegenden Operationen eines Iterators sind:

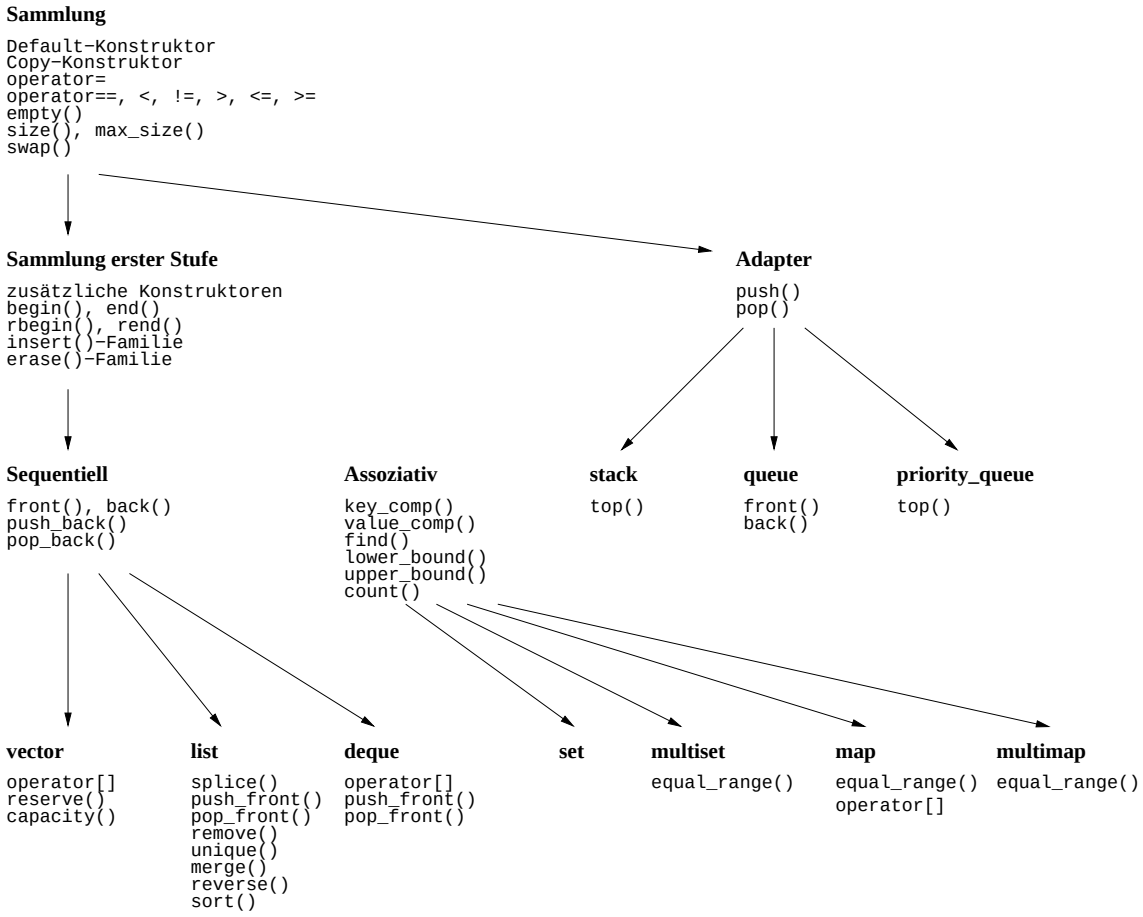


Abbildung 3.4: Hierarchie der Containerschnittstellen

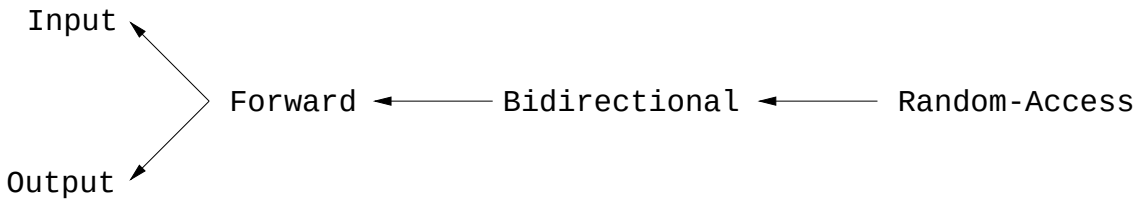


Abbildung 3.5: Hierarchie der Iteratoren

- die Dereferenzierung (Operatoren `*` und `->`), die ein Element des richtigen Typs liefert (Typ der Elemente des Containers)
- die pre- und post-Inkrement-Operatoren (`++` bzw. `--`)
- der Test auf Gleichheit/Ungleichheit (`==` bzw. `!=`)

Es sind nur die zusätzlichen Operatoren definiert, die für die jeweilige Containerklasse effizient ausführbar sind (z.B. kein Random-Access-Zugriff für Listen).

Bei Random-Access-Iteratoren sind zusätzlich

- der Index-Operator `[]`,

- die Addition/Subtraktion `+, +=, -, -=` und
- die Vergleichsoperatoren (`<, >, <=, >=`) definiert.

Im Rahmen dieses Skriptes können nur elementare Eigenschaften zur Nutzung der Iteratoren beschrieben werden. Zur Vertiefung, insbesondere für die genaue Unterscheidung verschiedener Iteratortypen (const-Iterator, Reverse-Iterator) und zur Konstruktion eigener Iteratoren, sei auf die Literatur verwiesen.

Iteratoren erhält man als Ergebnis einer entsprechenden Funktion (wie `begin()` oder `end()`) oder explizit über eine Deklaration wie:

```
vector<int>::iterator itV1;
```

Möchten Sie mittels eines Iterators innerhalb einer Konstant-Elementfunktion auf einen Container zugreifen, muss auch der Iterator als konstant deklariert werden, etwa:

```
list<int>::const_iterator citL1;
```

Iteratoren, die hinter das letzte Element eines Containers zeigen, heißen past-the-end. Die wichtigsten Funktionen von Containern, die einen Iterator liefern, finden Sie in Tabelle 3.6.

Methoden	Funktionalität
`iterator begin()`	Liefert Iterator auf das 1. Element
`iterator end()`	Liefert past-the-end. Dereferenzierung dieses Iteratorwertes ist nicht möglich.

Tabelle 3.6: Einige Funktionen, die Iteratoren liefern

Beispiel:

```
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;

void main()
{
  vector<int> v(3, 1);
  v.push_back(7);        // v: 1 1 1 7
  int sum = 0;
  vector<int>::iterator itV = v.begin();
  cout << "Summe(";
  while (itV != v.end())
  {
    cout << setw(2) << *itV;
    sum += *itV;
    itV++;
  }
```

```
  cout << " ) = " << sum << endl;
  return;
}
```

Die Ausgabe ist dann: `Summe( 1 1 1 7 ) = 10`

3.4.3 Algorithmen

Um einen Container der STL effektiv nutzen zu können, sollte dieser Standardoperationen wie das Abfragen der Größe, Kopieren, Iterieren, Sortieren und Suchen nach Elementen unterstützen. Dazu bietet die STL etwa 60 Algorithmen, welche die fundamentalen Anforderungen eines Anwenders erfüllen. Sie werden alle in `<algorithm>` deklariert.

Jeder Algorithmus wird durch eine Template-Funktion ausgedrückt. Der Zugriff auf Elemente einer Datenstruktur erfolgt über Iteratoren, sodass diese Algorithmen auf unterschiedliche Datenstrukturen (Container, C-Arrays, Sammlungen anderer Hersteller) angewendet werden können.

Nachfolgend werden einige Algorithmen vorgestellt. Die Anwendung dieser Algorithmen ist für die Lösung der Praktikumsaufgaben jedoch nicht vorgeschrieben. Dieses und das folgende Kapitel dienen nur zur vollständigen Beschreibung der STL.

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f)
```

Algorithmus `for_each` führt `f` mit jedem Element des Intervalls `[first, last)` aus und gibt den Eingabeparameter `f` zurück.

```
template<class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last,
              Predicate pred, Size& n)
```

`count_if` zählt die Elemente des Intervalls `[first, last)`, für die `pred` true ergibt und addiert die Anzahl zu `n`. Beachten Sie, dass `n` nicht automatisch mit Null vorbelegt wird.

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value)
```

`fill_n` weist, beginnend an der Position `first`, `n` Elementen den Wert `value` zu. Der Iterator steht dann an der Stelle `first + n`.

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last)
```

`sort` sortiert alle Element des Intervalls `[first, last)` in aufsteigender Reihenfolge. Zum Vergleich der Elemente wird der `<`-Operator verwendet. Es gibt eine zweite Version des Algorithmus bei der die gewünschte Vergleichsfunktion zusätzlich angegeben werden kann.

```
template<class T>
```

```
const T& max(const T& a, const T& b)
```

`max` gibt einen Zeiger auf den größeren der beiden Werte `a` und `b` zurück bzw. `a`, wenn die Werte gleich sind. Verglichen wird wieder mit dem <-Operator. Auch hier gibt es eine Version mit wählbarem Vergleichsoperator. Analog zu `max` gibt es auch einen `min`-Algorithmus.

3.4.4 Funktionen und Funktionsobjekte

Bei den Algorithmen fällt auf, dass einige als Parameter ein Prädikat bzw. eine Funktion verlangen. Es besteht somit die Möglichkeit, einem Algorithmus selbst geschriebenen Code zu übergeben bzw. die in der STL definierten Funktionsobjekte zu verwenden.

Soll eine Funktion, die für viele Elemente aufgerufen wird, zwischen den Aufrufen Daten aufheben und am Ende ein Resultat liefern, ist dafür eine Memberfunktion einer Klasse einer alleinstehenden Funktion vorzuziehen, da diese die Möglichkeit bietet, Daten zu verwalten. Dies soll folgendes Beispiel verdeutlichen.

