Praktikum Informatik

Contents

Willkommen zum Praktikum Informatik

Übersicht der Themen

- Einführung in Jupyter
- Grundlagen der Programmiersprache C++
- Namensbereiche in C++
- Klassen
  - Übungsaufgaben zu Klassen
- Vererbung
  - Übungsaufgaben zu Vererbung
- Smart Pointer
  - Übungsaufgaben zu Smart Pointern
- Container-Klassen
  - Übungsaufgaben zu Container-Klassen
- Exception Handling
  - Übungsaufgaben zu Exception Handling
- Templates
- Eingabeoperator
- Typumwandlung

Einführung in Jupyter

- Zellen
- Kernel
- Operatorenüberladung
- Shortcuts

In diesem Praktikum wird Jupyter verwendet, um Ihnen die grundlegende Funktionsweise von C++ näher zu bringen. Weder das Beschäftigen mit den Jupyter-Dateien, noch die Bearbeitung der Aufgaben in den Jupyter-Dateien ist verpflichtend. Ziel der Jupyter-Dateien ist es, Ihnen die Sprache C++ näher zu bringen, falls Sie bis jetzt wenig oder gar keine Erfahrung mit C++ haben.

Die Funktionsweise von Jupyter unterscheidet sich an einigen Stellen recht stark von dedizierten Entwicklungsumgebungen, die meistens einen deutlich größeren Funktionsumfang zur Fehlerbehebung und Organisation umfangreicher Softwareprojekte aufweisen. Für die hier vorgestellten Beispiele ist dieser allerdings nicht unbedingt notwendig; darüberhinaus entfällt so anfangs die individuelle Installation und Konfiguration, bei der sich erfahrungsgemäß häufig Probleme ergeben können. Jupyter kann von Ihnen einfach im Browser verwendet werden, wobei eine Instanz auf den Servern der RWTH läuft. Es ist ebenfalls möglich, Jupyter lokal zu installieren, was für die Bearbeitung dieses Praktikums aber nicht notwendig ist. Im folgenden soll Ihnen ein kurzer Überblick darüber gegeben werden, wie Sie mit Jupyter arbeiten können.

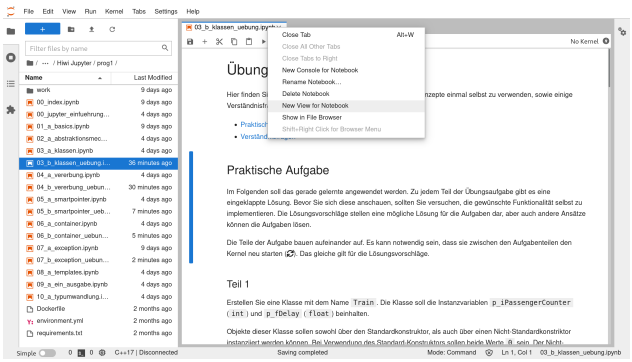
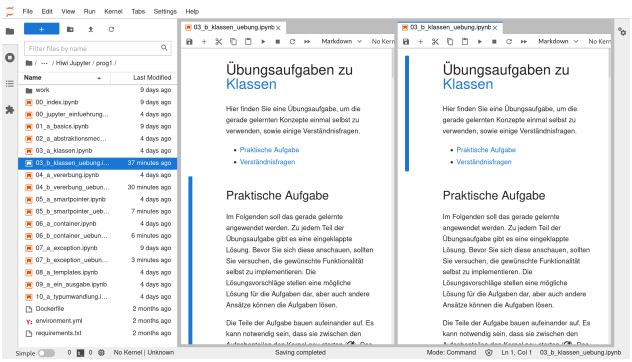
Sollten Sie bereits eine funktionierende Installation von Visual Studio oder einer anderen C++ Entwicklungsumgebung haben, ist es natürlich ebenfalls möglich, die Übungsaufgaben darin zu lösen.

Zellen

In Jupyter-Notebooks wird Text und Code in Zellen aufgeteilt. Diese sind prinzipiell frei verschiebbar und können auch kopiert oder gelöscht werden, jedoch sind Reihenfolge und Anzahl in den folgenden Notebooks aus naheliegenden Gründen bereits passend gewählt und müssen im Allgemeinen nicht von Ihnen verändert werden. Zellen können (mit Markdown formatierten) Text oder Code beinhalten. Die Auswahl geschieht je Zelle über ein Dropdown-Menü in der Toolbar.

- Ausführen einer einzelnen Zelle: -Button. Dabei ist zu beachten, dass der Code in einmal ausgeführten Zellen innerhalb des gesamten Notebooks seine Gültigkeit behält, sodass beispielsweise dort definierte Variablen oder Methoden in weiteren Zellen verwendet werden können. Die Reihenfolge der Ausführung wird in kleinen Ziffern links neben der Zelle angegeben.
- Ausführen aller Zellen eines Notebooks: im Menü: Run Run All Cells oder -Button (startet vorher den Kernel neu, was also insbesondere die Werte aller Variablen zurücksetzt)
- pro Zelle kann nur eine Methode implementiert werden, sonst kommt es zu Fehlermeldungen wie `function definition is not allowed here`. In den Jupyter-Files mit den Übungsaufgaben ist daher angegeben, in welcher Zelle welche Methode einer Klasse definiert werden soll.
- Zellen können durch einen Klick auf den blauen Balken links ein- und ausgeblendet werden. Die Lösungen zu den Übungsaufgaben sind im Normalfall ausgeblendet.
- Um zu verstehen, wie sich C++ verhält, wenn Code in einer anderen Reihenfolge ausgeführt wird, muss in Jupyter der Code nicht umgestellt werden. Es genügt, die auszuführende Zelle zu markieren und den -Button zu drücken.

- Die Verwendung von Namespaces sollte aufgrund der fehlenden Aufteilung in Dateien vermieden werden, da dies schnell unübersichtlich werden könnte und erfahrungsgemäß häufig zu wenig hilfreichen Fehlermeldungen führen kann. Unter anderem bedeutet das auch, dass die durchaus verbreitete Zeile `using namespace std;` wegfällt und stattdessen der Präfix `std::` vor bestimmte Ausdrücke wie `string`, `cout` usw. geschrieben wird.
- In den Notebooks, bei denen Sie eine eigene Implementierung erstellen sollen, kann es hilfreich sein, sich die Aufgabenstellung und später die Lösung nebeninander anzuzeigen, anstatt immer nach oben und unten zu scrollen. Dafür wählen Sie nach einem Rechtsklick auf den aktuellen Tab die Option "New View for Notebook" aus, was eine neue Ansicht desselben Notebooks erstellt. Diese beiden Ansichten bleiben im Hintergrund synchronisiert, können aber unabhängig voneinander durchgescrollt werden.

New View for Notebook	Ergebnis
	

Code

```
// Beispiel: Führen Sie diese Zelle aus!

#include <iostream>

std::cout << "Hello World" << std::endl;
```

Kernel

Hin und wieder kann es notwenig sein, den Kernel neu zu starten (-Button). Dies ist vor allem dann erforderlich, wenn Code in einer falschen Reihenfolge ausgeführt wird. Da Jupyter dazu ermuntert, Zellen nicht nur linear von oben nach unten auszuführen, kann dies häufiger auftreten.

Operatorenüberladung

In dem verwendeten Kernel gibt es in Zusammenhang mit der Operatorenüberladung einen Fehler. Daher ist in Jupyter-Dateien ein zusätzlicher Schritt notwendig:

- normalerweise würde eine Operatorenüberladung so aussehen:

Code

```
std::ostream & operator<<(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}
```

- In Jupyter ist das so nicht möglich. In den entsprechenden Jupyter-Files finden Sie daher folgendes:

Code

```
#define OPERATOR operator<<

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR
```

Hier wird also temporär eine Präprozessorkonstante definiert, die in der folgenden Funktionsdefinition verwendet werden kann. Für die Verwendung des auf diese Weise überladenen Operators ergeben sich keinerlei Unterschiede.

Shortcuts

Folgende Tastenkombinationen können beim Arbeiten mit Jupyter hilfreich sein:

- `STRG + ENTER`: führt den Code aus und bleibt in der Zelle
- `SHIFT + ENTER`: führt den Code aus und springt in die nächste Zelle bzw. legt eine neue Zelle an

Grundlagen der Programmiersprache C++

- Lexikalische Konventionen
  - Kommentare
  - Bezeichner
  - Schlüsselwörter
  - Literale
- Datentypen
  - Fundamelntale Datentypen
  - Zusammengesetzte Datentypen
  - Konstanten und Aufzählungen
- Grundlegende Konzepte
  - Deklaration und Definition
  - Gültigkeitsbereich
  - Lebensdauer und Speicherklassen
- Ausdrücke / Operatoren
- Anweisungen
  - `if-else`-Anweisung
  - `switch`-Anweisung
  - `while`-Anweisung
  - `for`-Anweisung
  - `do-while`-Anweisung
  - Sprunganweisungen
  - `extern`-Anweisung
- Funktionen
- Präprozessor
  - `include`
  - `define`
  - Include guards

Im Folgenden wird ein Überblick über die Programmiersprache C++ gegeben.

Lexikalische Konventionen

Einer der ersten Verarbeitungsschritte, die der Compiler durchführt ist die Zerlegung der Eingabe in sogenannte Token. Während die Eingabe aus einzelnen Buchstaben besteht, sind Token das Äquivalent zu Wörtern. In C++ gibt es die folgenden Token:

- Bezeichner (Identifier)
- Schlüsselwort (Keyword)
- Literal
- Operator
- Trennzeichen (Separator)

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

Kommentare

Ein Kommentar beginnt mit den Zeichen `/*` und endet mit `*/` und dient zum Erläutern und Beschreiben des Codes. Kommentare werden vom Compiler ignoriert.

Code

```
/*
Diese Schreibweise für einen Kommentar ermöglicht es den Text
über mehrere Zeilen zu verteilen.
*/
```

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

Code

```
// auskommentierte Zeile
int i;  // Zähler
```

Bezeichner

Ein Bezeichner 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.

Schlüsselwörter

In C++ sind u.a. folgende Schlüsselwörter reserviert und nicht anderweitig benutzbar:

Code

```
alignof  const_cast   goto      public           typedef
alignas  constexpr    friend    protected        try
and      continue     if        register         typeid
and_eq   decltype     inline    reinterpret_cast typename
asm      default      int       return           union
auto     delete       long      short            unsigned
bitand   do           mutable   signed           using
bitor    double       namespace sizeof           virtual
bool     dynamic_cast new       static           void
break    else         noexcept  static_assert    volatile
case     enum         not       static_cast      wchar_t
catch    explicit     not_eq    struct           while
char     export       nullptr   switch           xor
char16_t extern       operator  template         xor_eq
char32_t false        or        this
class    final        or_eq     thread_local
compl    float        override  throw
const    for          private   true
```

Bei Schlüsselwörtern wird wie bei allen Bezeichnern Groß- und Kleinschreibung unterschieden. Bei *American Standard Code for Information Interchange* (ASCII)-Texten werden folgende Zeichen zur Interpunktion verwendet:

Code

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

Außerdem folgende Zeichenkombinationen als Operatoren:

Code

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

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)`.

Zeichenkonstanten

Ein Zeichen (*character*), eingeschlossen in einfache Hochkommata ('x') ist eine Zeichenkonstante, 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)

`\r` Zeilenende (Carriage Return)

`\'` 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. entsprechenden Ausdrücken muss entweder die Exponentenschreibweise verwendet werden oder es muss ein Punkt enthalten sein, sonst werden Sie als Integer-Konstante interpretiert und es kann bei Ausdrücken zu unerwünschtem Verhalten kommen. So ergibt z.B. der Ausdruck `1/2` als Ergebnis `0`, da dies eine Integer-Division ist. Soll eine Fließkommadivision durchgeführt werden, so muss mindestens einer der Operanden den entsprechenden Typ haben, im Beispiel führt `1/2.0` oder `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 kann daher 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'` abgebildet.

Datentypen

Datentypen haben einen hohen Stellenwert in C++. Sie werden dazu benutzt, Problemstellungen genau abzubilden. Zunächst werden die fundamentalen Datentypen beschrieben, von denen alle anderen Datentypen abgeleitet sind.

Fundamentale Datentypen

Zeichen (*character*)

Code

```
char           // Zeichen (als Zahl 1Byte mit oder ohne Vorzeichen, je nach Compiler)
signed char    // Zeichen (als Zahl 1Byte mit Vorzeichen: -128 ... +127)
unsigned char  // Zeichen (als Zahl 1Byte ohne Vorzeichen: 0 ... 255)
```

Ob der `char`-Typ vorzeichenbehaftet ist oder nicht, ist nicht standardisiert. In sehr vielen Implementationen ist er aber `signed`.

Integer

```
short, short int   // mit Vorzeichen (2Byte: -32768 ... +32767)
int                // mit Vorzeichen (meist 4Byte: -21474883648 ... +21474883647)
long, long int     // mit Vorzeichen (4Byte: -21474883648 ... +21474883647)
```

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
```

In den allermeisten 32-Bit-Implementierungen, gelten die in angegebenen Werte. Bei 64-Bit-Implementierungen sind die `long`-Datentypen oft 8 Bit breit mit entsprechend größerem Wertebereich. Um unabhängig von einer speziellen Implementierung zu sein, sollte man sich bei beim Programmieren nicht auf die angegebenen Werte verlassen. Wenn es für die Implementierung wichtig ist, die Grenzen zu beachten, sollten die Konstanten aus dem Header `<numeric_limits>` verwendet werden, die in jeder Situation die korrekten Grenzen zurückgeben.

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	$1.4*10^{-45}$...$3.4*10^{+38}$
`double`	8	$4.94*10^{-324}$...$1.8*10^{+308}$
Boolean		
`bool`	1	*true, false*

Umwandlungen zwischen den Standard-Datentypen werden normalerweise implizit durchgeführt. Darüber hinaus können Umwandlungen erzwungen werden (siehe auch Typumwandlung):

```
int x = 5;
double y = double(x);  // double y = x auch möglich
```

Void

Der Datentyp `void` ist ein fundamentaler Datentyp, von dem keine Objekte, Arrays aus Objekten oder Referenzen erstellt werden können. `void` wird benutzt, um anzugeben, dass eine Funktion keinen Rückgabewert hat.