```
template<class T>
class Summe
{
  public:
    Summe(T i = 0) : res(i) {}
    void operator()(T x) { res += x; }  // Aufruf-Operator
    T ergebnis() { return res; }
  private:
    T res;
};

void f(list<double>& ld)
{
  Summe<double> s;
  // s() fuer jedes Element von ld aufrufen
  for_each(ld.begin(), ld.end(), s);
}
```

Man sieht, dass `s` über den Aufruf-Operator `operator()` wie eine Funktion behandelt wird. Objekte mit einem solchen Operator nennt man funktionsartiges Objekt, Funktor oder einfach Funktionsobjekt.

Um das Erstellen von Funktionsobjekten zu erleichtern, werden in der STL zwei Basisklassen `unary_function` (1 Parameter) und `binary_function` (2 Parameter) definiert, von denen eigene Funktionsobjekte abgeleitet werden können.

Funktionsobjekte der STL werden in folgende Gruppen unterteilt:

- Prädikate
- arithmetische Funktionsobjekte

- Adapteroperator()

Prädikate sind solche Funktionsobjekte, die einen bool zurückliefert. Sie werden oft zum Auslösen von Aktionen oder zum Ordnen von Elementen verwendet. Beispielsweise könnte man hier `equal_to, less` oder `logical_not` aufführen.

Arithmetische Funktionsobjekte stellen die standardisierten arithmetischen Funktionen zur Verfügung, wie `plus, multiplies` oder `modulus`.

Adapter nennen sich solche Funktionsobjekte, denen noch eine Hilfsfunktion zugeordnet ist, mit der das Erstellen von Instanzen vereinfacht wird. Zum Negieren eines Prädikates gibt es z.B. das Funktionsobjekt `unary_negate` mit der Hilfsfunktion `not1()`.

Alle Funktionsobjekte sind in `<functional>` deklariert und meist als Template definiert. Eine ausführliche Auflistung aller Funktionsobjekte finden Sie wiederum in der vorgeschlagenen Literatur.

4 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 2017“ eingesetzt.

4.1 Anmelden im CIP-Pool

Voraussetzung zum Arbeiten mit den CIP-Pool-Rechnern ist, dass Sie eine entsprechende Benutzerkennung (Benutzername) haben. Normalerweise besitzen Sie diese bereits aus vorangegangenen Veranstaltungen im CIP-Pool. Ansonsten erhalten Sie diese am ersten Praktikumstermin. In jedem Fall finden Sie Ihre Benutzerkennung auf Ihrem Testatbogen, den Sie am ersten Praktikumstermin bekommen.

4.2 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. Später können Sie das Passwort durch den entsprechenden Eintrag nach Drücken von `STRG+ALT+ENTF` (siehe Abbildung 4.1). 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 der Bluecard) 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.

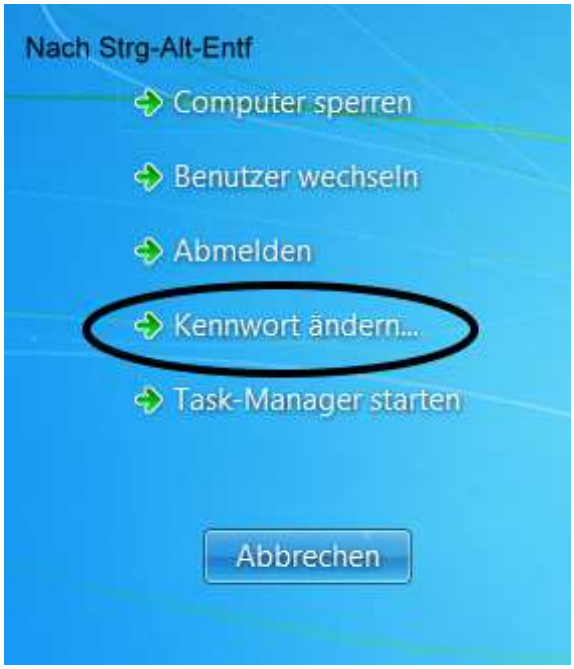


Abbildung 4.1: Nachträgliches Ändern des Login-Passworts

Dateien

Jeder Benutzer hat einen eigenen Speicherbereich auf dem Server, den er von allen Rechnern des CIP-Pools als Laufwerk U: erreichen kann. Maximal können dort **1 GB** Daten gespeichert werden. Bitte bearbeiten Sie die Projekte nur dort (nicht auf dem Desktop oder USB-Stick).

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 im Laufwerk Q: kann jeder Benutzer lesen, ändern und löschen! Alle Dateien auf diesem Laufwerk werden jedes Wochenende automatisch 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: (P:\UserGrp\PI2) 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. (s. Abbildung 4.2). Geben Sie dort Ihren Benutzernamen (CIP-Kennung = 7stellige Ziffernfolge) und Ihr Kennwort ein. Beim ersten Anmelden im CIP-Pool ist das Kennwort Ihre Matrikelnummer.

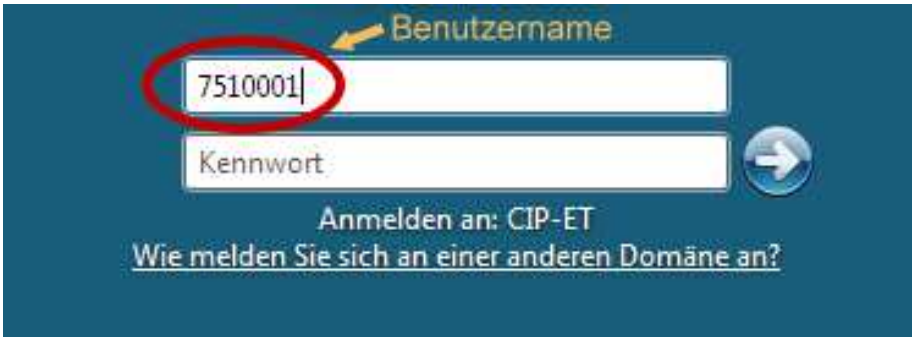


Abbildung 4.2: Anmelden unter Windows

Nach der Eingabe Ihrer Werte klicken Sie auf den Pfeil 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.

4.3 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. USB-Sticks können direkt am Arbeitsplatz benutzt werden. Warten Sie bitte nach dem Einstecken einige Zeit bis der Stick erkannt

wird. Sollte ihr Stick ausnahmsweise nicht erkannt werden, können Sie die Copy-Rechner zum Kopieren der Daten vom USB-Stick benutzen. Dort können Sie auch von DVD kopieren bzw. DVDs brennen. Der Stick kann ohne spezielles Auswerfen nach Beendigung des Kopiervorgangs rausgezogen werden.

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

4.4 Einstellungen der DOS-Box

Oftmals ist es empfehlenswert, in der DOS-Box mehr als 25 Ausgabezeilen anzuzeigen oder eine größere Breite als 80 Zeichen zur Verfügung zu haben (z.B. bei Programmausgaben). Um dies einzustellen, öffnen Sie in der DOS-Box, wenn sie von Ihrem Programm gestartet wurde, das Eigenschaften-Fenster und klicken Sie dort das Register Layout an. Stellen Sie die Höhe/Breite der Fensterpuffergröße auf einen höheren Wert (z.B. 300/120) und wählen Sie die Änderung für die Verknüpfung. In der DOS-Box erscheint dann am rechten/unteren Rand ein Scroll-Balken, so dass Sie jetzt mit diesem Balken durch 300 Ausgabezeilen bzw. 120 Zeichen scrollen können.

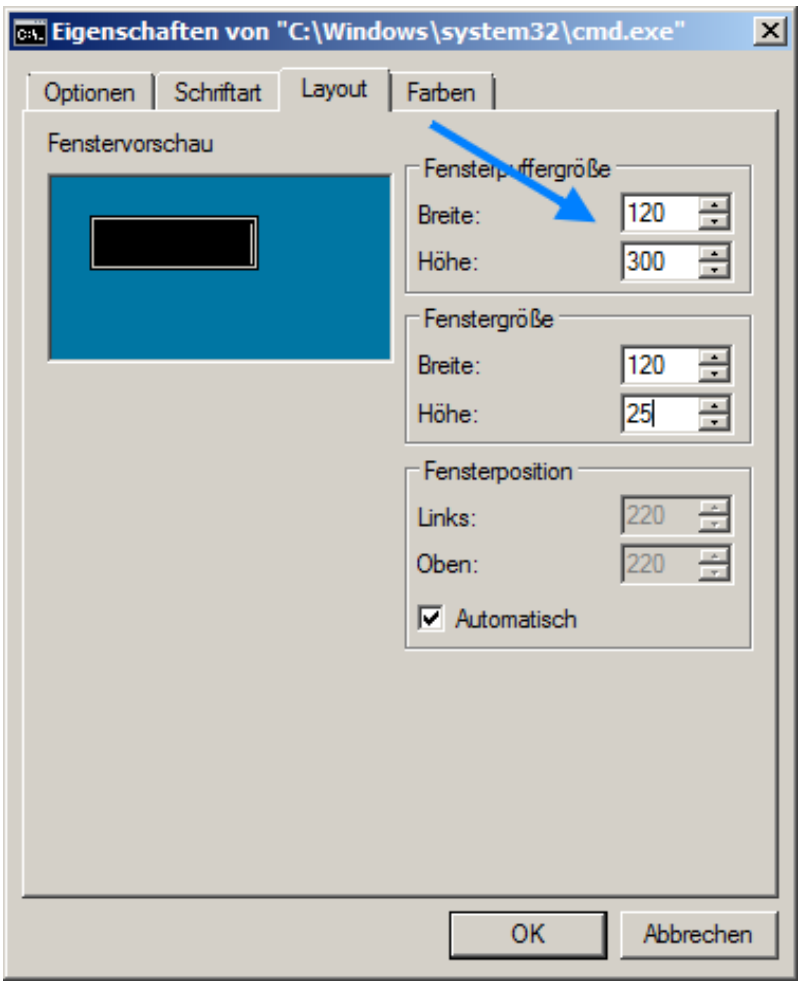


Abbildung 4.3: Fensterpuffergröße ändern

5 Arbeiten auf eigenem PC / Laptop

Für die Durchführung des Praktikums wird die Entwicklungsumgebung „Visual Studio“ unter Windows eingesetzt. Falls Sie das Praktikum auf Ihrem eigenen PC oder Laptop bearbeiten möchten, müssen Sie dort diese Software installieren. Für Apple-Mac-Benutzer empfehlen wir, Windows und VisualStudio in einer VirtualBox oder DualBoot-Umgebung zu installieren. VisualStudio für Mac ist wegen der benutzten Bibliotheken nicht geeignet. Abnahmen in anderen Entwicklungsumgebungen (Eclipse, XCode etc.) sind nicht möglich.

Für die Benutzung des Grafikservers muss Java auf dem Rechner installiert sein. Damit der direkte Aufruf aus der Entwicklungsumgebung funktioniert, muss dies die 32Bit-Variante sein. Diese kann ggf. zusätzlich zur 64Bit-Variante installiert werden.

5.1 Download der benötigten Dateien

Die für das Praktikum benötigte Software kann kostenfrei über die RWTH aus dem Internet heruntergeladen werden. Die Software finden Sie unter folgenden Links:

- `https://doc.itc.rwth-aachen.de/pages/viewpage.action?pageId=3475941` (Microsoft Imagine für Windows und Visual Studio)
- `https://java.com/de/download/manual.jsp` (Java Runtime Environment)
- `https://www.virtualbox.org/wiki/Downloads` (VirtualBox ggf. für Mac)

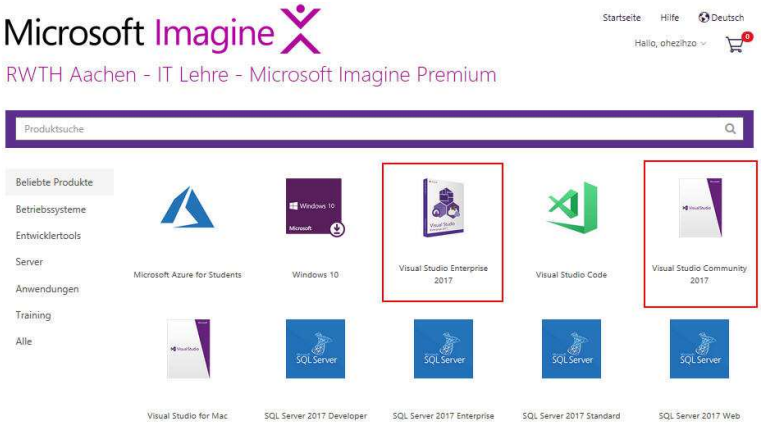


Abbildung 5.1: Microsoft Imagine

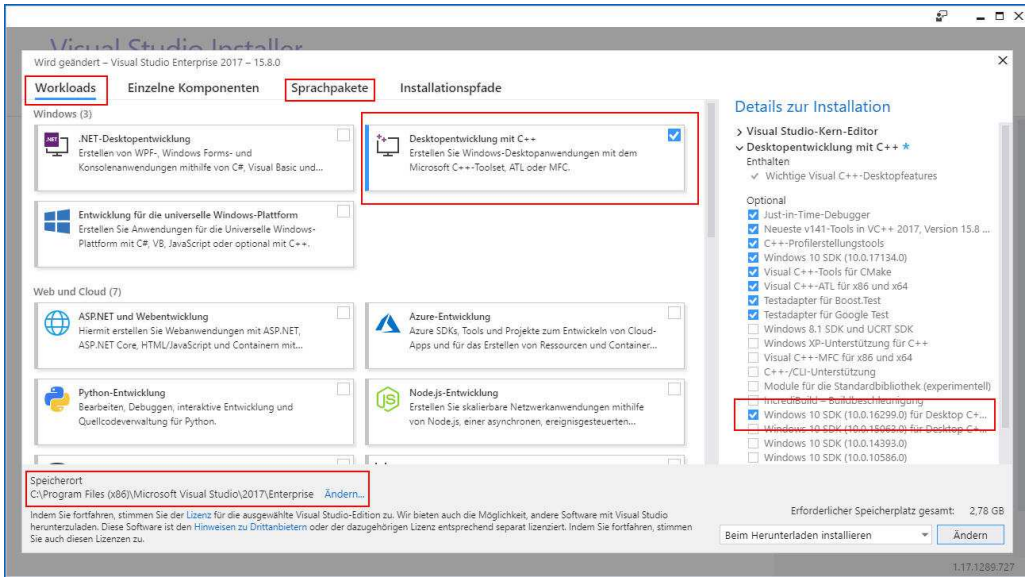


Abbildung 5.2: VS Installer: Auswahl

Wählen Sie beim Download (siehe Abb.5.1) für Visual Studio die Enterprise-Version, wenn Sie VS auch später noch einsetzen möchten. Das ist die professionelle Version mit allen Funktionen der Softwareentwicklung. Diese Version benötigt je nach Installation einige GB freien Festplatten-Speicherplatz auf dem Systemlaufwerk. Falls Sie wenig Speicherplatz haben und Visual Studio nur für das Praktikum benutzen möchten, ist auch die Community-Version dafür ausreichend.

5.2 Installation Visual Studio

Zur Installation von Visual Studio starten Sie (als Administrator) das Installationpaket (z.B. vs_enterprise_Studio_2017.exe). Dafür müssen Sie sich innerhalb des RWTH-Netzes befinden (direkt oder per VPN). Nach einer Lizenzbestätigung sehen Sie dann die Auswahl des Installers (siehe Abb. 5.2). Für das Praktikum wählen Sie dort bitte nur das Paket `Desktopentwicklung mit C++` und ergänzen Sie die vorgeschlagenen Optionen durch `Windows 10 SDK für Desktopentwicklung`. Je nach Bedarf können Sie den Installationspfad ändern oder eine zusätzliche Menü-Sprache installieren. Auch bei Änderung des Installationspfads wird noch Speicherplatz auf dem Systemlaufwerk benötigt.

Nach Starten der Installation werden die benötigten Pakete heruntergeladen und anschließend installiert (siehe Abb.5.3). Dies kann je nach System und Internetverbindung bis zu 1 Std. und länger dauern. Nach der Installation ist ein Neustart des Computers erforderlich.

Starten Sie dann Visual Studio 2017. Beim ersten Start werden noch einige Vorbereitung getroffen. Zunächst können Sie sich ggf. mit einem Microsoft-Konto anmelden (siehe Abb.5.4) . Alternativ können Sie dies auch verschieben. Sie haben dann 30 Tage Zeit, die Anmeldung nachzuholen oder einen Productkey einzutragen. Den Produktkey erhalten

Abbildung 5.3: VS Installer: Download

Abbildung 5.4: Anmelden über MS Konto

Abbildung 5.5: Umgebung auswählen

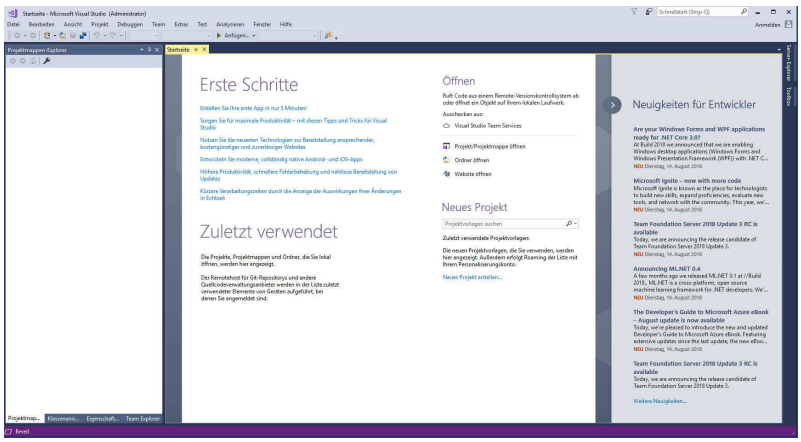


Abbildung 5.6: Entwicklungsumgebung

Sie bei der Bestellung in MS Imagine. Dort können Sie ihn auch später einsehen (unter `Meine Bestellungen`). Im nächsten Schritt müssen Sie die gewünschte Umgebung auswählen (siehe Abb.5.5). Wichtig ist, dass Sie hier statt `Allgemein` die Option `Visual C++` auswählen. Nach einigen weiteren Vorbereitungen sollte die Entwicklungsumgebung dann bereit zur Benutzung sein und etwa wie in Abb.5.5 aussehen.

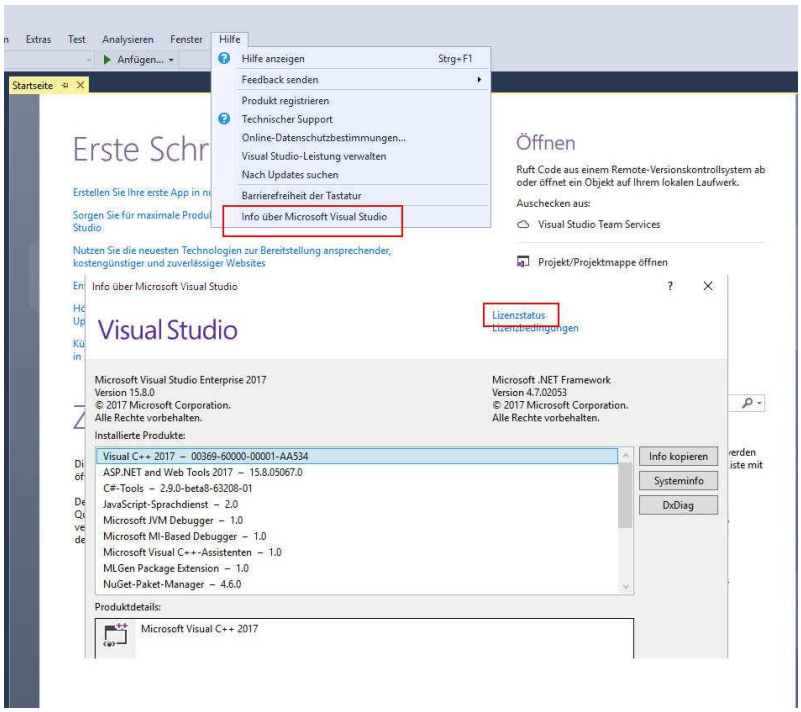


Abbildung 5.7: Eingabe Productkey zur Freischaltung

Zur Eingabe des Productkeys, den Sie bei der Bestellung in MS Imagine bekommen haben, rufen Sie `Hilfe` → `Info über ...` auf. Im darauf folgenden Bildschirm klicken Sie auf `Lizenzstatus` (siehe Abb. 5.7). Danach können Sie unter dem Punkt `Mit einem Productkey entsperren` die Software freischalten.

6 Visual Studio 2017

6.1 Einführung

Im Folgenden werden die wichtigsten Bedienelemente der Entwicklungsumgebung von Visual Studio 2017 (VS 2017) anhand eines kleinen „Hello World“-Programms erklärt. Außerdem geben wir Hinweise und Tipps zur Nutzung des Programms. Starten Sie VS 2017 über das Start-Menü. Danach wird ein neues Projekt erstellt, indem unter `Datei` → `Neu` → `Projekt` → `Visual C++` → *Leeres Projekt* 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. 6.1).

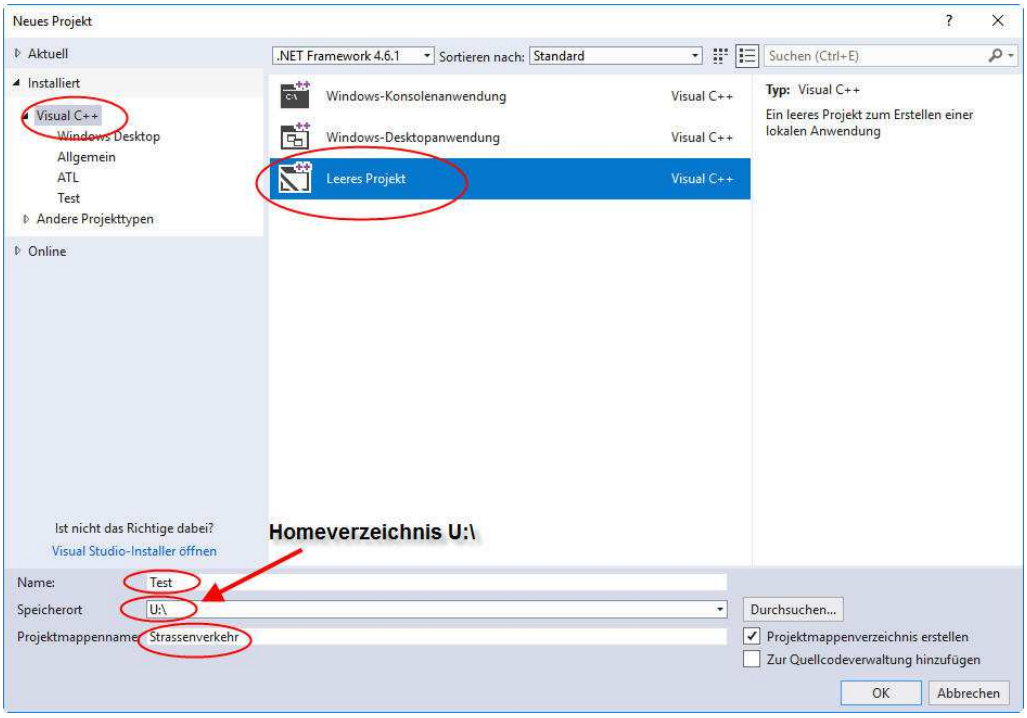


Abbildung 6.1: Neues Projekt erstellen

VS 2017 erkennt automatisch, dass ihr Homeverzeichniss ein Netzlaufwerk repräsentiert und setzt daher 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. 6.2.

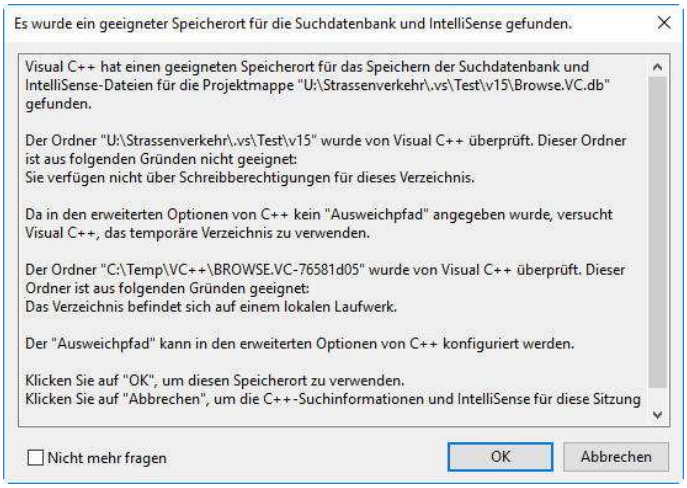


Abbildung 6.2: Hinweis Speicherort Netzlaufwerk

Es wurde als Unterverzeichnis von U:\das Arbeitsmappenverzeichnis U:\Strassenverkehr angelegt. Darunter wurde das Projektverzeichnis U:\Strassenverkehr\Test angelegt, das die Sourcen des Projektes enthalten muss. Alle (im Debug-Modus) erzeugten ausführbaren Programm finden Sie später in dem Debug-Unterverzeichnis der Projektmappe, also hier U:\Strassenverkehr\Debug.

Die wichtigsten Funktionen der Entwicklungsumgebung 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` → `Visual C++` *C++-Datei* wählen. Geben Sie als Namen `main.cpp` ein und drücken auf *Hinzufügen*. Schreiben Sie nun folgendes Programm in das neue Textfenster:

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

int main()
{
  char c;
  cout << "Hello World" << endl;
  cin >> c;
  return 0;
}
```