```
void f()
{
    // kein return
    // Funktion liefert keinen Wert.
}
```

Boolean

Logische Werte - sogenannte Wahrheitswerte - werden mit dem Typ `bool` dargestellt, der die Werte `true` und `false` annehmen kann.

Code

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

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

auto

Bei Deklaration von Variablen mit gleichzeitiger Initialisierung kann man den Datentyp auch anhand des Typs der Initialisierungsvariablen festlegen. Anstelle des Datentyps schreibt man dann `auto`.

Code

```
auto bIsVar = true;  // Variable vom Typ bool
auto i = 5;          // Variable vom Typ int
auto d = 2.0;        // Variable vom Typ double
```

Dieses Verhalten hat einige Vorteile:

- Ändert sich der Typ der rechten Seite, ändert sich auch der Datentyp der linken Seite
- Die Initialisierung wird vereinfacht

Diese Festlegung des Datentyps mit `auto` sollte man dennoch mit Sorgfalt verwenden. Sinnvolle Anwendungsmöglichkeiten sind lokale Variablen, Schleifenvariablen oder Zwischenwerte für Containerelemente.

Zusammengesetzte Datentypen

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

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

Referenzen

Eine Referenz ist ein weiterer Name für ein bestehendes Objekt. Das heißt, bei der Erstellung einer Referenz wird das Objekt nicht kopiert und Änderungen an der Referenz ändern auch das ursprüngliche Objekt.

Für einen Datentyp `Type` ist `Type&` eine Referenz auf ein Objekt dieses Typs. Zu beachten ist, dass Referenzen immer initialisiert werden müssen:

Code

```
int a;
// int& b;  // Fehler: keine Initialisierung
int& c = a; // Gültig.
c = 7;      // Auch a hat jetzt den Wert 7
```

Auf welches Objekt eine Referenz zeigt, kann nicht geändert werden. Referenzen spielen besonders bei Funktionsargumenten und Rückgabewerten eine wesentliche Rolle.

Pointer (Zeiger)

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

Code

```
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:

- Dereferenzieren (`*`) Zugriff auf das Objekt, auf das der Pointer zeigt.

- *Address of* (`&`) Erstellung eines Pointers auf ein Objekt über seine Adresse.

Beispiel:

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

Um darzustellen, dass ein Pointer (momentan) auf kein Objekt zeigt, kann ihm `nullptr` zugewiesen werden. Das ist für jeden Pointer möglich.

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

Die Konstante `NULL`, wie aus C bekannt, sollte für diesen Zweck nicht benutzt werden. **Das Dereferenzieren von einem `nullptr` ist ein Programmierfehler mit undefinierten Auswirkungen.**

Es gibt zwei wichtige Unterschiede zwischen Pointern und Referenzen:

- Pointern können mehrmals neue Objekte zugewiesen werden (Beispiel oben).
- Pointer können den speziellen Wert `nullptr` haben.

In aktuellen Versionen von C++ gibt es Smartpointer, die einige Vorteile gegenüber einfachen Pointer haben (siehe Smartpointer).

Felder

Felder sind ein- oder mehrdimensionale geordnete Ansammlungen eines Datentyps, z.B. ein Feld aus Integern.

Mit der folgenden Deklaration wird eine Feld aus fünf Fließkommazahlen erstellt:

Code
```
double arr[5];
```

Über den Index-Operator wird auf einzelne Elemente zugegriffen:

Code
```
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 ein Element
a[9] = 0.9;
k[3] = 125;
a[3] = k[3];
```

Mehrdimensionale Felder sind Felder, die wieder Felder als Elemente enthalten.

Code
```
// Feld mit 10 Elementen vom Typ "Array mit 20 Element vom Typ char"
// äquivalent: Matrix mit 10 x 20 char-Elementen
char matrix[10][20];
matrix[0][0]  = 'a';  // äußeres Feld zuerst indiziert
matrix[0][10] = 'b';
matrix[9][0]  = 'c';
```

Die Indizes eines mit n dimensionierten Feldes laufen immer von `0` bis `n - 1`. Die Indizierung wird nicht überprüft. **Zugriff auf ein Feld außerhalb seiner Grenzen ist ein Programmierfehler mit undefinierten Auswirkungen.**

C++ stellt in der Standardbibliothek Container zur Verfügung, bei denen eine Prüfung der Grenzen erfolgt.

Der Zugriff auf Felder kann auch über Pointer erfolgen. Dabei ist `a[i]` äquivalent zu `*(a+i)`.

Code
```
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 wird insbesondere bei der Übergabe von Feldern in Funktionen benutzt.

Speicherverwaltung

Im Gegensatz zu anderen Sprachen liegt die Speicherverwaltung bei C++ in der Verantwortung des Programmierers. Es gibt einige Anforderungen an die Speicherverwaltung:

- Alle erstellten Objekte müssen genau einmal gelöscht werden.
- Beim Zerstören müssen die Ressourcen (Speicher, offene Dateien etc.) freigegeben werden, die ein Objekt besitzt.

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 ist bzw. deren Art und Anzahl erst zur Ausführung des Programms bekannt ist. Dafür gibt es dynamische Speicherverwaltung.

Klassische dynamische Speicherverwaltung mit `new/delete` Klassisch gibt es dafür die Operatoren `new` und `delete`. Objekte, die durch `new` angelegt wurden, befinden sich im Freispeicher. Der Programmierer ist selbst für die Speicherverwaltung verantwortlich, d.h. Objekte, die mit `new` explizit angelegt werden, müssen auch wieder explizit mit `delete` zerstört werden.

Beispiel für das Erzeugen/Zerstören eines Objekts:

Code
```
int* p = nullptr;
delete p;     // Zerstören mit nullptr erlaubt, aber ohne Wirkung;
p = new int(5);  // Neuen Integerwert erstellen
delete p;     // Zerstören des Integerwertes (Wert des Zeigers unverändert)
p = nullptr;  // Zeiger wieder auf nullptr setzen
```

Beispiel für das Erzeugen/Zerstören eines Feldes von Objekten:

Code
```
char* p = new char [10]; // Feld mit 10 char-Werten
p[3] = 'c';              // Zugriff auf das 4.Zeichen
delete [] p;             // Löschen des Speichers für die Integerwerte
p = nullptr;             // Zeiger wieder auf nullptr setzen
```

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 einer anderen unerwünschten Programmfunktion führt.

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

Die Speicherverwaltung wird komplexer, sobald Referenzen und Pointer verwendet werden. Im folgenden Beispiel gibt die Funktion `badUse` eine ungültige Referenz zurück. Das Objekt auf dem Stack von `badUse` wurde zerstört, aber die Referenz existiert noch. Zugriffe auf die Referenz führen dann zu Speicherverletzungen (so genannte *dangling pointer*). **Daher ist darauf zu achten, dass Objekte länger als ihre Referenzen leben.**

Code
```
string& badUse() {
    string s = "schlecht";
    return s;
}
void f() {
    // Speicherverletzung: referenziertes Objekt wurde schon zerstört
    //string& s = badUse();
    //s.size();
}
```

Um den String sicher zurückzugeben, kann er als Wert zurückgegeben werden. Dabei wird aber eine Kopie erstellt und nicht mit dem ursprünglichen Objekt weitergearbeitet.

Code
```
string BadUse() {
    string s = "schlecht";
    return s;
}
void f() {
    string s_stack = badUse();
    string& s_ref = s_stack;  // OK. Objekt lebt lang genug.
    s_ref.size();
}
```

Strukturen

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

Code

```
// Beispiel für eine Strukturdeklaration:
struct BirthDate
{
    string name;
    int year, month, day;
};

// Definition eines Objektes:
BirthDate mm;
mm.name  = "Max Mueller";
mm.year  = 1999;
mm.month = 2;
mm.day   = 29;
```

Strukturen sind in C++ tatsächlich Klassen mit dem einzigen Unterschied, dass der Zugriff auf die Elemente standardmäßig nicht eingeschränkt (siehe auch Beschreibung Klassen) ist. Mit dem Punktoperator kann auf die einzelnen Elemente der Struktur zugegriffen werden. Bei Pointern kann die Dereferenzierung und der Punktoperator zum Pfeiloperator zusammengezogen werden:

Code

```
BirthDate mm;
BirthDate *pmm = &mm;

mm.name;        // Diese Zugriffe sind gleichwertig
(*pmm).name;
pmm->name;
```

Konstanten und Aufzählungen

Benutzerdefinierte Konstanten sind Werte, die einmal gesetzt und dann nicht mehr geändert werden. 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 nicht 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:

Code

```
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. Wenn der **Zeiger** `const` ist, steht das Schlüsselwort **rechts** vom `*`. Wenn der **Element** `const` ist, steht das Schlüsselwort **links** vom `*`.

Beispiel:

Code

```
char* c0;  // Zeiger auf char

char* const c1;  // Konstanter Zeiger auf char

char const* c2;  // Zeiger auf konstanten char
const char* c3;  // konstanter Zeiger auf (nicht konstanten) char

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

Aufzählungstyp

Mit `enum` wird ein neuer Datentyp definiert, der nur die angegebenen Werte annehmen kann:

Code

```
enum class Color
{
    red,
    yellow,
    blue
};
```

Intern ist der Datentyp als Integer repräsentiert. Bei Bedarf kann ein bestimmter Datentyp für die Repräsentation ausgewählt werden:

Code

```
enum class Color
    : byte
{
    red,
    yellow,
    blue
};
```

Wenn man keine Werte angibt, beginnt die Aufzählung bei 0 und wird jeweils um 1 erhöht. Optional können die Werte definiert werden, wobei fehlende Werte den nächsthöheren Wert erhalten (z.B. hier `C02 = 2`):

Code

```
enum class CentCoin
{
    C01 = 1,
    C02,
    C05 = 5,
    C10 = 10,
    C20 = 20,
    C50 = 50
};
```

Variablen werden wie folgt angelegt und zugewiesen:

Code

```
Color favColor = Color::red;
CentCoin aCoin;
aCoin = CentCoin::C50;
```

Um mit diesen Variablen Berechnungen durchzuführen, müssen sie erst zu einem Integer-Datentyp konvertiert werden (siehe auch Typumwandlung):

Code

```
Color favColor = Color::red;
int colorCode = (int)(Color::red);

CentCoin e = CentCoin::C05;
e = (CentCoin)((int)e + 5); // e = CentCoin::C10;
```

Die Vorteile gegenüber Integer-Konstanten sind:

- Der Datentyp überprüft, dass nur definierte Werte benutzt werden (leider nicht automatisch).
- Falls die Repräsentation als Integer gefordert ist, ist dies nur explizit möglich.
- Größere Übersichtlichkeit, da alle Werte an einem Ort definiert sind.
- Gleich benannte Werte unterschiedlicher enums kollidieren nicht:

Code

```
enum class Enum0
{
    Value = 0,
};
enum class Enum1
{
    Value = 0,
};
Enum0 e0 = Enum0::Value;
Enum0 e1 = Enum1::Value;
```

Präprozessorkonstanten

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

Code

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

Die Verwendung von `const` ist jedoch vorzuziehen.

Grundlegende Konzepte

C++ definiert einige Konzepte die hier kurz beschrieben werden.

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

Beispiele für Definitionen:

Code

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

Beispiele für reine Deklarationen:

Code

```
extern int a;
int f(int);
struct S;
```

Gültigkeitsbereich

Ein Gültigkeitsbereich definiert, in welchen Teilen eines Quelltextes ein Name sichtbar, d.h. verwendbar ist. In C++ gibt es folgende Arten von Gültigkeitsbereichen:

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

Global: Ein Name, deklariert außerhalb aller Blöcke und Klassen, 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 (`::`) 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.

Code

```
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
}
```

Lebensdauer und Speicherklassen

Objekte haben eine Lebensdauer, die direkt nach der Erstellung beginnt und am Ende des Blocks der Erstellung bzw. direkt vor dem Destruktor endet. Sie dürfen nur innerhalb ihrer Lebensdauer verwendet werden.

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. Zur Laufzeit werden sie auf einer Datenstruktur abgelegt, die als Stapelspeicher bzw. *Stack* bezeichnet wird.

Code

```
struct S {};

void func(S s0)  // Lebensdauer von s0 beginnt
{
    // Lebensdauer von s1 beginnt
    S s1 = s0;
    // Lebensdauer von s1 endet
}
// Lebensdauer von s0 endet
```

Statischer Speicher: Globale Variablen leben von Programmbeginn bis Programmende. Sie werden in der Reihenfolge ihrer Definition erstellt.

Code

```
int s0 = 10;      // Lebensdauer von s0 beginnt
int main()
{
    int s1 = s0 + 5;  // Lebensdauer von s1 beginnt
    return 0;       // Lebensdauer von s1 endet
}
// Lebensdauer von s0 endet
```

Im Beispiel wird erst die Variable `s0` erstellt, dann `s1`. Die Variable s1 lebt während der `main`-Funktion, s0 auch während des Restes des Programms. s1 wird beim Verlassen der Funktion, s0 erst am Ende des Programms zerstört, d.h. erst `s1` und dann `s0`. Auch mit `static` gekennzeichnete Variablen in Funktionen und Klassen haben diese Lebensdauer.

Freispeicher/dynamischer Speicher: Diese Speicherklasse wird benutzt, wenn Speicherplatz während der Programmausführung angefordert wird. Diese Art Speicher wird auch *dynamischer Speicher* genannt. Die Benutzung dieser Speicherklasse wird unter SmartPointer beschrieben.

Ausdrücke / Operatoren

C++ beinhaltet eine Vielzahl von Operatoren, die in Ausdrücken Werte verändern können. Die folgende Tabelle 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 fundamentale Typen zu. Zusätzlich kann man Bedeutungen für Operatoren definieren, die auf benutzerdefinierte Typen angewendet werden (siehe Operatoren überladen).

- `Klassenname` ist der Name einer Klasse,
- `Element` ist ein Elementname,
- `Objekt` ist ein Ausdruck, der ein Objekt einer Klasse ergibt,
- `Zeiger` ist ein Ausdruck, der einen Zeiger ergibt,
- `Ausdruck` ist ein Ausdruck und
- `Lvalue` ist 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)`
Postinkrement	`Lvalue++`
Postdekrement	`Lvalue--`
Typidentifikation	`typeid(Typ)`
Laufzeit-Typinformation	`typeid(Ausdruck)`
zur Laufzeit geprüfte Konvertierung	`dynamic_cast<Typ>(Ausdruck)`
zur Übersetzungszeit geprüfte Konvertierung	`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 Initialisierung)	`new Typ(Ausdrucksliste)`
Erzeugung (Plazierung)	`new(Ausdrucksliste) Typ`
Erzeugung(Plazierung und Initialisierung)	`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`

Bitweises Exklusiv-Oder	`Ausdruck ^ Ausdruck`

Bitweises Oder	`Ausdruck \`

Logisches Und	`Ausdruck && Ausdruck`
Logisches Oder	`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 \`
Exklusiv-Oder und Zuweisung	`Lvalue ^= Ausdruck`

Ausnahme werfen	`throw Ausdruck`

Komma (Sequenzoperator)	`Ausdruck, Ausdruck`

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 unspezifiziert, ob bei dem Code

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

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

Anweisungen

`if-else`-Anweisung

Bedingte Anweisung:

Code
```
if ( expression ) statement_1
```

bzw. mit `else`-Teil:

Code
```
if ( expression )
    statement_1
else
    statement_2
```

bzw. als Statement mit ?-Operator:

Code
```
( expression ) ? statement_1 : statement_2;
```

Bei dieser Anweisung wird zunächst die Bedingung (*expression*) ausgewertet. Ist das Ergebnis `true`, so wird *statement_1* ausgeführt, ansonsten wird in den beiden letzten Varianten *statement_2* ausgeführt.

Beispiel:

Code

```
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

z = (a>b)?a:b; // z = max(a,b)
```

`switch`-Anweisung

Bedingte Anweisung mit Ausführung abhängig vom Wert eines Ausdrucks.

Code

```
switch ( expression )
{
    case constant_expression_1:
        statements
    case constant_expression_2:
        statements
    case default:
        statements
}
```

Für *expression* sind nur int- oder char-Ausdrücke zulässig. Normalerweise besteht eine `switch`-Anweisung aus einem Block, mehreren Labels und Anweisungen. Wichtig ist, dass es sich um einen konstanten Ausdruck (*constant_expression*) hinter dem Schlüsselwort `case` handelt (keine Verwendung von Variablen, Funktionsaufrufen etc.). Es können beliebig viele `case` verwendet werden. Der optionale `case default` wird aufgerufen, wenn keiner der anderen Fälle zutrifft. Für benutzerdefinierte Datentypen ist `switch-case` nicht benutzbar. Dann wird eine Kette von `if-else`-Anweisungen benutzt.

Beispiel:

Code

```
char c = 'u';

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;
}
```

Bei `case` handelt es sich um ein Label, von dem 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 (*fall-through*). Im Regelfall wird daher jeder Block hinter einem `case`-Label durch ein `break`-Statement abgeschlossen.

`while`-Anweisung

Bei dieser Schleifenstruktur wird zunächst der Ausdruck bewertet. Ist dieser wahr, 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 ist. Dann wird mit der Anweisung, die hinter *statement* steht fortgefahren.

Code

```
while ( expression ) statement
```

Beispiel:

Code

```
int b=5;
while (b-- > 0)
{
...
}
```

`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.

Code

```
for ( expression_1; expression_2; expression_3 ) statement
```

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

Code

```
// 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:

Code

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

`for`-Schleife auf Range-Basis

Soll über einen Datentyp mit mehreren gleichen Elementen z.B. Felder iteriert werden, können range-basierte `for`-Schleifen genutzt werden. Die Syntax lautet:

Code

```
for ( range_declaration : range_datatype ) statement
```

Beispiel mit einem `Feld`:

Code

```
int v[] = {0, 1, 2, 3, 4, 5};

for (auto &i : v)  // Schleife über alle Elemente in v, das Schleifenelement wird als Referenz über i angesprochen
    cout << i << ' ';

// Ausgabe: 0 1 2 3 4 5
```

Bei `auto` werden die Elemente kopiert, bei `auto&` referenziert.

`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 wahr, so wird die bedingte Anweisung erneut durchgeführt.

Code

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

Man beachte das Semikolon am Ende der Konstruktion.

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.

`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 und nicht eine neue Definition dieser Variablen.

Output

Datei1.cpp:

Code

```
int x = 1;
```

Output

Datei2.cpp:

Code

```
extern int x;
int y = ++x;
// x == 2
// y == 2
```

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.

Code

```
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 Referenz deklariert werden. Wenn ein Pointer übergeben wird, wird zwar der Pointer kopiert, aber das Objekt nicht, sodass sich auch hier *call by reference* ergibt.

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 verloren gehen.

Code
```
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:

Code
```
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:

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

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

Für Funktionsparameter sollten die folgenden Konventionen verwendet werden:

Ausgehende Parameter: Rückgabewerte sollten immer Werte oder Referenzen sein: `X f()` oder `X& f()`. Bei Referenzen ist darauf zu achten, dass dann die Referenz auch außerhalb der Funktion gültig sein muss.

Ein- und Ausgehende Parameter: Parameter, die geändert werden, sollten Referenzen sein: `f(X&)`

Eingehende Parameter: Parameter elementarer Datentypen (z.B. `int`): `f(X)` Parameter aus zusammengesetzten Datentypen oder Klassen (z.B. `vector`): `f(const X&)`

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.

Code
```
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 und mit Defaultparametern:

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

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

double pos(double y, int x);  // gleiche Funktion (Name der Argumente spielt keine Rolle) !!!
int pos(double y, int x);     // falsche Überladung (Typ des Rückgabeparameters spielt keine Rolle) !!!
double pos(double x, int y=1);  // gleiche Funktion (auch zu double pos(double x));
```

Präprozessor

Bei C++ gibt es einen Präprozessor, der vor der Compilierung Ersetzungen durchführt. Zeilen, die im Sourcecode mit `#` eingeleitet werden, heißen Steuerzeilen.

`include`

Der Präprozessor wird hauptsächlich dazu verwendet, Deklarationen aus anderen Headerdateien verfügbar zu machen. Eine Steuerzeile der Form `#include <Dateiname>` oder `#include "Dateiname"` wird vom Präprozessor durch die spezifizierte Datei ersetzt.

Dabei ist die Form mit `""` relativ zur aktuellen Datei und sollte für eigenen Quellcode benutzt werden. Ordner werden mit einem Schrägstrich (`/`) abgetrennt, nicht wie bei Windows mit einem umgekehrten Schrägstrich (\). Das übergeordnete Verzeichnis kann mit `..` ausgewählt werden. Dateien, die in der anderen Form (`<>`) angegeben sind, werden in den voreingestellten Include-Verzeichnissen gesucht und dienen zum Einbinden von Systembibliotheken.

`define`

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

```
#define KONSTANTE WERT
```

Der Präprozessor ersetzt vor der Compilierung jedes Auftreten von `KONSTANTE` im Sourcecode durch den `WERT`. Dadurch können diese Konstanten auch da benutzt werden, wo die Werte zur Compilierzeit feststehen müssen. Beispiel:

```
#define ARRAY_SIZE 100
int array[ARRAY_SIZE];
```

In C++ ist es allerdings besser, das `const`-Schlüsselwort zu verwenden, bei dem die weitere Informationen z.B. über den Datentyp nicht verloren gehen:

```
const int array_size = 100;
int array[array_size];
```

Zur besseren Unterscheidung von Variablen und Funktionen sollten für Präprozessorkonstanten nur Großbuchstaben und Unterstriche verwendet werden.

Include guards

Mit den bedingten Direktiven `#ifdef` und `#ifndef` lässt sich prüfen, ob ein Makro gegenwärtig definiert ist oder nicht. Das ist hilfreich, um Mehrfacheinbindungen von Headerdateien zu verhindern:

Code

```
#ifndef HEADER_H
#define HEADER_H

class Klasse
{
};

#endif
```

Auch wenn der Header mehrmals eingebunden wird, wird nur eine Kopie des Inhalts eingefügt:

Code

```
#include "header.h"
#include "header.h"
```

Die Definition der Konstanten verhindert, dass die Headerdatei mehrmals eingefügt wird und dann mehrere Definitionen der Klasse sichtbar sind, was verboten ist.

Bei vielen Compilern kann dies einfacher durch die Direktive `#pragma once` erreicht werden:

Code

```
#pragma once

class Klasse
{
};
```

Eine weitere Anwendung des Präprozessor sind funktionsartige Makros. Weil dabei keine Typ-Informationen vorhanden sind, sollten sie vermieden werden und werden hier daher nicht weiter betrachtet.

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.

Ein Namensbereich ist ein Gültigkeitsbereich für Namen. In verschiedenen Namensbereichen können gleiche Namen verwendet werden, ohne dass es zu Verwechslungen kommt. Die üblichen Regeln für Gültigkeitsbereiche treffen auch für Namensbereiche zu.

Beispiel:

Code

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

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

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

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

void name2::h(std::string s)
{
    std::cout << s << name1::f() << " "
          << name1::g(number) << 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 `number` 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` und `cout` gehören zur 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` folgendes Beispiel:

Code

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

using namespace std;  // using Direktive

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

namespace name2
{
    void h(string s);
    int number;
    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(number) << 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
  `cpp using namespace std;` in allen .cpp- und .h-Dateien nach den Includes der Standardbibliothek eingesetzt werden.

Klassen

- Konstruktor und Destruktor
- Konstante Instanzmethoden
- Statische Klassenelemente
- Der this-Zeiger

Ein Problem bei der Entwicklung großer Programme besteht darin, dass bei vielen Variablen die Abhängigkeiten nicht mehr überschaubar sind. In `C++` fasst man daher Daten zu Objekten zusammen, die miteinander interagieren. Zusätzlich werden Methoden auf Objekten definiert. Diesen Ansatz nennt man Objektorientierte Programmierung (OOP).

Idealerweise erfüllt eine Klasse ein definiertes Interface. Die Details der Implementierung werden dann vor dem Anwender der Klasse versteckt. Bei der OOP können Funktionalitäten einer Klasse durch Vererbung in andere Klassen übernommen werden.

Der Datentyp eines *Objektes* ist eine *Klasse*. In einer Klasse werden die *Datenelemente* (Variablen) definiert, die ein Objekt hat. Zusätzlich können der Datenstruktur *Methoden* (Funktionen) zugeordnet werden. Diese dienen der Manipulation und Abfrage der in der Datenstruktur enthaltenen Datenelemente.

Auch wenn einfache Datenstrukturen (`struct`) in die gleichen Funktionalitäten haben wie Klassen, gibt es Unterschiede darin, wie sie benutzt werden: Datenstrukturen mit `struct` sind Ansammlungen von Daten, bei denen alle Zustände valide sind und es gibt keine dieser Datenstruktur zugeordneten Funktionen zum Zugriff und zur Veränderung der Daten.

Klassen werden wie die aus C bekannten Datenstrukturen definiert, aber mit dem Schlüsselwort `class` statt `struct` eingeleitet. Für die Zugriffskontrolle gibt es die Zugriffsspezifizierer `public` und `private`.

Code

```
class Point
{
    public:
        double x, y;
};

class Date
{
    private:
        int year, month, day;
};
```

Während auf öffentliche (`public`) Datenelemente von jeder Stelle des Programms aus zugegriffen werden darf, dürfen auf die privaten Datenelemente nur die Methoden der Klasse zugreifen. Zugriffsspezifizierer können in beliebiger Reihenfolge und auch mehrfach innerhalb der Klasse verwendet werden und sind bis zur Angabe eines neuen Zugriffsspezifizierers gültig. Die Datenelemente einer Klasse heißen auch Instanzvariablen, die Methoden Instanzmethoden. Zugriff auf die Mitglieder (Datenelemente und Methoden) eines Objektes erhält man über den Punkt-Operator (`.`) bei Referenzen bzw. den Pfeil-Operator (`->`) bei Pointern.

Beispiel:

Code

```
class Example
{
    public:
        int i;
};

int main()
{
    Example obj;
    obj.i = 1;  // Zugriff mit operator .
    Example* p = &obj;
    int x = p->i;  // Zugriff bei Pointer mit operator ->
}
```

Die Instanzvariablen sollten normalerweise `private` sein. Falls sie von außen geändert werden, dann sollten sogenannte Setter programmiert werden, die die neuen Werte prüfen. Falls lesender Zugriff auf private Datenelemente auch von außerhalb der Klasse möglich sein soll, sollte eine Hilfsfunktion (auch Getter genannt) definiert werden. Die Instanzmethoden sollten normalerweise `public` sein. Der Name der Getter sollte mit `get` beginnen und dann eine Beschreibung des Wertes enthalten, analog sollten die Setter mit `set` beginnen.

Code

```
class Date
{
    private:
        int p_year, p_month, p_day;
    public:
        void setMonth(int m)
        {
            if (1 <= m && m <= 12)
                p_month = m;
            else
                ;  // hier z. B. Exception werfen
        }
        int getYear()
        {
            return p_year;
        }
};
```

Klassennamen sollten – wie alle selbst definierten Datentypen – zur Unterscheidung von Variablen- und Funktionsnamen mit einem Großbuchstaben beginnen. Instanzvariablen sollten zur Unterscheidung von lokalen Variablen mit einem Präfix gekennzeichnet sein, z.B. `p_`.

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 mit fundamentalen Datentypen wie `int` oder `float` durch Angabe des Klassennamens gefolgt vom Variablennamen.

Beispiel:

Code

```
class Example
{
    private:
        int p_i;  // p_i ist komplett privat
        int p_j;  // p_j ist über den Getter und Setter lesbar und schreibbar
    public:
        int k;  // k ist öffentlich les- und schreibbar
        int getJ() const { return p_j; }    // Getter für j
        void setJ(int j) { p_j = j>0?j:0; } // Setter, nur positive Werte für j zulassen
};

int main()
{
    Example b;  // Instanziierung eines Objektes der Klasse Beispiel
    b.k = 3;     // erlaubt, da hier public

    int x = b.getJ();  // Zugriff nur über öffentlichen getter
    // b.p_j = 4;      // Fehler: j nicht öffentlich schreibbar
    b.setJ(4);         // zulässig über Setter

    // int y = b.p_i;  // Fehler: i nicht öffentlich lesbar
    // b.p_i = 4;      // Fehler: i nicht öffentlich schreibbar
}
```

Konstruktor und Destruktor

Konstruktoren sind besondere Funktionen, die ein Objekt einer Klasse erstellen. Bei allen Instanziierungen wird zwingend ein Konstruktor aufgerufen. Im Sourcecode erkennt man die Konstruktoren daran, dass sie denselben Namen wie die Klasse tragen. Bei Klassen sollen Konstruktoren dafür sorgen, dass das Objekt einen definierten Zustand erhält, auf dem alle Funktionen aufgerufen werden können. Dazu sollten in den Konstruktoren alle Datenelemente initialisiert werden. Bei mehreren Konstruktoren wird anhand von Anzahl und Typ der Parameter unterschieden, welcher Konstruktor aufgerufen wird. Der Konstruktor ohne Parameter heißt Default- oder Standardkonstruktor. Objekte werden durch den Destruktor zerstört. Er trägt ebenfalls den gleichen Namen wie die Klasse, hat aber zusätzlich eine Tilde "~" als Präfix. Der Destruktor hat keine Parameter. Er sollte immer `virtual` definiert werden, um ein korrektes Löschen bei Unterklassen sicherzustellen. Konstruktoren und der Destruktor haben keine Rückgabeparameter. Den Variablen kann bei der Definition direkt ein Wert zugewiesen werden. Dies sollte genutzt werden, um sicherzustellen, dass alle Variablen immer einen definierten Wert haben.

Beispiel:

Code

```
// 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 getSum(){return p_i+p_j;};
private:
   int p_j = 0;  // Initialisierung mit 0
   int p_i = 0;
};
```

Code

```
Sum::Sum()
{
    std::cout << "Standardkonstruktor" << std::endl;
}
```

Code

```
Sum::Sum(int i, int j)
{
    if (i > 0)   // Abfrage zu Wertebereich möglich
    {
        p_i = i; // Instanzvariable setzen
    }
    if (j > 0)   // Abfrage zu Wertebereich möglich
    {
        p_j = j; // Instanzvariable setzen
    }
    std::cout << "Nicht-Standardkonstruktor" << std::endl;
}
```

Code