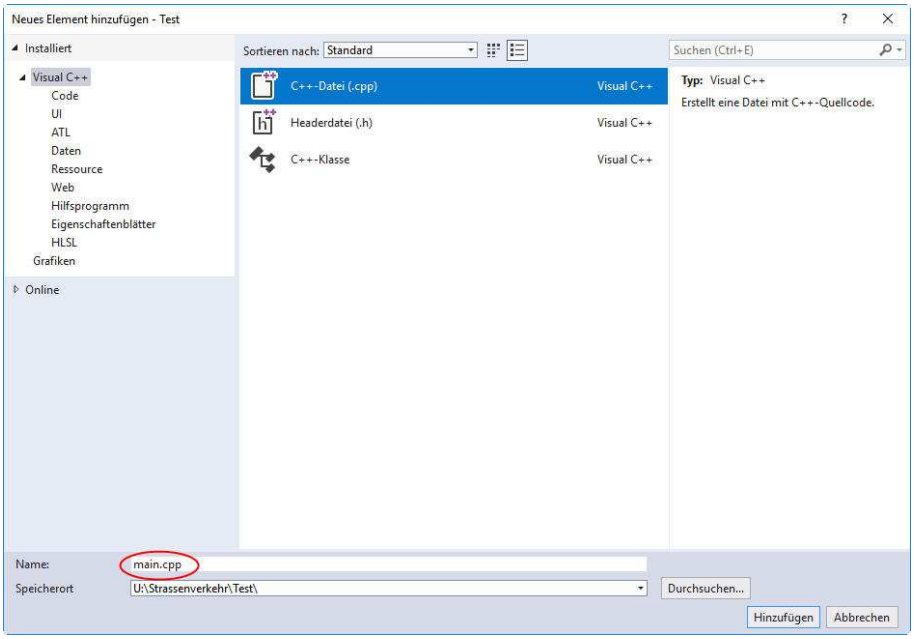


Abbildung 6.3: main.cpp anlegen

Speichern Sie alles ab (Diskettensymbole oder Strg-Umschalt-S), übersetzen Sie das Projekt `Erstellen` → `Test erstellen` und starten Sie es mit `Debuggen` → `Starten ohne Debugging` oder Strg-F5. Suchen Sie die entsprechenden Buttons oder Einträge in den Menüs (`Datei, Erstellen, Debuggen`).

Leider bleibt das Ausgabefester in VS2017 auch beim Starten ohne Debugger nach dem Programmende nicht offen. Sie müssen daher das Warten auf Eingabe selber programmieren oder im Debug-Modus am Ende des Programms einen Haltepunkt setzen (siehe folgende Beschreibung). Ohne Eingabeprogrammierung oder Haltepunkt wird das Programmfenster sonst nach Abarbeitung automatisch geschlossen.

Als nächstes wird beschrieben, wie Sie Klassen hinzufügen und bearbeiten. In Visual Studio ist es üblich, dass für jede Klasse eine Header-Datei (Datei mit Endung .h) und ein Code-Datei (Datei mit Endung .cpp) 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 automatisch.

Erstellen Sie als erstes eine neue Klasse `MyClass` (s. Abbildung 6.4). Wählen Sie dazu unter `Projekt` → `Klasse hinzufügen` → `C++` aus. Geben Sie der neuen Klasse den Namen `MyClass`. Verwenden Sie einen virtuellen Destruktor durch Ankreuzen der entsprechenden Option. Die neue Klasse wird nach `Ok` in den Arbeitsbereich (Workspace) eingefügt.

Implementieren Sie außerdem eine Funktion `vPrint` in die Klassenhierarchie (s. Abbildung 6.5). Markieren Sie dazu in der Klassenansicht die Klasse `MyClass` und wählen Sie im Kontextmenü („rechte Maustaste“) `Hinzufügen` → `Funktion hinzufügen` aus (auch bei Projekt im Menü erreichbar).

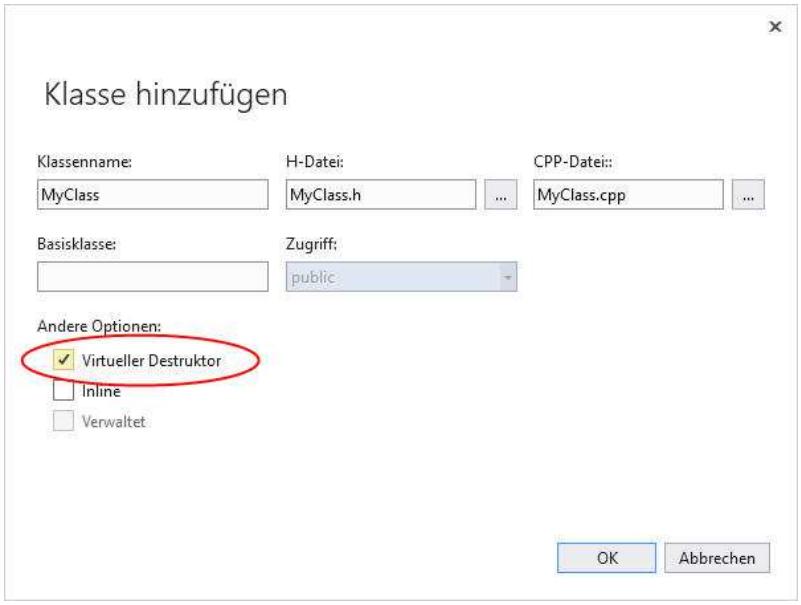


Abbildung 6.4: Klasse erstellen

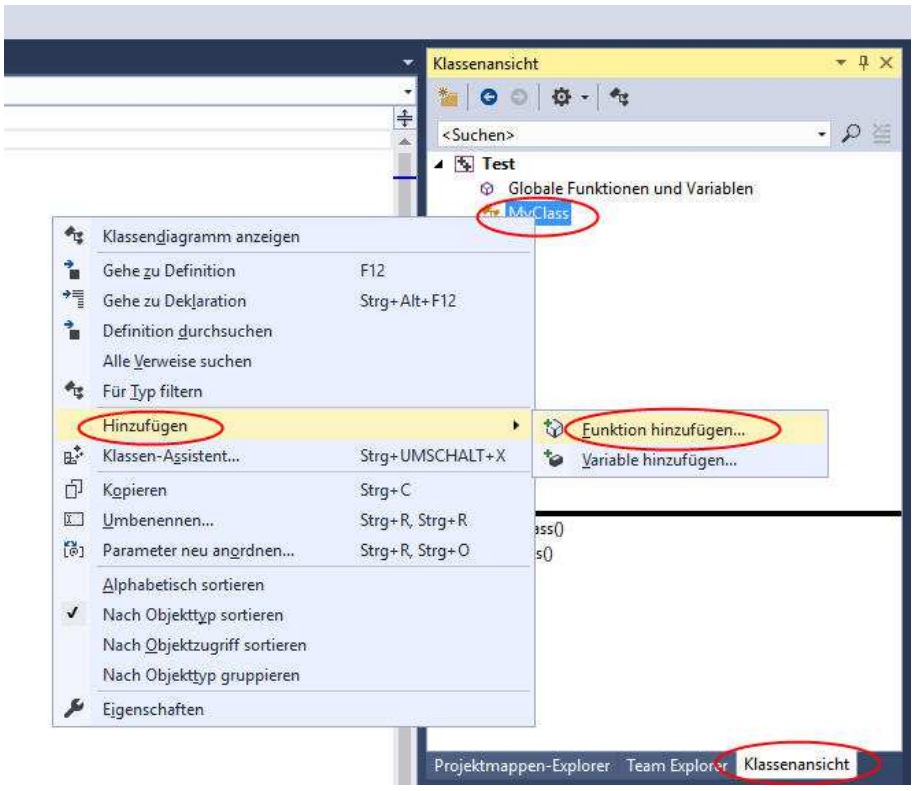


Abbildung 6.5: Assistent: Funktion einfügen

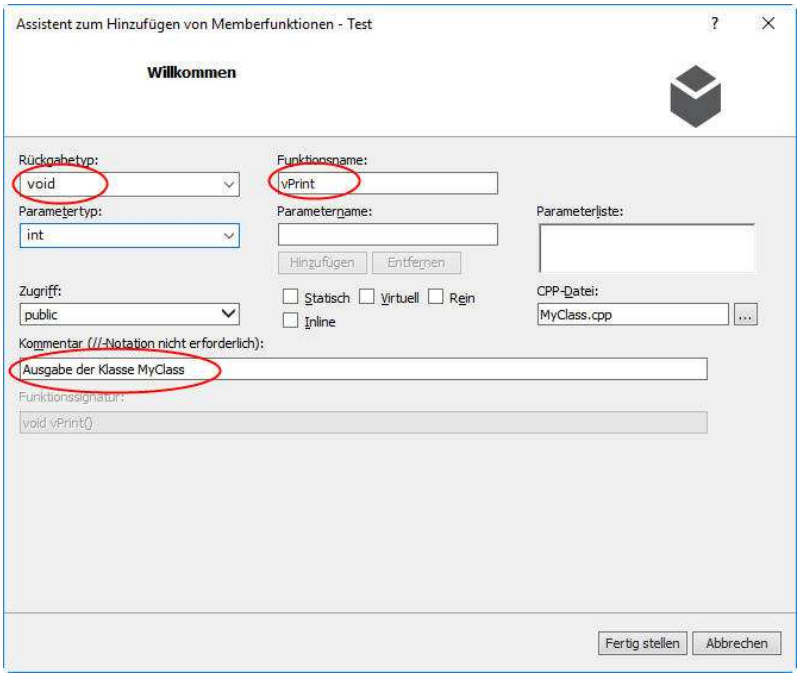


Abbildung 6.6: Dialog: Funktion einfügen

Diese Funktion bekommt im folgenden Dialog den Rückgabetyp `void`, den Funktionsnamen `vPrint`, der Zugriff wird als `public` deklariert und ein kurzer Kommentar angegeben (s. Abbildung 6.6).

Mit einem Doppelklick auf entsprechende Objekte im Arbeitsbereich kann schnell zu bestimmten Klassen, Methoden oder Variablen gesprungen werden. Doppelklicken Sie in der Klassenansicht beispielsweise auf `MyClass()`, öffnet sich ein Editorfenster mit der cpp-Datei. Der Cursor steht auf 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 `vPrint()` das gewünschte „Hello World“ aus. Definieren Sie dann in `main()` eine Instanz der Klasse `MyClass` und rufen Sie dessen Member Funktion `vPrint()` auf. In Abbildung 6.7 sind alle Programmtexte und der aktuelle Arbeitsbereich zu sehen.

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 6.7 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
```

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`.

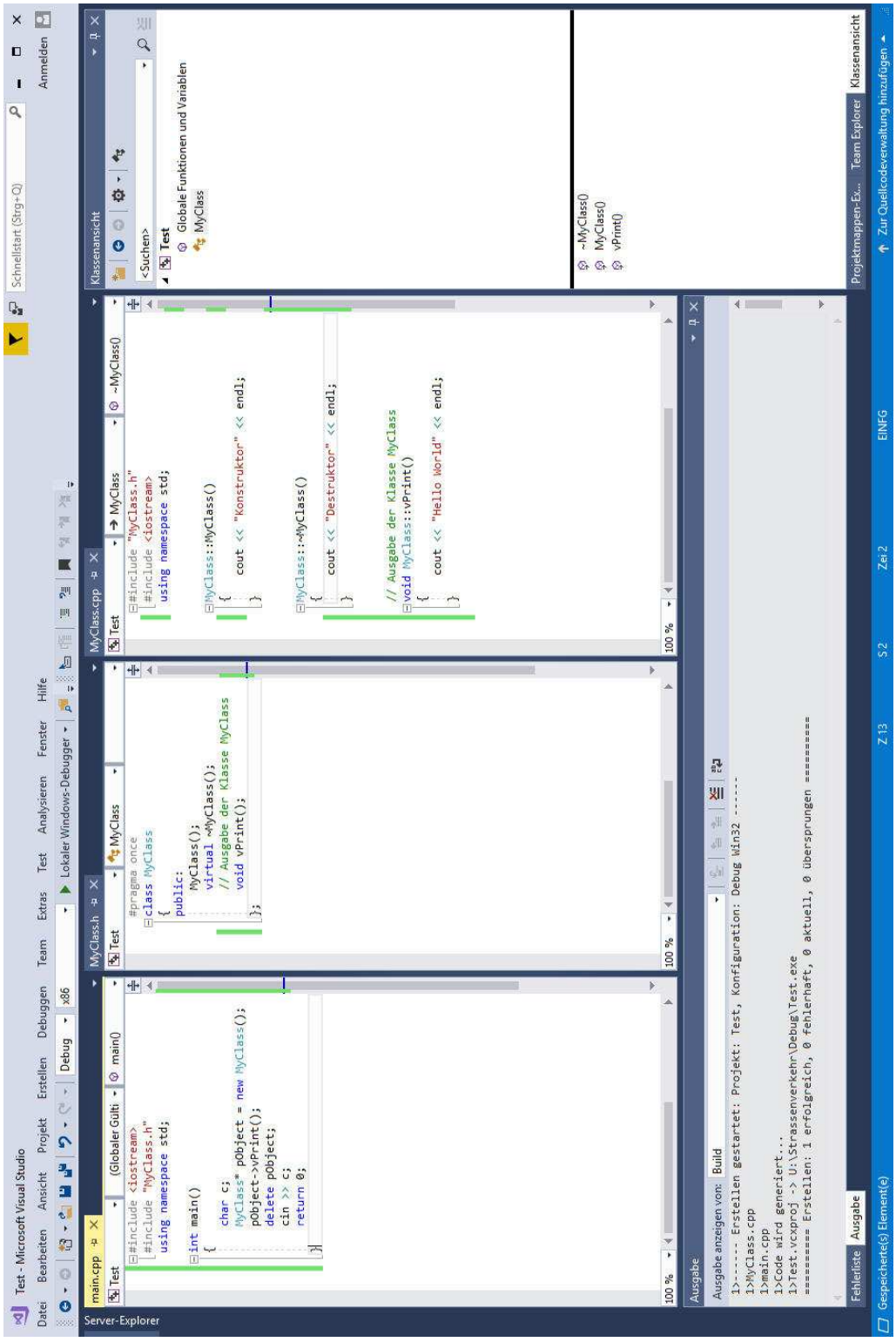


Abbildung 6.7: Komplettes Programm

6.2 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 werden kann, 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 `Strg+Z` (Strg-Taste gedrückt halten und Z drücken). Weitere Beispiele sind `F7` für *Projektmappe erstellen* oder `F5` für *Debuggen starten*.

Wenn Sie Dateien zwischen CIP-Pool und Laptop austauschen oder in neuem Projekt 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. Die Dateien 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`. Nach einem Import von Dateien müssen Sie das Projekt immer neu erstellen.

6.3 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.

6.3.1 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 Verhalten übereinstimmt. Daraus lassen sich dann Rückschlüsse ziehen, welche Teile des Programms fehlerhaft sein könnten und verändert werden sollten.

6.3.2 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.

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. Beim ersten Programmstart sind eventuell nicht alle Funktionen in der Debug-Toolbar zusammengestellt. Dies kann individuell angepasst werden.

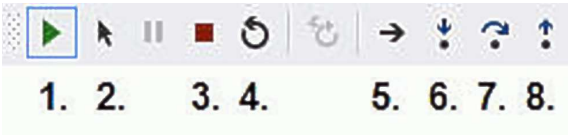


Abbildung 6.8: Debuggen-Funktionsleiste

Die wesentlichen Buttons dieser Funktionsleiste sind:

1. *Weiter.* (Weiter-)Ausführung des Programms bis zum nächsten Haltepunkt.
2. *Weiter bis Cursor.* (Weiter-)Ausführung des Programms bis zur Programmzeile des Cursors im Codefenster.
3. *Debuggen beenden.*
4. *Neu starten.* Debugger beginnt den Programmdurchlauf von vorne.
5. *Anzeigen Anweisung.* Anzeigen der nächsten ausführbaren Anweisung im Codefenster.
6. *Einzelschritt.* Programm wird zeilenweise ausgeführt. Es wird in Funktionen verzweigt.
7. *Prozedurschritt.* Programm wird zeilenweise ausgeführt. Funktionen werden in einem Schritt ausgeführt.
8. *Ausführen bis Rücksprung.* Führt die aktuelle Funktion bis zum Ende aus und springt zur aufrufenden Funktion zurück.

Eine nützliche Eigenschaft von Haltepunkten bei der Fehlersuche ist die Bedingung. Damit kann der Programmablauf gezielt unterbrochen werden. Die Bedingung für einen Haltepunkt kann mit einem Rechtsklick auf das Symbol neben der markierten Zeile definiert

werden. Mögliche Auslöser für die Bedingung kann die Auswertung eines bestimmten Ausdrucks (in Abbildung 6.9 z. B. wenn `p_iID==5`), die bisherige Trefferanzahl oder ein Systemfilter (z.B. ProzessID) sein.

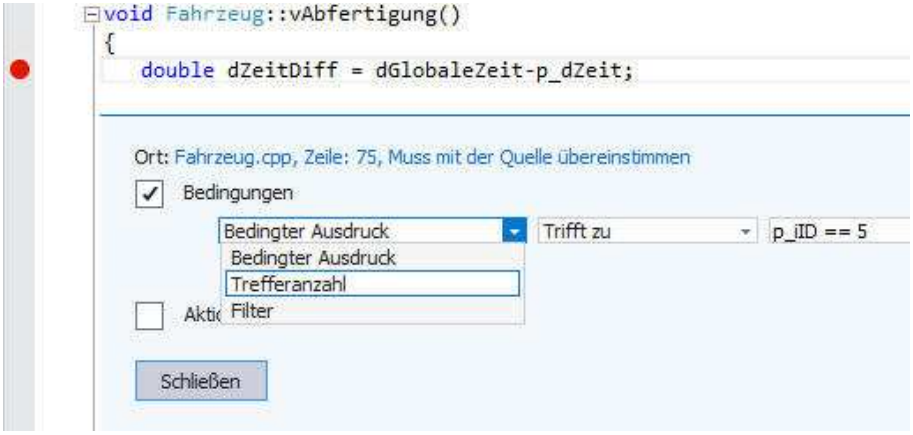


Abbildung 6.9: Setzen eines bedingten Haltepunktes

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`.

- *Lokal*: Hier werden alle lokale Variablen für die aktuelle Funktion oder den aktuellen Gültigkeitsbereich angezeigt
- *Auto*: Zeigt die Variablenwerte im Kontext der aktuell ausgeführten Programmzeile

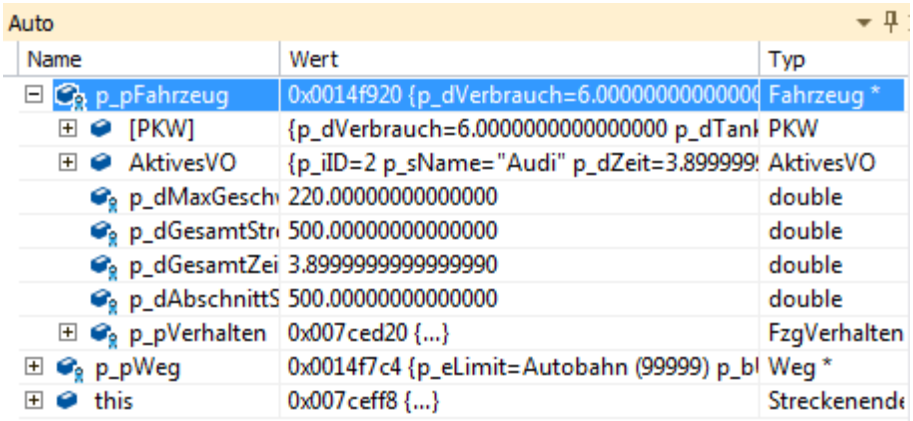


Abbildung 6.10: Debuggen Auto-Fenster

- *Überwachen* (1-4): Hier kann man ausgewählte Werte anzeigen lassen. In der Spalte "Name"kann man angeben, was überwacht, werden soll. Dabei kann man auch Funktionen aufrufen oder Zeiger dereferenzieren. Variablen, die im aktuellen Kontext nicht mehr bekannt sind, werden entsprechend angezeigt (siehe `sKreuzung` in Abbildung 6.11)

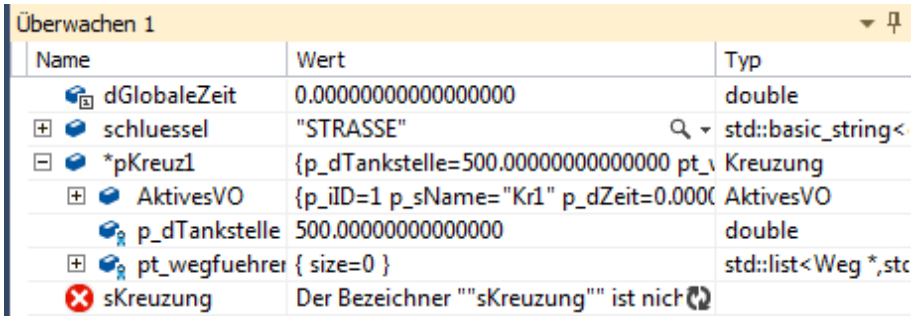


Abbildung 6.11: Debuggen Überwachen-Fenster

- *Aufrufliste*: Hier werden die Funktions- und Prozeduraufrufe angezeigt, die sich derzeit im Stapel befinden. Am Anfang und am Ende der Liste finden Sie ggf. Funktionen, die Sie nicht geschrieben haben. Das sind Bibliotheksfunktionen, die automatisch eingebunden wurden. Diese können Sie normalerweise ignorieren. Wenn Sie eine Zeile in diesem Fenster anklicken, wird der zugehörige Code angezeigt.

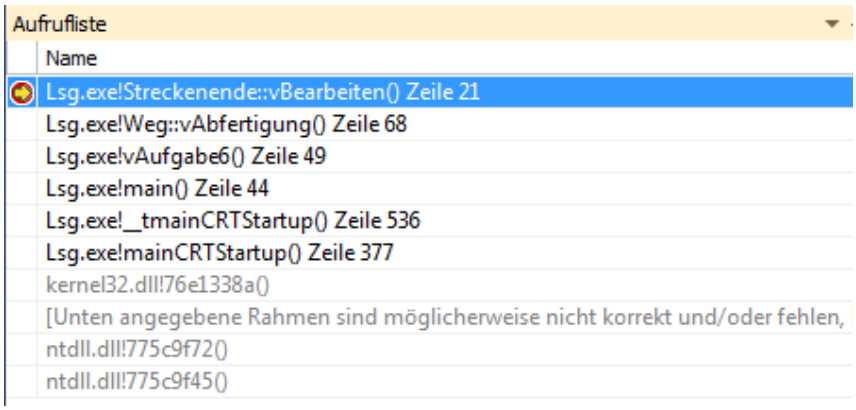


Abbildung 6.12: Debuggen Aufrufliste

Sie können mehrere Fenster gleichzeitig anzeigen lassen oder je nach Bedarf Fenster öffnen und schließen. Die Fensteranordnung bleibt beim nächsten Aufruf so, wie Sie sie zuvor gemacht haben. Möchten Sie die ursprüngliche Anordnung wiederherstellen, können Sie dies mit `Fenster` → `Fensterlayout zurücksetzen`.

Um den Umgang mit dem Debugger etwas zu üben, sind in die Aufgabenstellung dazu eigene Unteraufgaben eingefügt (siehe Aufgabe 1.10 und Aufgabe 5.3). Diese sind von jedem Teilnehmer zu bearbeiten. Falls Sie noch keine Erfahrung im Umgang mit einem Debugger haben, empfehlen wir, etwas mehr Zeit in diese Unterpunkte 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.

7 Aufgaben

7.1 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 2017 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 aufgabentechnisch nicht möglich).

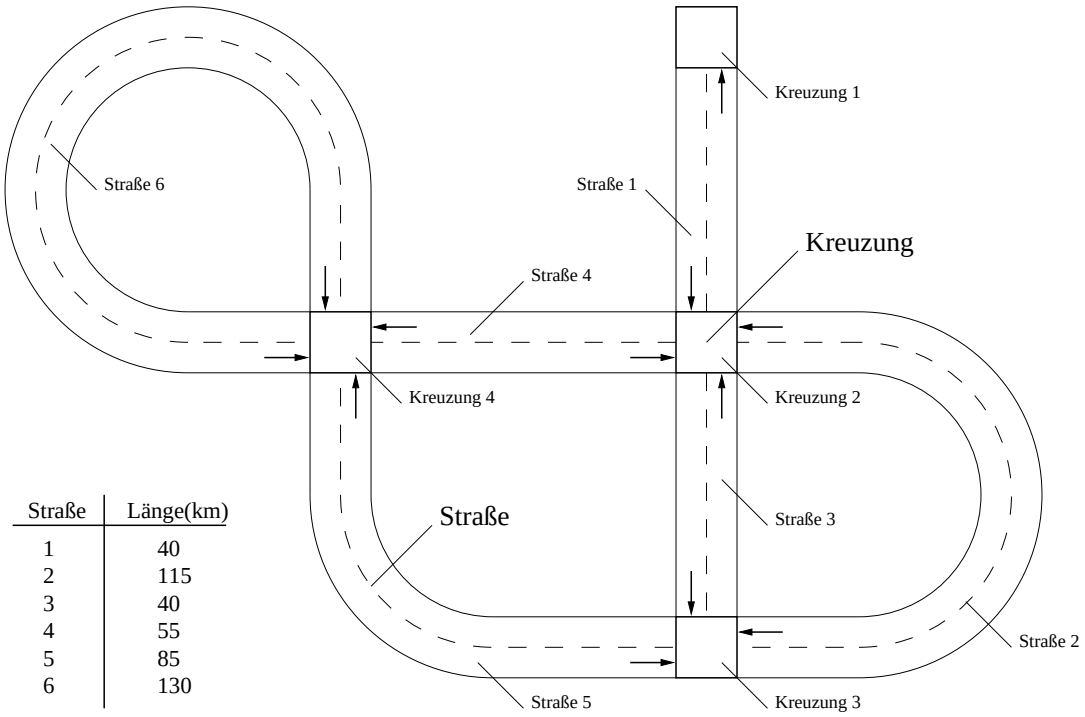
Alle drei Aufgabenblöcke sollen in **einer** gemeinsamen Projektmappe liegen. Darauf wird bei den einzelnen Abnahmen geachtet und es geht auch mit in die Bewertung ein.

7.2 Zielaufgabe

Es soll der Straßenverkehr in einer wenig erschlossenen Gegend modelliert und simuliert werden (s. Abbildung 7.1).

Verschiedene Arten von Fahrzeugen (PKW, Fahrrad) werden zu einem individuellen Startzeitpunkt von einem Knotenpunkt (Kreuzung) 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 7.1: 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 letzte Abfertigungszeit des Verkehrsobjektes mit der globalen Zeit verglichen und 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 7.2 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.

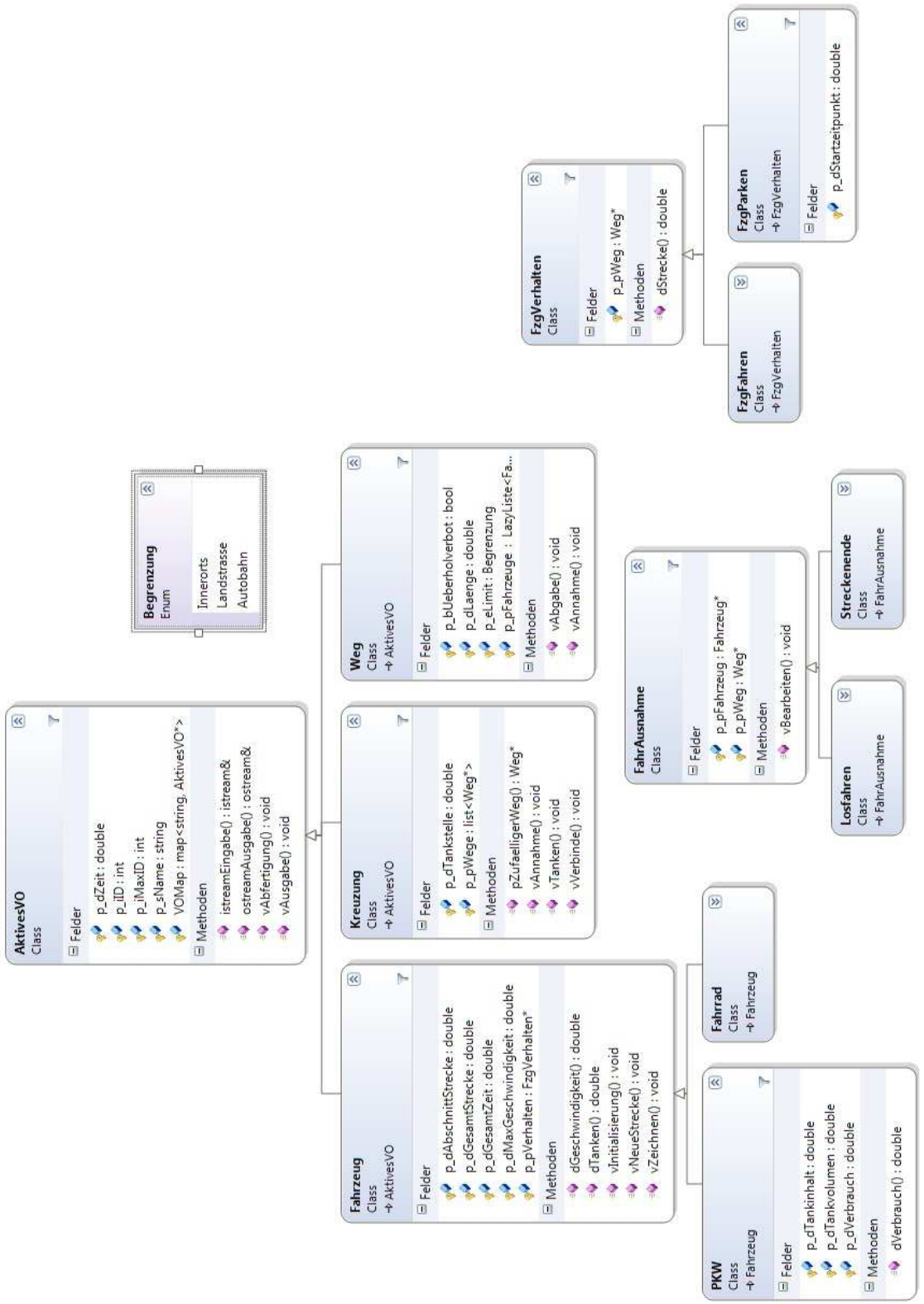


Abbildung 7.2: Klassenhierarchie

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.

7.3 Hinweise zur Implementierung

7.3.1 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 (mit X=1,2,3)` vom Typ *Leeres Projekt* 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 ein neues Projekt `Aufgabenblock_1` vom Typ *(Leeres Projekt)* und die Projektmappe `Strassenverkehr`. Markieren Sie *Projektmappenverzeichnis erstellen*. Klicken Sie nun `Fertig stellen`. In Ihrem Userverzeichnis (`U:\`) sollte jetzt ein Verzeichnis `Strassenverkehr` mit dem Unterverzeichnis `Aufgabenblock_1` existieren.