```
Sum::~Sum()
{
    std::cout << "Aufruf des Destruktors" << std::endl;
}
```

Code

```
void Sum::setI(int i)
{
    if (i > 0) // Abfrage zu Wertebereich möglich
    {
        p_i = i; // Instanzvariable setzen
    }
}
```

Code

```
int main()
{
    std::cout << "Anfang" << std::endl;
    Sum tS1;         // Statische Erzeugung mit Standardkonstruktor
    tS1.setI(6);     // Zugriff auf p_i nur über Funktion möglich.
    Sum tS2(1,3);    // Erzeugung mit Nicht-Standardkonstruktor
    std::cout << tS1.getSum() << std::endl;
    std::cout << tS2.getSum() << std::endl;
    std::cout << "Ende" << std::endl;
}
```

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

Code

```
main();
```

Der Default-Konstruktor sollte immer definiert werden. Er wird automatisch generiert, wenn keine anderen Konstruktoren definiert sind, ansonsten nicht. Automatisch generierte Konstruktoren und Destruktoren kan man aktivieren (`=default`) und deaktivieren (`=delete`). Als `default` definierte Funktionen brauchen dann nicht implementiert werden, es entspricht der leeren Implementierung durch {} .

Code

```
class Date
{
    // ...
    public:
        Date(int year, int month, int day);
        Date() = default;
        virtual ~Date() = default;
};
```

Es gibt mehrere Möglichkeiten, die Datenelemente eines Objektes bei der Konstruktion zu initialisieren. Die einfachste ist es, wie oben in der Klassendefinition Default-Werte anzugeben. Bei der Konstruktion werden

dann in der Definitionsreihenfolge die Werte konstruiert. Dabei ist es möglich, sich auf vorhergehende Elemente zu beziehen. Der Vorteil dieser Methode ist, dass die Vorbesetzungen in allen Konstruktoren gleich sind.

Code

```
class Date
{
    // valides Datum
    private:
        int p_year = 1970;
        int p_month = 1;
        int p_day =  1;

        int p_day_after = p_day + 1;
};
```

Für die nächste Variante muss ein Konstruktor definiert werden. In der sogenannten Initialisierungsliste werden Elemente mit Werten konstruiert. Falls ein Element nicht erwähnt wird, wird die Konstruktion aus der Klassendefinition verwendet. Damit können selektiv einzelne Elemente angegeben werden. Die anderen werden dann wie in der Klassendefinition initialisiert. Es wird jeweils zu dem Namen einer Instanzvariablen in Klammern die entsprechende Initialisierung gesetzt.

Code

```
Date::Date(int year, int month, int day)
    // Initialisierungsliste
    : p_year(year),
      p_month(month),
      p_day(day)
    // implizit: p_day_after = p_day + 1;
{}
```

Datenelemente, die nur einmal bei der Initialisierung einen Wert erhalten, können als `const` definiert werden. Sie können dann nach der Initialisierung nicht mehr verändert werden. Dabei können auch Operatoren eingesetzt und auf bereits initialisierte Variablen zugegriffen werden.

Code

```
class Date
{
    // valides Datum
    private:
        const int p_year = 1970;
        const int p_month = 1;
        const int p_day =  1;

        const int p_day_after = p_day + 1;
};
Date::Date(int year, int month, int day)
    // Initialisierungsliste
    : p_year((year>1970)?year:0), // Jahr frühestens 1970
      p_month((month>0 && month<13)?month:0),
      p_day((day>0 && day<32)?day:0),
      p_day_after(p_day + 1)
{
}
```

Die dritte Variante, Datenelemente zu initialisieren ist, ihnen im Konstruktorrumpf Werte zuzuweisen. Das ist notwendig, wenn man etwa aufwändigere Tests bzgl. der erlaubten Tage abhängig Monat oder vom Jahr (Schaltjahr) durchführen möchte. Da dabei aber erst die Elemente wie in der ersten Methode initialisiert werden, sind die Werte schon vor der Zuweisung konstruiert. Daher ist die Zuweisung eine doppelte Operation und sollte, wenn möglich, vermieden werden. Bei Datenelementen, die `const` sind, ist eine Zuweisung im Konstruktorrumpf nicht möglich, sondern nur die Konstruktion mit einer der ersten beiden Methoden.

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:

Code

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

Der Copykonstruktor existiert immer automatisch, 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 oder bestimmte eindeutige Kennzeichnungen 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 mit `=delete` deaktivieren:

Code
```
class Class
{
private:
    Class(const Class&) = delete;
};
```

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:

Code
```
class HString
{
private:
    char* p_pString = nullptr;
    int p_iSize = 0;
public:
    HString() = default;
    HString(const HString&);
};

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

Konstante Instanzmethoden

Methoden, die die Instanzen ihrer Klasse nicht verändern, sollten zur besseren Wartbarkeit als konstant markiert werden. Dazu wird hinter der Funktionsdefinition das Schlüsselwort `const` hinzugefügt.

Beispiel:

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

int Class::getElement() const
{
    // iElement = 3;  // Fehler, da Element verändert wird
    return iElement;
}
```

Konstante Methoden dürfen den Zustand des Objektes nicht verändern, weswegen es im Beispiel an der auskommentierten Stelle zu einem Fehler beim Compilieren kommen würde. Konstante Methoden dürfen nur ebensolche Methoden aufrufen. Eine Markierung mit `const` ändert die Signatur der Funktion, d.h. eine Definition von `int Klasse::getElement() const` entspricht nicht einer Deklaration `int getElement()` in der Klasse.

Statische Klassenelemente

Datenelemente einer Klasse können mit Hilfe des Schlüsselworts `static` als Klassenvariable deklariert werden. Von Klassenvariablen 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.

Funktionen können mit `static` als Klassenfunktion 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.

Die Definition und Initialisierung statischer Datenelemente erfolgt entweder in der Klasse mit dem zusätzlichen Schlüsselwort `inline`

Code
```
class S {
    private:
        static inline int p_max = 0;
};
```

oder auf Datei-Ebene mit der Deklaration in der Header-Datei:

Code
```
class S {
    private:
        static int p_max;
};
```

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

Code
```
class Example
{
    private:
        static inline int p_iCounter = 0;  // statische Variable
        const int p_iNumber = p_iCounter++;
    public:
        static void printCounter()  // statische Methode
        {
            std::cout << "# erzeugte Objekte: " << p_iCounter << std::endl;
        }
        void printNumber() const
        {
            std::cout << "Objekt#: " << p_iNumber << std::endl;
        }
};
```

Code
```
int main()
{
    Example::printCounter();

    Example a;
    Example::printCounter();
    a.printNumber();

    Example b;
    Example::printCounter();
    b.printNumber();
}
```

Die Ausgabe des obigen Beispiels sieht wie folgt aus:

Code
```
main();
```

Versuchen Sie an dieser Stelle, den Code-Block, der die `main()`-Funktion aufruft, mehrmals hintereinander auszuführen. Überlegen Sie sich vorher, welche Ausgabe Sie erwarten

Der `this`-Zeiger

Werden mehrere Objekte einer Klasse erzeugt, so sind nur die Datenelemente in diesem Objekt vorhanden. Die Methoden hingegen werden für jede Klasse nur einmal erzeugt, da ihr Code für jedes Objekt gleich sind. Bei Aufruf einer Methode wird deswegen das konkrete Objekt übergeben. Der Compiler übergibt bei jedem Methodenaufruf das jeweilige Objekt als Pointer. Innerhalb der Funktion kann man durch `this` auf diesen Pointer zugreifen. Diese Adresse wird auch als `this`-Zeiger bezeichnet.

Übungsaufgaben zu diesem Kapitel finden Sie **hier**.

Übungsaufgaben zu Klassen

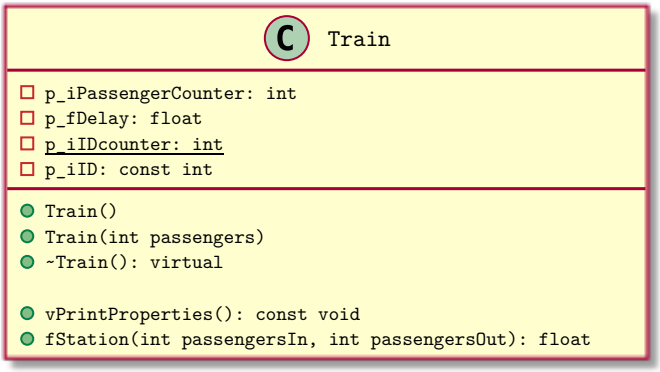
Hier finden Sie eine Übungsaufgabe, um die gerade gelernten Konzepte einmal selbst zu verwenden, sowie einige Verständnisfragen.

- Praktische Aufgabe
- Verständnisfragen

Praktische Aufgabe

Im Folgenden soll das gerade gelernte angewendet werden. Zu jedem Teil der Übungsaufgabe gibt es eine eingeklappte Lösung. Bevor Sie sich diese anschauen, sollten Sie versuchen, die gewünschte Funktionalität selbst zu implementieren. Die Lösungsvorschläge stellen eine mögliche Lösung für die Aufgaben dar, aber auch andere Ansätze können die Aufgaben lösen.

Die Teile der Aufgabe bauen aufeinander auf. Es kann notwendig sein, dass sie zwischen den Aufgabenteilen den Kernel neu starten (). Das gleiche gilt für die Lösungsvorschläge.



Teil 1

Erstellen Sie eine Klasse mit dem Name `Train`. Die Klasse soll die Instanzvariablen `p_iPassengerCounter` (`int`) und `p_fDelay` (`float`) beinhalten.

Objekte dieser Klasse sollen sowohl über den Standardkonstruktor, als auch über einen Nicht-Standardkonstriktor instanziiert werden können. Bei Verwendung des Standard-Konstruktors sollen beide Werte `0` sein. Der Nicht-Standardkonstruktor soll die Möglichkeit bieten, die Variable `p_iPassengerCounter` zu setzen.

Erstellen Sie anschließend mit beiden Konstruktoren je eine Instanz der Klasse.

zum Lösungsvorschlag

Teil 2

Erweitern Sie Ihre Klasse um eine Methode, die die Instanzvariablen ausgibt (`void printPropierties()`). Rufen Sie die Methode für beide Instanzen auf.

zum Lösungsvorschlag

Teil 3

Erstellen Sie eine Methode `float station(int passengersIn, int passengersOut)`. Die Methode soll die Variable `p_iPassengerCounter` aktualisieren. Der Rückgabewert der Methode soll die aufgebaute (positive Zahl) oder abgebaute (negative Zahl) Verspätung in Minuten sein. Dabei wird davon ausgegangen, dass ein aussteigender Passagier 5 Sekunden und ein einsteigender Passagier 10 Sekunden benötigt. Um keine Verspätung aufzubauen, darf der Halt 2 Minuten dauern.

Außerdem soll die Verspätung in der Variable `p_fDelay` aktualisiert werden. Die Verspätung soll aber nicht negativ sein können.

Für den Fall, dass mehr Personen aussteigen sollen als es Passagiere im Zug gibt, soll die Methode `0` returnen und die Anzahl der Passagiere nicht verändern.

zum Lösungsvorschlag

Teil 4

Zuletzt soll die Klasse ein `static`-Member erhalten, um jedem erzeugten Zug eine eindeutige ID zuweisen zu können. Erweitern Sie Ihrer Klasse dazu durch die Membervariablen `static p_idCounter` und `p_iID`. Auch `printProperties()` soll entsprechend angepasst werden.

Erstellen Sie zum Testen der Funktionalität einige Züge und lassen Sie sich die Eigenschaften ausgeben.

zum Lösungsvorschlag

Code
```
#include <iostream>
```

Code
```
// Train.h
// hier: Klasse Train implementieren
```

Code
```
// Train.cpp
// hier: Konstruktor implementieren
```

Code
```
// Train.cpp
// hier: printProperties() implementieren
```

Code
```
// Train.cpp
// hier: station(...) implementieren
```

Code
```
// main.cpp
// hier: Instanzen erzeugen, Ausgabe
```

Lösungen zum Programmier-Beispiel

Lösung zu Teil 1

zur Aufgabenstellung

Code
```
#include <iostream>
```

Code
```
// Train.h
class Train
{
    private:

        int p_iPassengerCounter = 0;
        float p_fDelay = 0;

    public:
        Train() = default;
        Train(int passengers);
        virtual ~Train() = default;
};
```

Code
```
// Train.cpp
Train::Train(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
// main.cpp
Train aTrain = Train();
Train bTrain = Train(20);
```

Output

```
Aufruf des Nicht-Standardkonstruktors
```

Lösung zu Teil 2

zur Aufgabenstellung

Code

```
#include <iostream>
```

Code

```
// Train.h
class Train
{
    private:

        int p_iPassengerCounter = 0;
        float p_fDelay = 0;
    public:
        Train() = default;
        Train(int passengers);
        virtual ~Train() = default;

        void vPrintProperties() const;
}
```

Code

```
// Train.cpp
Train::Train(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
// Train.cpp
void Train::vPrintProperties() const
{
    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
    std::cout << "momentane Verspaetung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
// main.cpp
Train aTrain = Train();
Train bTrain = Train(20);

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
aTrain.vPrintProperties();

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
bTrain.vPrintProperties();
```

Output

```
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
Anzahl Passagiere: 0
momentane Verspaetung: 0 Minuten

Eigenschaften 'bTrain':
Anzahl Passagiere: 20
momentane Verspaetung: 0 Minuten
```

Lösung zu Teil 3

zur Aufgabenstellung

Code

```
#include <iostream>
```

Code

```
// Train.h
class Train
{
    private:

        int p_iPassengerCounter = 0;
        float p_fDelay = 0;

        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;
    public:
        Train() = default;
        Train(int passengers);
        virtual ~Train() = default;

        void vPrintProperties() const;
        float fStation(int passengersIn, int passengersOut);
}
```

Code

```
// Train.cpp
Train::Train(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
// Train.cpp
void Train::vPrintProperties() const
{
    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
    std::cout << "momentane Verspaetung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
// Train.cpp

// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/


float Train::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Minuten
    float minutes_change = (120 - passengersIn * 10 - passengersOut * 5)/60.0;
    p_fDelay = (p_fDelay - minutes_change < 0) ? 0 : (p_fDelay - minutes_change);

    return minutes_change;
}
```

Code

```
// main.cpp
Train aTrain = Train();
Train bTrain = Train(20);

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
aTrain.vPrintProperties();

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
bTrain.vPrintProperties();

// ---


bTrain.fStation(15, 10);
std::cout << std::endl << "\n\n\nEigenschaften 'bTrain' nach 1. Halt:" << std::endl;
bTrain.vPrintProperties();

bTrain.fStation(3, 12);
std::cout << std::endl << "Eigenschaften 'bTrain' nach 2. Halt:" << std::endl;
bTrain.vPrintProperties();
```

Output

```
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
Anzahl Passagiere: 0
momentane Verspaetung: 0 Minuten