7.3.2 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` als *(Leeres Projekt)*. 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 2017 müssen nun die kopierten Dateien dem neuen Projekt noch bekannt gemacht werden (`Projekt` → `Vorhandenes Element hinzufügen` ...). Legen Sie das neue Projekt schließlich noch als Startprojekt fest (`Projekt` → `Als Startprojekt festlegen`).

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` → `Projekt neu erstellen`.

7.3.3 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 nur eine private/protected Variable setzt oder zurückliefert, ist das erste Wort des eigentlichen Variablennamens `get` bzw. `set`. Diese Funktionen heißen auch getter/setter-Methoden.
- 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.

7.3.4 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 2017 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 7.1)
- Testen Sie alle erstellten Klassen und Funktionen für (weitgehend) alle denkbaren Situationen. Ein einzelnes Testdatum zeigt noch nicht die Korrektheit des Programms. Wählen Sie entsprechend mehrere repräsentative Testfälle. Dieser Punkt (z.B. „Übersehene“ Testfälle oder nur automatisierte Zufallstests) geht auch in die Bewertung ein.

7.4 Aufgabenblock 1: Grundlagen des Verkehrssystems

7.4.1 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.

7.4.2 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 und virtuelle Vererbung
- Überladen von Operatoren

7.4.3 Aufgabe 1: Fahrzeuge (Einfache Klassen)

1. Starten Sie Visual Studio 2017 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 7.3.1.

2. Implementieren Sie eine Klasse `Fahrzeug` zur Verwaltung verschiedener Fahrzeuge. Die Klasse soll zunächst lediglich private Membervariablen 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 und die ID des erzeugten bzw. gelöschten Objekts 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>`). Programmieren Sie die Zeilenwechsel nach jeder Ausgabe nicht in der Funktion `vAusgabe()` sondern in der Hauptfunktion.

   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 (u.a. Gesamtstrecke um die im vergangenen Zeitraum fahrbare Strecke erhöhen). 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 6.3 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. Nutzen Sie einen bedingten Haltepunkt, um gezielt vor dem 3. Durchlauf der Schleife das Programm anzuhalten.

    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?

7.4.4 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 jetzt 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 Liter, 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. Die Zeitabfrage dazu soll im Testprogramm erfolgen, nicht innerhalb von `dTanken()`. Geben Sie die Ergebnisse (Daten aller Fahrzeuge) nach jeder Abfertigung aus.

   Beachte: Gleichheit von `double`-Werten kann immer nur gegen eine Toleranz ϵ getestet werden, da Fließkomma-Werte aufgrund der Rundung fast nie genau gleich werden. Berechnen Sie dazu z.B. den Absolutbetrag der Differenz bei Gleichheit oder reduzieren Sie eine der Seiten des Vergleichs um ϵ bei $\geq$ oder $\leq$. Die Funktion für den Absolutbetrag `fabs()` finden Sie in der Bibliothek `<math.h>`. Beachten Sie dieses Rundungsproblem bei allen Vergleichen zwischen Fließkomma-Werten.

7.4.5 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. Dazu brauchen Sie die Funktion `vAusgabe()` jedoch nicht aus Ihrem Projekt zu löschen.

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=()` (Das ist nicht dasselbe ;-): 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 (keine doppelten IDs). Begründen Sie Ihre Entscheidung.

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

7.5 Aufgabenblock 2: Erweiterung des Verkehrssystems

7.5.1 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 besitzt 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.

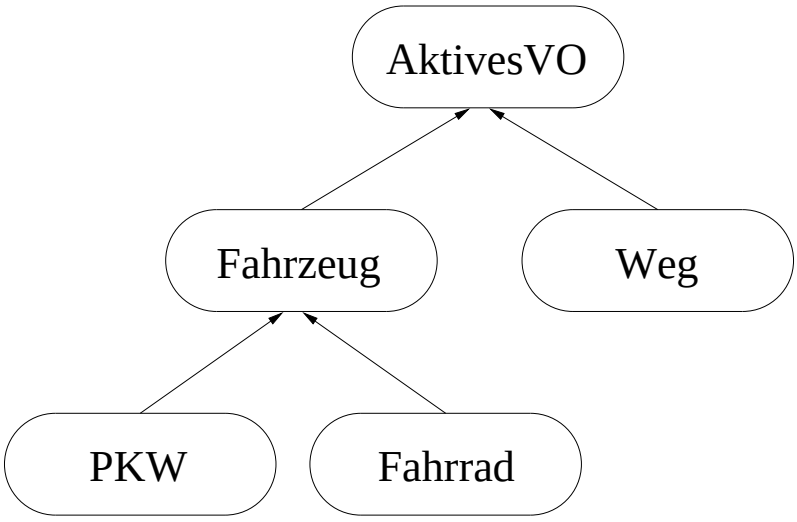


Abbildung 7.3: Klassenhierarchie Aufgabenblock 2

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 eigenes Template erstellt werden.

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

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

1. Fügen Sie der bereits vorhandenen Projektmappe `Strassenverkehr` ein neues Projekt vom Typ *Leeres Projekt* 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 7.3.2).

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. 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. Dieses Prinzip gilt auch für die Konstruktoren. Der Aufruf des Konstruktors der Oberklasse erfolgt dabei über eine Initialisierungsliste.

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 Geschwindigkeitslimit (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. Um allgemein Probleme mit zirkulären Abhängigkeiten (circular dependencies) in den Headerdateien zu vermeiden, kann man meistens folgende Faustregel anwenden: Nur die Headerdatei der Oberklasse in die h-Datei einbinden und alle anderen Typen mit `class xxx` dort bekanntmachen. In der cpp-Datei müssen dann alle notwendigen Headerdateien eingebunden werden, um die Schnittstellen der Funktionen bereitzustellen.

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`

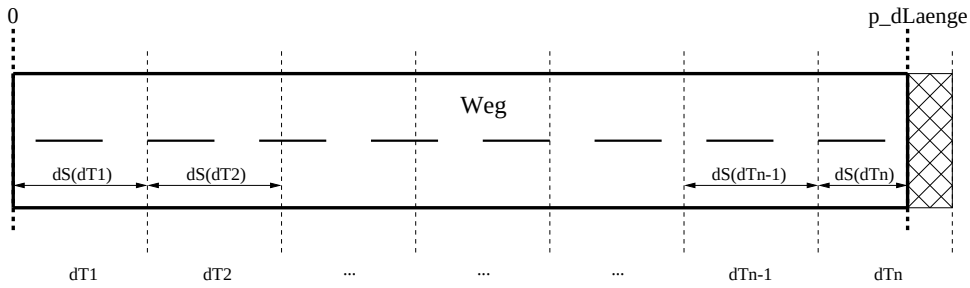
4. Da Fahrzeuge *später* auf verschiedenen Wegen fahren sollen, führen wir hier eine zusätzliche Membervariable `p_dAbschnittStrecke` ein. Diese speichert immer nur die auf dem aktuellen Weg zurückgelegte Strecke. Sie wird in gleicher Weise wie bisher `p_dGesamtStrecke` aktualisiert und beim Betreten des Weges auf 0 gesetzt. Fügen Sie diese Variable Ihren Berechnungen und Ausgaben für `Fahrzeug` hinzu. Die Variable `p_dGesamtStrecke` soll weiterhin gepflegt werden.

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, 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, ohne das Wegende zu überschreiten. 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.



dT : Zeittakt der Abfertigung

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

Abbildung 7.4: Funktionsweise dStrecke()

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?

Da es zur Zeit noch keine Einschränkungen für die Fahrzeuge gibt, soll die Funktion `dStrecke()`, wie in Abbildung 7.4 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()`.

7.5.3 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?

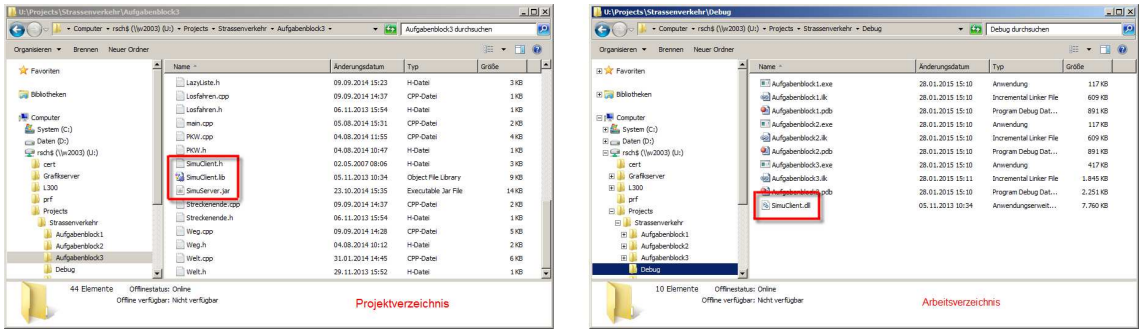


Abbildung 7.5: Grafikdateien im Projekt- und Arbeitsverzeichnis

2. Testen Sie in `vAufgabe_5()` die gerade implementierte Ausnahmebehandlung. Setzen Sie fahrende und parkende Fahrzeuge auf einen Weg und fertigen diesen ab.

   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. Aufgabe zur Nutzung des Debuggers:
   Kontrollieren Sie mit Hilfe des Debuggers, ob das Losfahren immer zum richtigen Zeitpunkt auftritt. Lassen Sie dazu ein Fahrzeug beim Zeitpunkt 3.0 losfahren. Überprüfen Sie den Startzeitpunkt einmal bei einem Zeittakt der globalen Zeit von 0.25 und einmal bei 0.3. Korrigieren Sie ggf. Ihren Code so, dass in beiden Fällen beim Zeitpunkt 3.0 losgefahren wird.

4. 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.

   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 (Verzeichnis mit cpp/h-Dateien). 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 (Verzeichnis mit exe-Dateien). (siehe Abb. 7.5) Um die Datei `SimuClient.dll` angezeigt zu bekommen, müssen Sie vorher im Windows-Dateiexplorer unter `Anzeige` → `Optionen` den Button `Alle Dateien anzeigen` auswählen.

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

   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 RelPosition, double KmH, double Tank);`

- `bool bZeichneFahrrad(string FahrradName, string WegName, double RelPosition, 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 `RelPosition` 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.

- `bool bLoescheFahrzeug(string FahrzeugName)`

  Diese Funktion löscht ein Fahrzeug aus der Simulation. Das Fahrzeug wird nicht mehr angezeigt und aus der Fahrzeugliste im Simulationsserver gelöscht. Die Funktion ist auf PKW und Fahrräder gleichermaßen anwendbar. Die Funktion sollte nur beim endgültigen Entfernen eines Fahrzeugs aus der Siumation benutzt werden, nicht beim Wechseln auf eine andere Strasse.

- `void vBeendeGrafik();`

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

  Beachte: Die Funktionen der Grafikbibliothek arbeiten nur mit den übergebenen Werten, kennen also keine anderen Daten Ihres Projektes. Die Werte werden beim Aufruf aber auf syntaktische und semantische Plausibilität geprüft. Das bedeutet:

  a) Zahlenwerte müssen in einem sinnvollen Wertebereich liegen.

  b) Die Relative Position muss innerhalb [0,1] liegen. Auch durch Rundungsfehler darf das Intervall nicht verlassen werden.

  c) Die Namen dürfen nur Buchstaben, Ziffern und _ enthalten, insbesondere keine Blanks.

  d) Fahrzeuge können nur auf Wegen gezeichnet werden, die vorher als Strasse definiert wurden. Achten Sie auf die exakt gleiche Schreibweise der Namen.

Um die Abfertigung der PKWs und Fahrräder zu simulieren, erzeugen Sie nun zwei Wege (Länge = 500.0 km), auf die Sie jeweils ein parkendes und fahrendes Fahrzeug mit unterschiedlichen Geschwindigkeiten setzen. Wählen Sie auf einem Weg eine Geschwindigkeitsbegrenzung. 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.

7.5.4 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()` mit den Daten aus `vAufgabe_5()`. Wahrscheinlich kommt es zu einer Fehlermeldung bei der Abfertigung des Weges, da der Iterator über die Fahrzeuge nach dem Löschen bzw. Umsetzen eines Fahrzeugs nicht mehr definiert ist.

2. Probleme, die auftreten können, sind Iteratoren, die nach dem Löschen nicht mehr existieren, aber erhöht werden sollen (siehe Punkt 1) oder die Nichtabfertigung von Fahrzeugen, die in der Liste durch Umsetzen von Elementen nach hinten oder vorne 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.

   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.

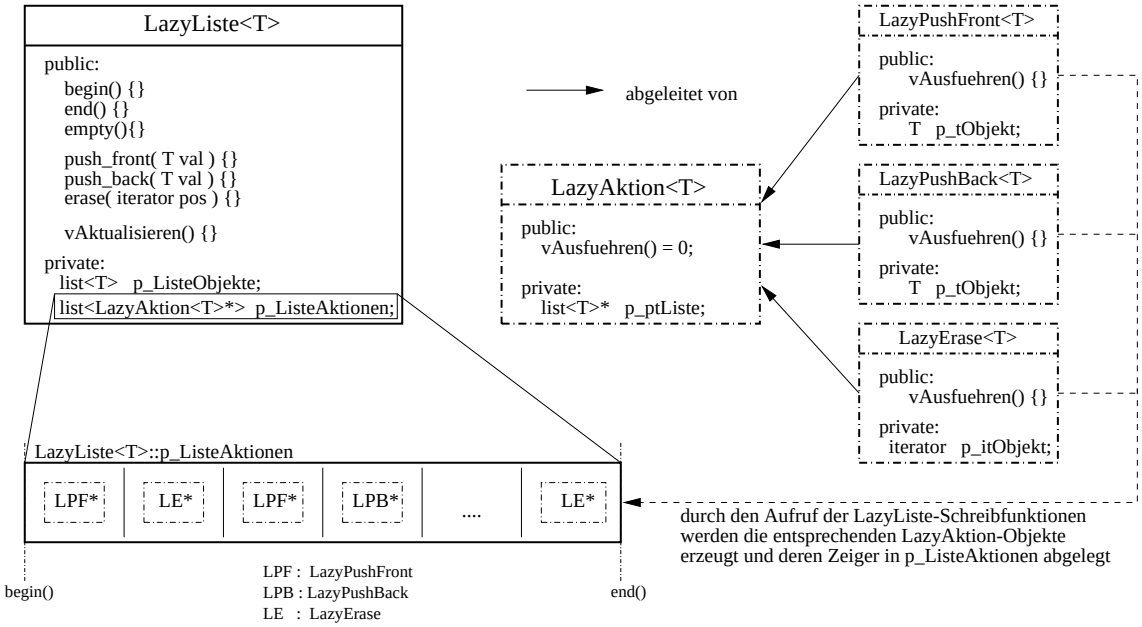


Abbildung 7.6: Prinzipielle Funktionsweise der LazyListe

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 7.6 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 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>`.

Sollten beim Einbinden von LazyListe.h diverse Fehlermeldungen mit undefinierten Variablen auftauchen, liegt dies an einer Default-Parametereinstellung in Visual Studio 2017. Bei den Projekteigenschaften muss unter `C/C++` → `Sprache` der Konformitätsmodus ausgeschaltet werden (Nein).

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 in der Liste richtig umgesetzt und am Ende des Weges aus der Liste gelöscht werden.

7.6 Aufgabenblock 3: Simulation des Verkehrssystems

7.6.1 Motivation

Im ersten Teil dieses Blocks soll ein einfaches algorithmisches Problem implementiert werden: Die Fahrzeuge sollen sich auf den Wegen, die ein Überholverbot haben, nicht mehr überholen. Bisher können nur einzelne, nicht zusammenhängende Wege erzeugt werden und die Fahrzeuge nur auf einem 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 Kreuzungen eingeführt. 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

7.6.2 Lernziele

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

7.6.3 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 Leeres Projekt* 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 7.3.2).
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.

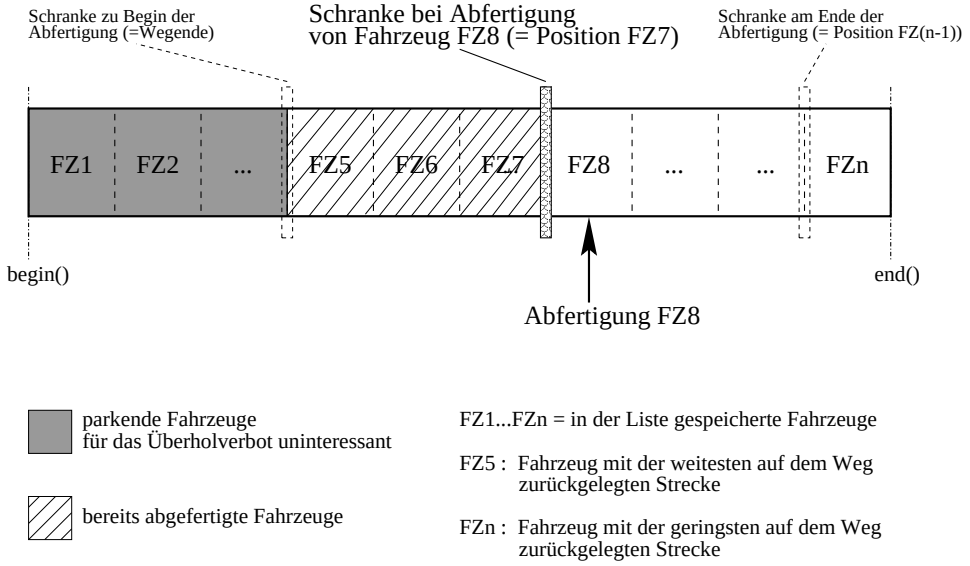


Abbildung 7.7: Schranke zum Zeitpunkt der Abfertigung von FZ8

Schreiben Sie dazu eine Hauptfunktion `vAufgabe_7()`, die zwei parkende Fahrzeuge (ein Fahrrad und ein PKW) auf einen Weg stellt und den Weg abfertigt (der zweite Weg dient hier nur der grafischen Darstellung). Nach der Hälfte der Zeit soll ein weiteres Fahrzeug vom Weg (parkend) angenommen werden. Ihr Eventhandler sollte so lange abfertigen, dass alle Fahrzeuge losfahren und das Ende des Weges erreichen. 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). Ergänzen Sie entsprechend den Konstruktor von **`Weg`** um einen Parameter für das Überholverbot. Falls dieser Parameter nicht angegeben wird, soll standardmäßig davon ausgegangen werden, dass dieser Weg schlecht ausgebaut ist.

   Tipp: Führen Sie zusätzlich zur Weglänge eine *„virtuelle Schranke“* ein, die jeweils auf die Position des aktuell auf diesem Weg abgefertigten Fahrzeugs gesetzt wird. Das nächste Fahrzeug darf dann nicht weiter als bis zu dieser Schranke fahren. Schreiben Sie eine getter-Funktion für die virtuelle Schranke, die berücksichtigt, ob es ein Überholverbot auf diesem Weg gibt. Ersetzen Sie dann in den Abfragen zur Berechnung der Reststrecke Weglänge durch den Aufruf dieser Funktion. Machen Sie sich klar, dass Sie wegen des speziellen Aufbaus der Liste (s. Abbildung 7.7) 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.

7.6.4 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. jeder Weg kennt seinen direkten Rückweg und jeder dieser Wege 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 7.8 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

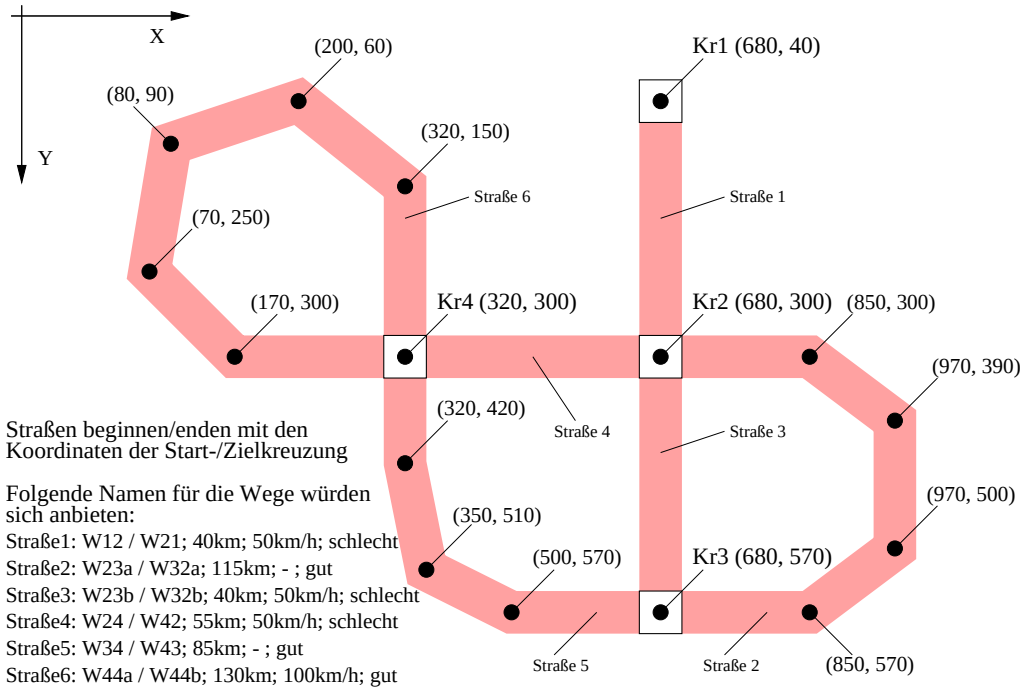


Abbildung 7.8: Koordinaten des Verkehrssystems

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]

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 7.8. 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.

7.6.5 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. Kopieren Sie dazu diese Datei in das Projektverzeichnis und machen Sie sie dem Projekt bekannt. 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 <NameQ> <NameZ> <NameW1> <NameW2> <Länge> <Geschw> <Überholverbot>
   ```

```
PKW <PKW-Daten> <NameS> <Zeitpunkt des Losfahrens>
FAHRRAD <Fahrrad-Daten> <NameS> <Zeitpunkt des Losfahrens>
```