Eigenschaften 'bTrain':
Anzahl Passagiere: 20
momentane Verspaetung: 0 Minuten




Eigenschaften 'bTrain' nach 1. Halt:
Anzahl Passagiere: 25
momentane Verspaetung: 1.33333 Minuten

Eigenschaften 'bTrain' nach 2. Halt:
Anzahl Passagiere: 16
momentane Verspaetung: 0 Minuten
```

Lösung zu Teil 4

zur Aufgabenstellung

Code

```
#include <iostream>
```

Code

```
// Train.h
class Train
{
    private:

        int p_iPassengerCounter = 0;
        float p_fDelay = 0;

        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;
    public:
        Train() = default;
        Train(int passengers);
        virtual ~Train() = default;

        void vPrintProperties() const;
        float fStation(int passengersIn, int passengersOut);
}
```

Code

```
// Train.cpp
Train::Train(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
// Train.cpp
void Train::vPrintProperties() const
{
    std::cout << "ID: " << p_iID << std::endl;

    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
    std::cout << "momentane Verspaetung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
// Train.cpp

// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/


float Train::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Minuten
    float minutes_change = (120 - passengersIn * 10 - passengersOut * 5)/60.0;
    p_fDelay = (p_fDelay - minutes_change < 0) ? 0 : (p_fDelay - minutes_change);

    return minutes_change;
}
```

Code

```
// main.cpp

Train aTrain = Train();
Train bTrain = Train(20);
Train cTrain = Train(52);
Train dTrain = Train(5);

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
aTrain.vPrintProperties();

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
bTrain.vPrintProperties();

std::cout << std::endl << "Eigenschaften 'cTrain':" << std::endl;
cTrain.vPrintProperties();

std::cout << std::endl << "Eigenschaften 'dTrain':" << std::endl;
dTrain.vPrintProperties();
```

Output

```
Aufruf des Nicht-Standardkonstruktors
Aufruf des Nicht-Standardkonstruktors
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
ID: 0
Anzahl Passagiere: 0
momentane Verspaetung: 0 Minuten

Eigenschaften 'bTrain':
ID: 1
Anzahl Passagiere: 20
momentane Verspaetung: 0 Minuten

Eigenschaften 'cTrain':
ID: 2
Anzahl Passagiere: 52
momentane Verspaetung: 0 Minuten

Eigenschaften 'dTrain':
ID: 3
Anzahl Passagiere: 5
momentane Verspaetung: 0 Minuten
```

Verständnisfragen

1. Die folgende Zeile produziert eine Fehlermeldung:

Code

```
Train thisIsATrain = Train();

std::cout << thisIsATrain.p_iPassengerCounter << std::endl;
```

Output

```
input_line_34:3:27: error: 'p_iPassengerCounter' is a private member of '__cling_N514::Train'
std::cout << thisIsATrain.p_iPassengerCounter << std::endl;
                          ^

input_line_29:6:13: note: declared private here
        int p_iPassengerCounter = 0;
            ^
```

Output

```
Interpreter Error:
```

- Was müsste im Code der Klasse geändert werden, um diese Fehlermeldung zu verhindern? Verändern Sie die entsprechende Stelle und beobachten Sie was mit der Fehlermeldung passiert.
- Grundsätzlich vermeidet man es, Instanzvariablen zu lesen oder zu bearbeiten, wie es in diesem Beispiel versucht wird. Falls eine Membervariable von außen gelesen/bearbeitet werden soll, wie soll dies geschehen?

2. Führen Sie den letzen Block aus der Musterlösung mehrmals hintereinander aus, ohne den Kernel neu zu starten. Was können Sie beoachten? Wieso?

Lösungen zu den Verständnisfragen

1.

- In `Train` müsste `p_iPassengerCounter` unter `public` stehen. Da man versucht, dem Anwender einer Klasse die Details der Implementierung zu verbergen, ist es nicht hilfreich, alle Member `public`, also von außen zugänglich zu machen.
- Es sollte für jede von außen zu lesende/bearbeitende Variable ein Getter/Setter programmiert werden.

2. Lokale Variablen (*hier: aTrain, bTrain, ...*) werden bei jeder Ausführung überschrieben und neu erzeugt, 'p_idCounter' wird nicht zurückgesetzt -> 'p_ID' steigt

Vererbung

- Zugriffsspezifizierer
- Polymorphie
- Abstrakte Klassen
- Freundschaften
- Überladen
  - Methoden überladen
  - Operatoren überladen

Die Vererbung (engl.: *Inheritance*) ist ein Mechanismus, der es erlaubt, Datenstrukturen und Methoden einer Klasse (Basisklasse) bei der Bildung einer neuen Klasse (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.

Als Beispiel sei hier die Klasse **Train** genannt, die als Basisklasse für die Unterklasse **PassengerTrain** dient:

Code

```
// Beispiel
class PassengerTrain: public Train;
```

Zugriffsspezifizierer

Mit der obigen Syntax wird eine Klasse deklariert, 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. Ein öffentliches Datenelement der Basisklasse ist bei `public`-Vererbung auch in der abgeleitete Klasse öffentlich, `private` Elemente bleiben `private`. Mit `private`-Vererbung kann der Zugriff auch für Datenelemente eingeschränkt werden, die in der Basis öffentlich sind. Dadurch ist es möglich, eine Vererbung komplett vom Anwender der Klasse zu verstecken.

In vielen Fällen will man auf Datenelemente von Basisklassen auch in abgeleiteten Klassen zugreifen. Dafür gibt es einen dritten Zugriffsspezifizierer: `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.

Element in Basis	Vererbung	Element in abgeleiteter Klasse
`private`	`public`	Zugriff nicht möglich
`protected`	`public`	`protected`
`public`	`public`	`public`
-	-	-
`private`	`private`	Zugriff nicht möglich
`protected`	`private`	`private`
`public`	`private`	`private`

Tabelle 3.1: Zugriffsmöglichkeiten auf Basisklassenelemente

Tabelle 3.1 gibt einen Überblick über die Zugriffsmöglichkeiten auf Elemente von Basisklassen. In der Regel verwendet man für die Vererbung `public`, da dann der Zugriff auf Mitglieder der abgeleiteten Klasse in gleicher Weise möglich ist wie der Zugriff auf Mitglieder der Basisklasse. Falls der Zugriffsspezifizierer weggelassen wird, ist die Vererbung `private`.

Bei der Konstruktion von Objekten wird zunächst der Konstruktor der Basisklasse aufgerufen, danach der Konstruktor der abgeleiteten Klasse. Um zu steuern, welcher Konstruktor der Basisklasse benutzt werden soll, kann als erstes in einer Initialisierungsliste der gewünschte Konstruktor der Basisklasse angegeben werden. Wird dort kein Konstruktor angegeben, wird der Standardkonstruktor der Basisklasse aufgerufen. Das folgende Beispiel zeigt die Verwendung einer Klassenhierarchie anhand von einfachen grafischen Elementen:

Code

```
#include <iostream>
```

Code

```
class Shape
{
    protected:
        double p_x = 0.0;
        double p_y = 0.0;
    public:
        Shape () = default;
        Shape(double x, double y) : p_x(x), p_y(y) {}
        double area () const {
            return 0.0;
        }
};

class Box : public Shape // Box abgeleitet von Shape
{
    double p_w = 0.0;
    double p_h = 0.0;

    public:
    Box() = default;
    Box(double x, double y, double w, double h)
        : Shape(x, y), // Aufruf des Konstruktors der Basisklasse
            p_w(w),
            p_h(h)
        {}
    double area () const
    {
        return p_w * p_h;
    }
};

class Line : public Shape {
    double p_end_x = 0.0;
    double p_end_y = 0.0;

    public:
    Line() = default;
    Line(double x, double y, double end_x, double end_y)
        : Shape(x, y), // Aufruf des Konstruktors der Basisklasse
            p_end_x(end_x),
            p_end_y(end_y)
        {}
};
// double area() const geerbt von Shape
```

Code

```
int main () {
    Box box(10.0, 10.0, 6.0, 7.0);
    Line line(10.0, 10.0, 30.0, 30.0);
    // Fläche berechnen
    std::cout << "Box area:\t" << box.area() << std::endl;
    std::cout << "Line area:\t" << line.area() << std::endl;         // 0.0, geerbt
};
```

Code

```
main();
```

Polymorphie

Viele Klassen lassen sich auf die gleiche oder ähnliche Weise benutzen. Im oben gezeigten Beispiel der Shape-Klasse würde die Methode area von mehreren Klassen geerbt werden, wobei die Implementierung für jede Klasse unterschiedlich, die Schnittstelle aber gleich ist. Damit alle Objekte, die die Schnittstelle implementieren, über eine Referenz als Shape oder einen Pointer auf Shape korrekt angesprochen werden, muss die Funktion in der Basisklasse als virtuell (Schlüsselwort `virtual`) deklariert werden. Wenn area() über Referenz oder Pointer der Basisklasse aufgerufen wird, wird bei einer virtuellen Funktion zunächst in der Unterklasse überprüft, ob eine Implementierung für diese Funktion vorliegt. Dann wird die richtige Implementierung für die area-Methode ausgewählt. Man spricht dann von Polymorphie (Vielgestaltigkeit). Wird die Funktion in der Basisklasse nicht virtuell definiert, wird sonst bei einem Zeiger oder einer Referenz der Basisklasse direkt die Funktion der Basisklasse aufgerufen. Man spricht hier auch von dynamischer bzw. später Bindung (engl.: late binding).

Beispiel: (identisch bis auf die Definition von Shape::area() als **virtual**)

Code

```
namespace x {
class Shape
{
    protected:
        double p_x = 0.0;
        double p_y = 0.0;
    public:
        Shape () = default;
        Shape(double x, double y) : p_x(x), p_y(y) {}
        virtual double area () const {    // <-------Deklaration als virtuelle Methode
            return 0.0;
        }
};
class Box : public Shape // Box abgeleitet von Shape
{
    double p_w = 0.0;
    double p_h = 0.0;

    public:
    Box() = default;
    Box(double x, double y, double w, double h)
        : Shape(x, y), // Aufruf des Konstruktors der Basisklasse
            p_w(w),
            p_h(h)
        {}
    double area () const
    {
        return p_w * p_h;
    }
};

class Line : public Shape {
    double p_end_x = 0.0;
    double p_end_y = 0.0;

    public:
    Line() = default;
    Line(double x, double y, double end_x, double end_y)
        : Shape(x, y), // Aufruf des Konstruktors der Basisklasse
            p_end_x(end_x),
            p_end_y(end_y)
        {}
};}
```

Wenn dann Objekte als Referenz oder Pointer von `Shape` verwendet werden, werden die Methoden der Unterklasse aufgerufen, sofern sie existieren. Wenn sie in der abgeleiteten Klasse nicht überschrieben wird, wird die Methode aus der Basisklasse aufgerufen.

Code

```
int main()
{
x::Line line (10.0 , 10.0 , 30.0 , 30.0);
x::Shape* shape1 = new x::Box (10.0 , 10.0 , 20.0 , 20.0);

std::cout << shape1 ->area () << std::endl; // ruft Box:: area auf

x::Shape& shape_ref1 = *shape1;
std::cout << shape_ref1.area() << std::endl; // ruft Box::area auf

x::Shape& shape_ref2 = line;
std::cout << shape_ref2.area() << std::endl; // ruft Shape::area auf , da Line::area nicht existiert
}
```

Code

```
main();
```

Würde hier das `virtual` in `Shape` weglassen werden, würde immer die Funktion aus der Basisklasse (`Shape`) aufgerufen. Sinnvoll ist das Speichern von Pointern der Basisklasse besonders bei gemeinsamen Feldern oder Containern. Die Überschreibung von Methoden geht nur bei gleichen Signaturen. Die Signatur einer Methode besteht aus dem Namen, Anzahl und Datentyp der Argumente sowie `const`. Häufig wird versucht irrtümlich, Methoden, die in der Basis `const` sind, mit nicht-`const`-Methoden zu überschreiben oder umgekehrt:

Code

```
class Base
{
    public:
        virtual void method() const;
};
class Derived : public Base {
    public:
        virtual void method(); // hier fehlt const
};
int main() {
    Derived d;
    Base& base = d;
    base.method(); // ruft Base::method auf, da die const Funktion nicht überschrieben wird
}
```

Da in Derived das `const` fehlt, hat `Derived` zwei Methoden, die `method` heißen: `void method()` und `void method() const` (von Base geerbt). Da das in den allermeisten Fällen nicht beabsichtigt ist, sollten alle überschreibenden Methoden mit dem Schlüsselwort `override` gekennzeichnet werden. Der Compiler gibt dann Fehler aus, wenn keine passende Methode in der Basisklasse vorhanden ist.

Code

```
void method() override; // Fehler: Markiert als override, aber überschreibt nicht
```

Wenn die oberste Methode `virtual` ist, sollten alle überschreibenden Methoden auch `virtual` sein. Angemerkt sei noch, dass Destruktoren immer `virtual` erklärt werden sollten. Ein virtueller Destruktor sorgt dafür, dass der Destruktor einer abgeleiteten Klassen aufgerufen wird, auch wenn ein Pointer verwendet wird, der vom Typ der Basisklasse ist. So wird sichergestellt, dass alle belegten Ressourcen freigegeben werden.

Konstruktoren dürfen nicht virtuell erklärt werden. Wenn ein Konstruktor virtuell wäre, würde die Auswahl der Implementierung vom Datentyp abhängen. Da zur Konstruktionszeit das Objekt noch nicht existiert, kann diese Auswahl nicht getroffen werden. Außerdem macht es keinen Sinn, ein Objekt mit unbekanntem Datentyp zu erstellen.

Da die späte Bindung zur Laufzeit erfolgt, ist sie etwas langsamer als statische Aufrufe. Daher ist es nicht sinnvoll, alle Methoden einer Klasse ohne Grund `virtual` zu definieren.

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 sollen. So könnte z.B. eine Basisklasse `Zug` die gemeinsamen Eigenschaften von `Personenzug` und `Güterzug` enthalten. Allerdings wäre es nicht sinnvoll, allgemeine Züge zu erstellen, da die Objekte in diesem Beispiel entweder der einen oder der anderen Kategorie angehören sollen. Analog sollten sich auch keine Objekte der Klasse Shape erstellen lassen. In C++ realisiert man eine abstrakte Klasse durch Deklaration mindestens einer **„rein-virtuellen“** Methode. Eine rein-virtuelle Methode wird bei der Deklaration mit $= 0$ markiert. Die Klasse heißt dann ***„abstrakt“***. Es können keine Objekte einer abstrakten Klasse erzeugt werden. Es ergibt sich folgende Syntax für eine solche Methode:

```
virtual return-type function-name (parameter-list ) = 0;
```

Zusätzlich können auch rein-virtuelle Methoden mit `const` markiert werden. Eine rein-virtuelle Methode braucht in der Basisklasse nicht implementiert werden. Eine zusätzliche Implementierung ist aber möglich. In allen abgeleiteten Klassen müssen rein-virtuelle überschrieben werden. Mit rein-virtuellen Funktionen gibt man so ein Implementierungsinterface für die Unterklassen vor. Wie bei allen Klassen mit virtuellen Methoden, sollte auch bei einer abstrakten Klasse ein virtueller Destruktor definiert werden. Wenn man keine eigene Funktion als rein virtuell definieren will, kann man daher den Destruktor benutzen, um eine Klasse als abstrakt zu kennzeichnen. Ansonsten kann man ihn mit `=default` definieren.

Freundschaften

Ein grundlegendes Konzept von C++ ist die Datenkapselung, d.h. nur die Methoden einer Klasse besitzen Zugriff auf die privaten Mitglieder. Hierdurch kann das Objekt selbst kontrollieren, welche Werte in seine Instanzvariablen geschrieben werden und Fehleingaben abfangen. In Ausnahmefällen kann es jedoch sinnvoll sein, dass auch Funktionen, die nicht Mitglied der Klasse sind, Zugriff auf 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 sparsam einsetzen und durch entsprechende Zugriffsfunktionen ersetzen. Das Beispiel zeigt die Verwendung von Freundschaften:

Code
```
class Person
{
    private:
        int p_iAge;
    public:
        Person() = default;
        Person(int age): p_iAge(age){};
        int getAge() const {return p_iAge;}

    //hier wird der Funktion "vBirthday" die Freundschaft angeboten
    friend void vBirthday(Person&);

};
```

Code
```
void vBirthday(Person &person) // globale Methode, nicht in Klasse
{
    // Zugriff auf privates Mitglied von p
    person.p_iAge++;
}
```

Code
```
int main ()
{
    Person hans = Person(42);
    std::cout << hans.getAge() << std::endl;
    vBirthday(hans);
    std::cout << hans.getAge() << std::endl;
}
```

Code
```
main();
```

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.

Überladen

In C++ können mehrere Funktionen mit gleichen Namen definiert werden. Diese Technik wird als Überladen (engl. *overload*) bezeichnet. Da Operatoren auch Funktionen sind, können sie auch überladen werden. Welche überladene Funktion aufgerufen wird, bestimmen Typ und Anzahl der Argumente. Das bedeutet, dass sich überladene Funktionen in der Anzahl 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.

Methoden überladen

Das folgende Beispiel zeigt eine überladene Funktion, die das Maximum zweier **int**-oder **char**-Werte liefert:

Code
```
int max(int a, int b) { return (a > b) ? a : b; }     //int comparison
```

Code
```
char max(char a, char b) { return (a > b) ? a : b; }  //char comparison
```

*Bemerkung: Der Methodenrumpf ist hier identisch, da für Variablen vom Datentyp **char**, die intern als ASCII-Werte gespeichert werden, der Vergleichsoperator $>$ überladen ist (dazu im nächsten Abschnitt mehr)*

Der Versuch, die Funktion ein weiteres Mal zu überladen, so dass der größere zweier **int**-Werte ausgegeben wird, führt allerdings zu einem Fehler. Ansonsten wäre nämlich beim Aufruf dieser mehrfach überladenen Methode nicht entscheidbar, welche der beiden Implementierungen mit identischer Signatur hier verwendet werden soll.

Code
```
void max(int a, int b)
{
if (a > b) std::cout << a << " is bigger than " << b << std:: endl;
else if (b > a) std::cout << b << " is bigger than " << a << std:: endl;
else std::cout << a << " equals " << b << std:: endl;
}
```

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

Konkret unterscheidet sich diese Funktion von `int max(int a, int b)` nur im Typ des Rückgabewerts (**void** statt **int**). Daher könnte 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: Default-Konstruktor, Copykonstruktor und einen parametrisierten Konstruktor.

Code

```
class Studi
{
    private:
        int p_ID;
        int p_Semester;
    public:
        Studi();                        // Standard-Konstruktor
        Studi(const Studi& aStud);    // Copykonstruktor
        Studi(int id, int semester); // param. Konstruktor
};
```

Code

```
int main (){
    Studi studi;                // Standard-Konstruktor
    Studi studi2 (studi);       // Copykonstruktor
    Studi arthur (123456, 42); // param. Konstruktor
}
```

Man beachte die Unterschiede zwischen den vorgestellten Möglichkeiten, Methoden mit gleichen Namen zu definieren:

Bezeichnung	Parameter	Ort
1. Vererbung	identisch	in Unterklasse, geerbt von Basisklasse
2. Polymorphie	identisch	in verschiedenen Klassen
3. Überladung	verschieden	in der gleichen Klasse

Operatoren überladen

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 in int a = **operator**+(4, 3); übersetzt.

Für die fundamentalen Datentypen wie z. B. `int` können keine Operatoren überladen werden. Daher muss bei der Operatorüberladung mindestens der 1. Operand ein Objekt sein. Operatoren auf komplexeren Datenstrukturen können individuell definiert werden. Der Plus-Operator zur Stringverkettung ist zum Beispiel eine Überladung. Bei den folgenden Operatoren kann eine Überladung sinnvoll sein:

- Vergleich (==, !=, <, <=, > und >=)
- Arithmetisch (+, -, *, / und %)
- Index-Zugriff ([])
- Zuweisung (=, +=)
- Ein-/Ausgabe («, »)

Es können auch noch weitere Operatoren überladen werden.

In C++ werden Klassenmethoden wie normale Funktionen behandelt, die implizit als ersten Parameter den `this`-Zeiger der zugehörigen Instanz bekommen. Das gilt auch für Operatoren. 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.

Manchmal ist eine Definition innerhalb der Klasse unmöglich, z.B. wenn der erste Parameter nicht ein Objekt der Klasse sein soll, da bei Methoden der `this`-Zeiger der erste Parameter ist. Beispiele sind die Ein- und Ausgabeoperatoren («, »), 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 Klasse X:

Code

```
class X
{
    public:
        // Inkrement:
        const X operator ++( int); // Postfix -Inkrement
        X& operator ++();          // Praefix -Inkrement

        // Zuweisungen:
        X& operator +=( const X&); // X+=X
        // analog für andere Zuweisungen

        // Andere:
        X& operator []( int); // Subskript
};
```

Code

```
// arithmetisch und logisch:
X& operator +( const X&, const X&)
// andere analog
```

Code

```
//Vergleich
bool operator ==( const X&, const X&);
// analog für !=,<,>,<=,>=
```

Code

```
// Ein - und Ausgabe
std::istream& operator >>( std::istream&, X&);
```

Code

```
std::ostream& operator <<( std::ostream&, const X&);
```

Anmerkungen:

- Wie bei den meisten Methoden sollten konstante Referenzen übergeben werden, damit Objekte nicht beim Funktionsaufruf kopiert werden. 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 ein Element der Klasse (normalerweise wird dies this sein) zurückgeben, da dadurch Zuweisungsketten wie a=(b=c); oder auch (a=b)=c; möglich sind. Beachten Sie den Unterschied zwischen =-Operator und Copykonstruktor: Beim Copykonstruktor wird ein neues Element erzeugt, beim =-Zuweisungsoperator sind beide Elemente bereits vorher erstellt worden. Beide Operatoren sind per default vorhanden und kopieren die Variablen byteweise.

Code

```
MyClass x;      // x wird per Standardkonstruktor erstellt
MyClass y(x);   // y wird per Copykonstruktor auf der Basis von x erzeugt

x = y           // y wird x zugewiesen (keine Erstellung)
MyClass y = x   // Falle: identisch zu MyClass y(x); hier
                //wird der Copykonstruktor aufgerufen , keine Zuweisung
```

- 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 std::map der Parameter ein `const string &`.
- Syntaktisch wird die Definition von Präfix bzw. Postfix durch den übergebenen ‘int unterschieden, er hat ansonsten keine weitere Bedeutung und wird meistens nicht benannt. Um den semantischen Unterschied zwischen den Präfix- und PostfixVarianten 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:

Code

```
// Präfix -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.
- Operatoren, die außerhalb der Klasse deklariert werden und auf Member zugreifen wollen, müssen ggf. als `friend` deklariert werden oder eine Memberfunktion zum Zugriff benutzen.
- Der Ausgabeoperator `operator<<()` sollte eine Referenz auf `ostream` zurückliefern, damit Anweisungen wie `cout << a << b;` möglich sind, das nämlich zu `(cout.operator<<(a)).operator<<(b);` übersetzt wird. Gleiches gilt für den Eingabeoperator.

Hinweis zur Operatorüberladung in Jupyter Notebooks

Eine Entwicklungsumgebung sollte idealerweise keinen negativen Einfluss auf die Implementierung haben, was bisher bei Ihnen mit JupyterLab hoffentlich zutraf, insbesondere weil keine individuelle Softwareinstallation notwendig ist. Im Fall der Operatorüberladung besteht allerdings im Modul Xeus-Cling, welches wir zur Kompilierung der C++-Dateien verwenden (da JupyterLab alleine nur Python unterstützt) ein Bug, der folgenden Workaraound notwendig macht:

Code

```
#define OPERATOR operator<<

std::ostream& OPERATOR (std::ostream & os, const Color& c)
{
    os << c.r << ", " << c.g << ", " << c.b << std::endl;
    return os;
}

#undef OPERATOR
```

Übungsaufgaben zu diesem Kapitel finden Sie **hier**.

Übungsaufgaben zu Vererbung

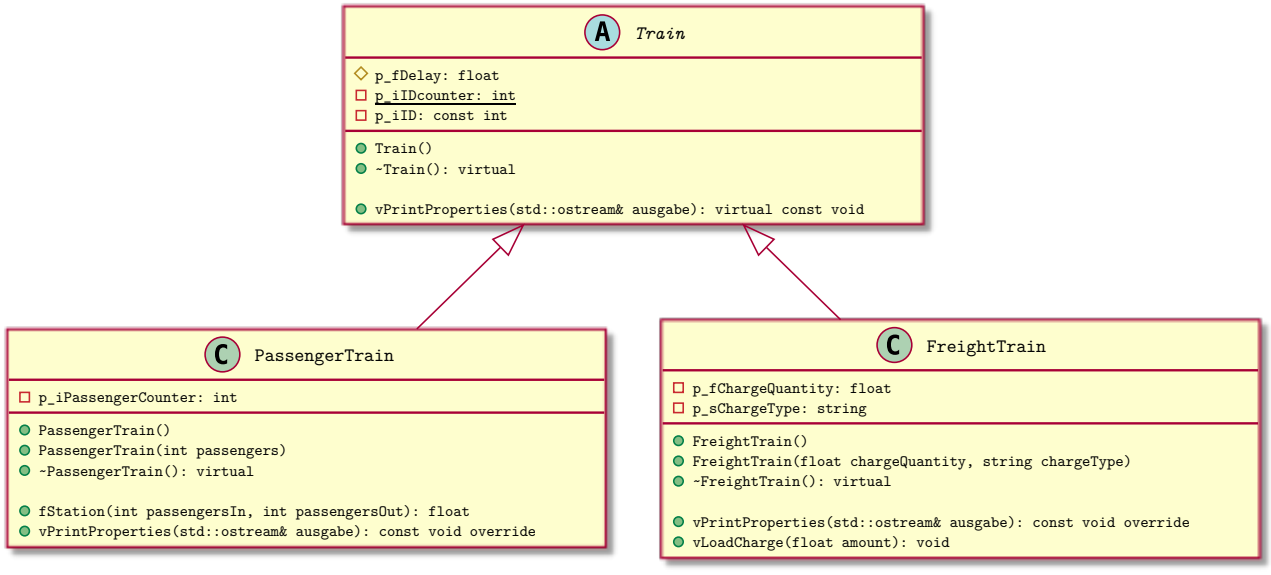
- Praktische Aufgabe
  - Unterklassen
  - Klassenspezifische Methoden
  - Operatorüberladung
- Verständnisfragen
  - Unterklassen
  - Zugriffsspezifizierer
  - Operatorüberladung

Im vorangegangenen Abschnitt "Vererbung" wurde Ihnen nähergebracht, wie eine Vererbungsstruktur in C++ realisiert werden kann. Zusammenfassend lässt sich sagen, dass eine von einer Basisklasse erbende Unterklasse sämtliche Methoden und Instanzvariablen jener Basisklasse übernimmt und um eigene Definitionen ergänzt. Der Zugriffsspezifizierer (`protected` oder `public`) bestimmt, auf welche Instanzvariablen der Basisklasse von der Unterklasse aus zugegriffen werden kann.

In diesem Kontext können Methoden mit gleichem Namen auftreten, was auf die Mechanismen **Vererbung** (in Unterklasse), **Polymorphie** (in unterschiedlichen Klassen) oder **Überladung** (unterschiedliche Argumente) zurückgeführt werden kann. Es wurde genauer auf die Überladung von Operatoren eingegangen, wobei es einige Besonderheiten, auch in Bezug auf Jupyter Notebooks, zu beachten gibt.

Praktische Aufgabe

Hier soll nun das Zugbeispiel aus dem vorangegangen Kapitel zu Klassen fortgeführt werden. Wie bereits im einführenden Beispiel angedeutet, kann die `Train`-Klasse durch die Einführung von Unterklassen für bestimmte Typen von Zügen weiter diversifiziert werden. Konkret bietet sich hier die Unterteilung in Personenzüge `PassengerTrain` und Güterzüge `FreightTrain`an, die über jeweils passende Instanzvariablen und entsprechende Methoden verfügen.



Unterklassen

Implementieren Sie die Klasse `PassengerTrain`, die von `Train` auf die übliche Weise erbt. Damit nur diese Klasse über die Instanzvariable `p_iPassengerCounter` verfügt, kopieren Sie deren Implementierung von der Ober- in die Unterklasse und löschen sie aus `Train`. Gleiches gilt für die zugehörige Methode `fStation`.

Die Klasse `FreightTrain` wiederum soll ebenfalls von `Train` erben, aber die Instanzvariablen `p_fChargeQuantity` und `p_sChargeType` beinhalten. Standardmäßig sollen diese mit `0.0` bzw. `"empty"` initialisiert werden. Damit der Datentyp `String` verwendet werden kann, müssen Sie diesen mit `#include <string>` explizit einbinden.

Überschreiben Sie in den Unterklassen die Methode `vPrintProperties()` von `Train`, in der die jeweils hinzugekommenen Instanzvariablen ausgegeben werden. Um Redundanzen bei der Ausgabe der gemeinsamen Instanzvariablen (also derjenigen in der Oberklasse `Train`) zu vermeiden, wird als erste Operation in den Ausgabemethoden der Unterklasse `Train::vPrintProperties()`aufgerufen.