Dabei gelten folgende Wertekonventionen:

```
NameQ:  Name der Quellkreuzung
NameZ:  Name der Zielkreuzung
NameW1: Name des Weges von der Quell- zur Zielkreuzung
NameW2: Name des Weges von der Ziel- zur Quellkreuzung
Geschwindigkeitsbegrenzung:
    enum: 1(innerorts) 2(Landstraße) 3(Autobahn)
Überholverbot:
    bool: 0(falsch) oder 1(wahr)
NameS:  Name der Startkreuzung
```

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 für alle Fehler entsprechende Ausnahmen geworfen werden und korrigieren Sie jeweils die Fehler in Ihrer Kopie von *Simu.dat*. Alle Fehler sollen vom Programm erkannt werden. Korrigieren Sie keine Fehler, bevor diese vom Programm erkannt wurden.

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

Tabellenverzeichnis

Index

RWTH Aachen
Institute for Automation of Complex Power Systems

Univ.-Prof. Antonello Monti, Ph.D.
Mathieustr. 10
52074 Aachen
Germany

Tel.: +(49)-241-8049700
Fax: +(49)-241-8049709

http://www.acs.eonerc.rwth-aachen.de

E-Mail:

RWTH Aachen
Lehrstuhl für Inetgrierte Digitale Systeme und Schaltungsentwurf

Univ.-Prof. Dr.-Ing. Tobias Gemmeke
Mies-van-der-Rohe-Str. 15
52074 Aachen
Germany

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

http://www.ids.rwth-aachen.de

PI2Betreuung@ids.rwth-aachen.de