Erstellen Sie auch für beide Unterklassen die parametrisierten Konstruktoren, erzeugen damit jeweils einen `FreightTrain` sowie `PassengerTrain` und testen die Ausgabemethoden.

zum Lösungsvorschlag

Klassenspezifische Methoden

Die Verzögerung pro Zug soll nun gespeichert werden, anstatt nur von der Methode `fStation()` zurückgegeben zu werden. Legen Sie dafür eine neue Instanzvariable `p_fDelay` in `Train` an, die mit 0 initialisiert wird. Überlegen Sie sich unter Berücksichtigung des folgenden Absatzes, welcher Zugriffsspezifizierer für `p_fDelay` erforderlich ist.

Verrechnen Sie in der Methode `PassengerTrain::fStation` den aktuellen Wert von `p_fDelay` mit der Umsteigezeit und schreiben Sie diesen in `p_fDelay`, wobei die Verspätung allerdings niemals kleiner als `0` werden kann. Zurückgegeben wird weiterhin die berechnete Umsteigezeit.

Eine ähnliche Methode soll für die Klasse `FreightTrain` implementiert werden, die als Argument die Menge der auf- bzw. abzuladenden Güter als `float` übergeben bekommt. Falls nicht mehr abgeladen werden soll, als der Zug laut `p_fChargeQuantity` geladen hat, wird die aktualisierte Gütermenge in dieser Instanzvariable gespeichert. Geben Sie abschließend die ausgeführte Operation (auf- oder abladen, Menge, Typ) in Textform aus.

zum Lösungsvorschlag

Operatorüberladung

Schließlich soll das Konzept der Operatorüberladung auf den Ausgabeoperator der beiden Unterklassen angewandt werden, wobei deren jeweilige Instanzvariablen ausgegeben werden.

Durch die Überladung wird nun folgendes Konstrukt zur Ausgabe der Daten eines Zuges möglich: `std::cout << aPassengerTrain << std::endl`, also ohne direkten Aufruf der `vPrintProperties()` Methode. Da Sie diese allerdings schon implementiert haben, bietet es sich an, eben jene Methode im überladenen Operator aufzurufen.

Darüberhinaus kann die Methode `Train::vPrintProperties(...)` als `virtual` deklariert werden, um sicherzustellen, dass von `Train` erbende Klassen diese überschreiben. Wie auch die Ausgabemethoden in den Unterklassen muss ihr ebenfalls ein Parameter vom Typ `std::ostream&` hinzugefügt werden, dessen Argument beim Aufruf mittels `<<` stets `std::ostream` ist.

Passen Sie abschließend die Ausgabe in `main.cpp` entsprechend an.

zum Lösungsvorschlag

Code

```
#include <iostream>
```

Code

```
//Train.h
//hier: Version aus vorheriger Lektion aktualisieren
 class Train
{
    private:
        int p_iPassengerCounter = 0;

        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;
    public:
        Train() = default;
        Train(int passenger);
        virtual ~Train() = default;

        void vPrintProperties() const;
        float station(int passengersIn, int passengersOut);
}
```

Code

```
//Train.cpp
//hier: Version aus vorheriger Lektion aktualisieren
Train::Train(int passenger): p_iPassengerCounter(passenger)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
//Train.cpp
//hier: Ausgabemethode aus vorheriger Lektion aktualisieren
void Train::vPrintProperties() const
{
    std::cout << "ID: " << p_iID << std::endl;

    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
//PassengerTrain.h
//hier: Klasse PassengerTrain implementieren
```

Code

```
//PassengerTrain.cpp
//hier: Konstruktor für PassengerTrain implementieren
```

Code

```
//PassengerTrain.cpp
//hier: Ausgabemethode implementieren
```

Code

```
//PassengerTrain.cpp
//hier: fStation(...) implementieren
```

Code

```
//FreightTrain.h
//hier: Klasse FreightTrain implementieren
```

Code

```
//FreightTrain.cpp
//hier: Konstruktor für FreightTrain implementieren
```

Code

```
//FreightTrain.cpp
//hier: Ausgabemethode implementieren
```

Code

```
//FreightTrain.cpp
//hier: vLoadCharge(...) implementieren
```

Code

```
//main.cpp
//hier: Ausgabeoperator überladen
#define OPERATOR operator<<

#undef OPERATOR
```

Code

```
//main.cpp
//hier: Instanzen erzeugen, Methoden aufrufen und Ausgabe testen
```

Lösungsvorschlag zu 1.

zur Aufgabenstellung

Code

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

Code

```
//Train.h
class Train
{
    private:
        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;

    public:
        Train() = default;
        Train(int speed, bool electric);
        virtual ~Train() = default;

        void vPrintProperties() const;
}
```

Code

```
//Train.cpp
void Train::vPrintProperties() const
{
    std::cout << "ID: " << p_iID << std::endl;

    //ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl; -> jetzt in PassengerTrain
}
```

Code

```
//PassengerTrain.h
class PassengerTrain : public Train
{
    public:
        PassengerTrain() = default;
        PassengerTrain(int p_iPassengerCounter);
        virtual ~PassengerTrain() = default;

        void vPrintProperties() const;
        float fStation(int passengersIn, int passengersOut);

    private:
        int p_iPassengerCounter = 0;

}
```

Code

```
//PassengerTrain.cpp
PassengerTrain::PassengerTrain(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
//PassengerTrain.cpp
void PassengerTrain::vPrintProperties() const
{
    Train::vPrintProperties();
    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
//PassengerTrain.cpp
// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/

float PassengerTrain::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Sekunden, Umrechnung in Minuten spaeter
    float secondsChange = 120 - passengersIn * 10 - passengersOut * 5;
    return secondsChange/60;
}
```

Code

```
//FreightTrain.h
class FreightTrain : public Train
{
    private:
        float p_fChargeQuantity = 0.0;
        std::string p_sChargeType = "default";

    public:
        FreightTrain() = default;
        FreightTrain(float chargeQuantity, std::string chargeType);
        virtual ~FreightTrain() = default;

        void vPrintProperties() const;
}
```

Code

```
//FreightTrain.cpp
FreightTrain::FreightTrain(float chargeQuantity, std::string chargeType): p_fChargeQuantity(chargeQuantity),
 ↪  p_sChargeType(chargeType)
{}
```

Code

```
//FreightTrain.cpp
void FreightTrain::vPrintProperties() const
{
    Train::vPrintProperties();
    std::cout << "Ladung: " << p_fChargeQuantity << " " << p_sChargeType << std::endl;
}
```

Code

```
//main.cpp
PassengerTrain aTrain = PassengerTrain(30);
FreightTrain bTrain = FreightTrain(2.0, "Kisten Schokolade");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
aTrain.vPrintProperties();

for(int i = 0; i < 3; i++)
{
    std::cout << std::endl << "Eigenschaften 'aTrain' nach "<< i+1 << ". Aufruf von Station: " << std::endl;
    aTrain.fStation(3, 10);
    aTrain.vPrintProperties();
}
```

Output

```
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
ID: 0
Anzahl Passagiere: 30

Eigenschaften 'aTrain' nach 1. Aufruf von Station:
ID: 0
Anzahl Passagiere: 23

Eigenschaften 'aTrain' nach 2. Aufruf von Station:
ID: 0
Anzahl Passagiere: 16

Eigenschaften 'aTrain' nach 3. Aufruf von Station:
ID: 0
Anzahl Passagiere: 9
```

Lösungsvorschlag zu 2.

zur Aufgabenstellung

Code

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

Code

```
//Train.h
class Train
{
    private:
        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;

    protected:
        float p_fDelay = 0;

    public:
        Train() = default;
        Train(int speed, bool electric);
        virtual ~Train() = default;

        void vPrintProperties() const;
}
```

Code

```
//Train.cpp
void Train::vPrintProperties() const
{
    std::cout << "ID: " << p_iID << std::endl;

    //ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl; -> jetzt in PassengerTrain
    std::cout << "momentane Verspätung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
//PassengerTrain.h
class PassengerTrain : public Train
{
    public:
        PassengerTrain() = default;
        PassengerTrain(int p_iPassengerCounter);
        virtual ~PassengerTrain() = default;

        void vPrintProperties() const;
        float fStation(int passengersIn, int passengersOut);

    private:
        int p_iPassengerCounter = 0;

}
```

Code

```
//PassengerTrain.cpp
PassengerTrain::PassengerTrain(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
//PassengerTrain.cpp
void PassengerTrain::vPrintProperties() const
{
    Train::vPrintProperties();
    std::cout << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
//PassengerTrain.cpp
// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/

float PassengerTrain::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Sekunden, Umrechnung in Minuten spaeter
    float secondsChange = 120 - passengersIn * 10 - passengersOut * 5;
    p_fDelay = (p_fDelay - secondsChange < 0) ? 0 : (p_fDelay - secondsChange)/60;

    return secondsChange/60;
}
```

Code

```
//FreightTrain.h
class FreightTrain : public Train
{
    private:
        float p_fChargeQuantity = 0.0;
        std::string p_sChargeType = "default";

    public:
        FreightTrain() = default;
        FreightTrain(float chargeQuantity, std::string chargeType);
        virtual ~FreightTrain() = default;

        void vPrintProperties() const;
        void vLoadCharge(float amount);
}
```

Code

```
//FreightTrain.cpp
FreightTrain::FreightTrain(float chargeQuantity, std::string chargeType) : p_fChargeQuantity(chargeQuantity),
↪  p_sChargeType(chargeType)
{}
```

Code

```
//FreightTrain.cpp
void FreightTrain::vPrintProperties() const
{
    Train::vPrintProperties();
    std::cout << "Ladung: " << p_fChargeQuantity << " " << p_sChargeType << std::endl;
}
```

Code

```
//FreightTrain.cpp
// abladen: amount < 0
void FreightTrain::vLoadCharge(float amount)
{
    if (p_fChargeQuantity + amount < 0)
    {
        std::cout << "Es kann nicht mehr abgeladen werden als geladen ist. Abbruch." << std::endl;
        return;
    }
    p_fChargeQuantity += amount;
    std::cout << "Es wurden " << amount << " " << p_sChargeType;

    if (amount > 0) std::cout << " aufgeladen.";
    else std::cout << " abgeladen.";
}
```

Code

```
//main.cpp
PassengerTrain aTrain = PassengerTrain(30);
FreightTrain bTrain = FreightTrain(2.0, "Kisten Schokolade");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
aTrain.vPrintProperties();

for(int i = 0; i < 3; i++)
{
    std::cout << std::endl << "Eigenschaften 'aTrain' nach "<< i+1 << ". Aufruf von Station: " << std::endl;
    aTrain.fStation(13, 20);
    aTrain.vPrintProperties();
}

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
bTrain.vPrintProperties();

std::cout << std::endl << "\n\n\nEigenschaften 'bTrain' nach 1. load:" << std::endl;
bTrain.vLoadCharge(-10);
bTrain.vPrintProperties();

std::cout << std::endl << "\n\n\nEigenschaften 'bTrain' nach 2. load:" << std::endl;
bTrain.vLoadCharge(3.3);
bTrain.vPrintProperties();
```

Output

```
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
ID: 0
momentane Verspätung: 0 Minuten
Anzahl Passagiere: 30

Eigenschaften 'aTrain' nach 1. Aufruf von Station:
ID: 0
momentane Verspätung: 1.83333 Minuten
Anzahl Passagiere: 23

Eigenschaften 'aTrain' nach 2. Aufruf von Station:
ID: 0
momentane Verspätung: 1.86389 Minuten
Anzahl Passagiere: 16

Eigenschaften 'aTrain' nach 3. Aufruf von Station:
Es koennen nicht mehr Personen aussteigen als sich im Zug befinden
ID: 0
momentane Verspätung: 1.86389 Minuten
Anzahl Passagiere: 16

Eigenschaften 'bTrain':
ID: 1
momentane Verspätung: 0 Minuten
Ladung: 2 Kisten Schokolade



Eigenschaften 'bTrain' nach 1. load:
Es kann nicht mehr abgeladen werden als geladen ist. Abbruch.
ID: 1
momentane Verspätung: 0 Minuten
Ladung: 2 Kisten Schokolade



Eigenschaften 'bTrain' nach 2. load:
Es wurden 3.3 Kisten Schokolade aufgeladen.ID: 1
momentane Verspätung: 0 Minuten
Ladung: 5.3 Kisten Schokolade
```

Lösungsvorschlag zu 3.

zur Aufgabenstellung

Code

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

Code

```
//Train.h
class Train
{
    private:
        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;

    protected:
        float p_fDelay = 0;

    public:
        Train() = default;
        Train(int speed, bool electric);
        virtual ~Train() = default;

        virtual void vPrintProperties(std::ostream& ausgabe) const;
}
```

Code

```
//Train.cpp
void Train::vPrintProperties(std::ostream& ausgabe) const
{
    ausgabe << "ID: " << p_iID << std::endl;

    //ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl; -> jetzt in PassengerTrain
    ausgabe << "momentane Verspätung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
//PassengerTrain.h
class PassengerTrain : public Train
{
    public:
        PassengerTrain() = default;
        PassengerTrain(int p_iPassengerCounter);
        virtual ~PassengerTrain() = default;

        void vPrintProperties(std::ostream& ausgabe) const;
        float fStation(int passengersIn, int passengersOut);

    private:
        int p_iPassengerCounter = 0;

}
```

Code

```
//PassengerTrain.cpp
PassengerTrain::PassengerTrain(int passengers) : p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
//PassengerTrain.cpp
void PassengerTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
//PassengerTrain.cpp
// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/

float PassengerTrain::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Sekunden, Umrechnung in Minuten spaeter
    float secondsChange = 120 - passengersIn * 10 - passengersOut * 5;
    p_fDelay = (p_fDelay - secondsChange < 0) ? 0 : (p_fDelay - secondsChange)/60;

    return secondsChange/60;
}
```

Code

```
//FreightTrain.h
class FreightTrain : public Train
{
    private:
        float p_fChargeQuantity = 0.0;
        std::string p_sChargeType = "default";

    public:
        FreightTrain() = default;
        FreightTrain(float chargeQuantity, std::string chargeType);
        virtual ~FreightTrain() = default;

        void vPrintProperties(std::ostream& ausgabe) const;
        void vLoadCharge(float amount);
}
```

Code

```
//FreightTrain.cpp
FreightTrain::FreightTrain(float chargeQuantity, std::string chargeType) : p_fChargeQuantity(chargeQuantity),
↪  p_sChargeType(chargeType)
{}
```

Code

```
//FreightTrain.cpp
void FreightTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Ladung: " << p_fChargeQuantity << " " << p_sChargeType << std::endl;
}
```

Code

```
//FreightTrain.cpp
// abladen: amount < 0
void FreightTrain::vLoadCharge(float amount)
{
    if (p_fChargeQuantity + amount < 0)
    {
        std::cout << "Es kann nicht mehr abgeladen werden als geladen ist. Abbruch." << std::endl;
        return;
    }
    p_fChargeQuantity += amount;
    std::cout << "Es wurden " << amount << " " << p_sChargeType;

    if (amount > 0) std::cout << " aufgeladen.";
    else std::cout << " abgeladen.";
}
```

Code

```
#define OPERATOR operator<<

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR
```

Hinweis: Der Block oben sollte eigentlich so aussehen:

Code

```
std::ostream & operator<<(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}
```

Der Rest ist nur in Jupyter notwendig

Code

```
//main.cpp
PassengerTrain aTrain = PassengerTrain(30);
FreightTrain bTrain = FreightTrain(2.0, "Kisten Schokolade");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
std::cout << aTrain;

for(int i = 0; i < 3; i++)
{
    std::cout << std::endl << "Eigenschaften 'aTrain' nach "<< i+1 << ". Aufruf von Station: " << std::endl;
    aTrain.fStation(13, 20);
    std::cout << aTrain;
}

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
std::cout << bTrain;

std::cout << std::endl << "\n\n\nEigenschaften 'bTrain' nach 1. load:" << std::endl;
bTrain.vLoadCharge(-10);
std::cout << bTrain;

std::cout << std::endl << "\n\n\nEigenschaften 'bTrain' nach 2. load:" << std::endl;
bTrain.vLoadCharge(3.3);
std::cout << bTrain;
```

Output

```
Aufruf des Nicht-Standardkonstruktors

Eigenschaften 'aTrain':
ID: 0
momentane Verspätung: 0 Minuten
Anzahl Passagiere: 30

Eigenschaften 'aTrain' nach 1. Aufruf von Station:
ID: 0
momentane Verspätung: 1.83333 Minuten
Anzahl Passagiere: 23

Eigenschaften 'aTrain' nach 2. Aufruf von Station:
ID: 0
momentane Verspätung: 1.86389 Minuten
Anzahl Passagiere: 16

Eigenschaften 'aTrain' nach 3. Aufruf von Station:
Es koennen nicht mehr Personen aussteigen als sich im Zug befinden
ID: 0
momentane Verspätung: 1.86389 Minuten
Anzahl Passagiere: 16

Eigenschaften 'bTrain':
ID: 1
momentane Verspätung: 0 Minuten
Ladung: 2 Kisten Schokolade



Eigenschaften 'bTrain' nach 1. load:
Es kann nicht mehr abgeladen werden als geladen ist. Abbruch.
ID: 1
momentane Verspätung: 0 Minuten
Ladung: 2 Kisten Schokolade



Eigenschaften 'bTrain' nach 2. load:
Es wurden 3.3 Kisten Schokolade aufgeladen.ID: 1
momentane Verspätung: 0 Minuten
Ladung: 5.3 Kisten Schokolade
```

Verständnisfragen

Unterklassen

Bei welchen Anwendungsfällen kann es sinvoll sein, trotz fertig implementierter Unterklassen Variablen oder Methoden vom Typ der Basisklasse zu verwenden?

Zugriffsspezifizierer

Warum muss für `p_iPassengerCounter` ein anderer Zugriffsspezifizierer als für die restlichen Instanzvariablen von `Train` verwendet werden? Welche anderen Möglichkeiten gäbe es, um die gewünschte Funktionalität zu erreichen?

Operatorüberladung

Einen Operator zu überladen, ist nicht nur beim Programmieren eigener Klassen nützlich, sondern wird auch in einigen integrierten Strukturen von C bzw. C++ angewandt. Welche Situationen dafür fallen Ihnen ein?

Lösungsvorschläge zu den Verständnisfragen

1. Unterklassen

Der am einfachsten nachzuvollziehende Fall läge sicherlich vor, wenn sich der Endanwender nicht für eine der Unterklassen entscheiden will. Hier könnte das beispielsweise bei einem System zur Simulation der Schienenauslastung zutreffen, falls es für die Belegung eines Schienenwegs der Typ des Zugs unerheblich ist.

Es kommt außerdem häufig vor, dass allgemeinere Datenstrukturen notwendig sind, um die Instanzen der verschiedenen Unterklassen verwalten zu können.

Vorstellbar wäre im vorliegenden Fall beispielsweise eine Liste von Güter- und Personenzügen, die von einer Bahnbetriebsgesellschaft geführt wird. Damit sie beide Typen von Zügen enthalten kann, müsste die Liste vom Typ `Train` sein. Um dann alle Elemente dieser Liste automatisiert ausgeben zu können, würde ein Aufruf von `vPrintProperties()` beziehungsweise des Ausgabeoperators ausreichen. Dass diese Operationen auch von den Unterklassen implementiert sind, wird durch das Keyword `virtual` bei der Methode in der Basisklasse `Train` sichergestellt wird. Die Erstellung und Verwaltung von Listen wird in Lektion 5 (Containerklassen) genauer erläutert.

2. Zugriffsspezifizierer

`p_iPassengerCounter` muss `protected` sein, damit in `PassengerTrain::fStation` darauf lesend und schreibend zugegriffen werden kann.

Alternativ könnte diese Methode in `Train` als `friend` deklariert werden, sodass sie auch auf `private` Attribute zugreifen kann. Eine weitere Möglichkeit bestünde in der Definition von `public` Getter- und Setter-Methoden, die dann in `PassengerTrain::fStation` aufgerufen werden müssten.

3. Operatorüberladung

Wie die meisten anderen Programmiersprachen unterscheidet auch C++ für den Endanwender auf den ersten Blick nicht zwischen mathematischen Operationen auf Zahlen vom Datentyp `int`, `float` oder Mischungen davon. Zu beachten ist allerdings, dass bei letzterem Fall, also bspw. der Addition von `int` und `float` Werten, eine Typumwandlung stattfindet. Konkret würde hier der `float` in eine Ganzzahlt konvertiert werden, indem lediglich die Stellen vor dem Komma berücksichtigt werden. Auf diese Weise können unter Umständen unerwünschte Effekte auftreten, da die Genauigkeit des Ergebnisses abnehmen kann.

Eine weitere Überladung liegt beim Ausgabeoperator vor, den Sie auch in dieser Lektion sowohl zum Ausgeben von `String`

Smart Pointer

- Einführung/Überblick
- unique-pointer
- shared-pointer
- weak-pointer

Einführung/Überblick

In neuen Versionen von C++ findet man eine Alternative zu `new/delete` Funktionen, die die Effizienz dynamischer Objekte mit dem Besitzverhalten eines Wertes oder Containers kombinieren. Dadurch kann das korrekte Löschen der Objekte automatisert werden, wenn keine Referenzen auf das Objekt mehr bestehen:

- `unique_ptr`, die eindeutig auf ein Objekt zeigen (und den Besitz dieses Objekts haben)
- `shared_ptr`, die ggf. mehrfach auf ein Objekt zeigen (und alle den Besitz dieses Objekts teilen)
- `weak_ptr`, die aus einem `shared_ptr` erzeugt werden (und keinen Besitz an diesem Objekt haben)

Dereferenzieren funktioniert, außer bei `weak_ptr`, genauso wie bei einfachen Pointern, `get()` gibt einen einfachen Pointer auf das Objekt zurück. Bei `weak_ptr` erfolgt der Zugriff auf das Objekt durch Aufruf der Funktion `lock()` , um zu testen, ob der zugehörige `shared_ptr` noch gültig ist. Für die Nutzung der Smart Pointer muss der Header `<memory>` eingebunden werden. Die Smart Pointer sind Teil der Standartbibliothek `std`. Wenn der Smart Pointer erstellt wird, wird Speicherplatz reserviert und das Objekt konstruiert. Smart Pointer selber werden wie andere Daten automatisch gelöscht, wenn sie die Gültigkeit verlieren. Gegenüber klassischer Speicherverwaltung mit `new` und `delete` haben Smart Pointer mehrere Vorteile:

- Sie werden automatisch zerstört und geben ihren Speicher frei. Manuelle Aufrufe von `delete` sind nicht notwendig. Allerdings sind ggf. Besitzzyklen zu vermeiden (siehe weak_ptr).
- Sie sind Exception-sicher. Das heißt, im Falle einer Exception wird sämtlicher angeforderter Speicher wieder freigegeben.
- Der Besitz ist eindeutig. Wenn stattdessen einfache Pointer für die Speicherverwaltung benutzt werden, ist nicht klar, ob das Objekt, auf das ein Pointer zeigt, gelöscht werden soll oder es noch weitere Referenzen gibt.

Smart Pointer können auch in Feldern gespeichert werden, etwa zur Verwaltung mehrerer polymorpher Objekte. Während bei klassischen Pointern die Objekte des Feldes explizit gelöscht werden müssen, werden bei Smart Pointern die Destruktoren korrekt aufgerufen.

unique-pointer

`unique-pointer` sind sehr ressourcenschonend: Sie brauchen genauso viel Speicherplatz wie ein normaler Pointer. Wie der Name schon verrät, hat das Objekt, auf den der `unique-ptr` zeigt, genau einen Besitzer. Das heißt, dass es genau eine Instanz eines Objekt gibt. **Dementsprechend kann dieser Pointer auch nicht einfach in einen anderen Pointer kopiert werden.** Eine Alternative ist der `move()`-Befehl. Dieser überträgt den Wert des einen Pointers in den gewünschten neuen Pointer oder die gewünschte Funktion. Der ursprüngliche Pointer ist danach ein `nullptr` und wird nach Beendigung der Funktion/des Programms gelöscht. Genauso kann man den Wert des Pointers (also das Objekt, auf das gezeigt wird) nicht per `call by value` übergeben, sondern muss per `call by reference` übergeben werden. Es ist zu empfehlen, den Befehl `make_unique()` beim Erzeugen eines Objektes zu benutzen. Wird der `unique_ptr` gelöscht, so wird auch das Objekt zerstört.

Beispiel:

Code

```
#include <memory>
#include <iostream>
```

Code

```
int squareIt(std::unique_ptr<int> s) {
    int squared = *s;
    return squared*squared;
}
```

Code

```
{
    std::unique_ptr<int> number = std::make_unique<int>(15);

    if(number){
        std::cout << "Der Pointer `number` existiert noch und hat den Inhalt: " << *number << std::endl;
    } else {
        std::cout << "Der Pointer `number` existiert nicht mehr." << std::endl;
    }

    int numberSquared = squareIt(move(number));
    std::cout << "Der Wert des Pointers wurde zur Berechnung übergeben: " << numberSquared << std::endl;

    if(number){
        std::cout << "Der Pointer `number` existiert noch und hat den Inhalt: " << *number << std::endl;
    } else {
        std::cout << "Der Pointer `number` existiert nicht mehr." << std::endl;
    }
}
```

Man kann die Pointer auch in Containerklassen (siehe Abschnitt zu Containerklassen für mehr Informationen darüber) speichern.
Ein Beispiel dafür:

Code

```
#include <memory>
#include <string>
#include <iostream>
#include <vector>
```

Code

```
class Animal{
    private:
        std::string species;
        int age;
        double weight;
    public:
        std::string getSpecies(){return species;}
        int getAge(){return age;}
        double getWeight() {return weight;}
        Animal(){};
        Animal(std::string species, int age, double weight);
};
```

Code

```
Animal::Animal(std::string species, int age, double weight):
    species(species), age(age), weight(weight)
{

}
```

Code

```
{
    std::vector<std::unique_ptr<Animal>> animalVector;
    std::unique_ptr<Animal> cat = std::make_unique<Animal>("Cat", 7, 3.9);
    std::unique_ptr<Animal> dog = std::make_unique<Animal>("Dog", 11, 36.4);
    std::unique_ptr<Animal> turtle = std::make_unique<Animal>("Turtle", 52, 1.5);
    animalVector.push_back(move(cat));
    animalVector.push_back(move(dog));
    animalVector.push_back(move(turtle));

    if(cat){
        std::cout << "Der Pointer `cat` existiert noch mit Inhalt: " << cat->getSpecies() << ", age: " << cat->getAge() << ",
↪   weight: " << cat->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Unique_Ptr `cat` ist jetzt nullpointer." << std::endl;
    }

    if(dog){
        std::cout << "Der Pointer `dog` existiert noch mit Inhalt: " << dog->getSpecies() << ", age: " << dog->getAge() << ",
↪   weight: " << dog->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Unique_Ptr `dog` ist jetzt nullpointer." << std::endl;
    }
    if(turtle){
        std::cout << "Der Pointer `turtle` existiert noch mit Inhalt: " << turtle->getSpecies() << ", age: " << turtle->getAge()
↪   << ", weight: " << turtle->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Unique_Ptr `turtle` ist jetzt nullpointer." << std::endl;
    }


    for(auto& i : animalVector){
        std::cout << "Species: " << i->getSpecies() << ", Age: " << i->getAge() << " years,  Weight: " << i->getWeight() << "
↪   kg" << std::endl;
    }
}
```

shared-pointer

`shared_ptr` sind ähnlich zu `unique_ptr`, haben aber den großen Unterschied, dass ein Objekt mehr als einen Besitzer haben kann. Das heißt allerdings im Umkehrschluss, dass `shared_ptr` einen höheren Speicherbedarf und Verwaltungsaufwand haben als `unique_ptr` (**Es ist also empfehlenswert, `unique_ptr` überall zu nutzen, wo es möglich ist**). Die Anzahl der Referenzen/Besitzer, die auf das Objekt zeigen, werden entsprechend hoch- und runtergezählt. Wenn dieser Counter auf 0 gefallen ist (es zeigt kein `shared_ptr` mehr auf dieses Objekt), wird das Objekt gelöscht. Daduruch können `shared_ptr` auch kopiert und *ohne* den `move()`-Befehl an Funktionen oder Container übergeben werden. Das heißt, dass `shared_ptr` ihren Wert auch per `call by value` übergeben können (bspw. an Funktionen) und nicht nur per `pass by reference`. Es ist zu empfehlen, den Befehl `make_shared()` zum Erzeugen eines Shared-Pointers zu benutzen.

Das obige Beispiel sieht nun so aus:

Code

```
#include <iostream>
#include <memory>
```

Code

```
int squareIt(std::shared_ptr<int>& s) {
    return (*s)*(*s);
}
```

Code

```
{
    std::shared_ptr<int> i_number = std::make_shared<int>(15);

    if(i_number){
        std::cout << "Der Pointer `i_number` existiert noch und hat den Inhalt: " << *i_number << std::endl;
    } else {
        std::cout << "Der Pointer `i_number` existiert nicht mehr." << std::endl;
    }

    int numberSquared = squareIt(i_number);
    std::cout << "Der Wert des Pointers wurde zur Berechnung übergeben: " << numberSquared << std::endl;

    if(i_number){
        std::cout << "Der Pointer `i_number` existiert noch und hat den Inhalt: " << *i_number << std::endl;
    } else {
        std::cout << "Der Pointer `i_number` existiert nicht mehr." << std::endl;
    }
}
```

Genauso lassen sich `shared_ptr` ebenfalls in Containerklassen (siehe Abschnitt zu Containerklassen für mehr Informationen darüber) speichern.
Das obige Beispiel sieht mit `shared_ptr` so aus:

Code

```
#include <memory>
#include <string>
#include <iostream>
#include <vector>
```

Code

```
class Animal{
    private:
        std::string species;
        int age;
        double weight;
    public:
        std::string getSpecies(){return species;}
        int getAge(){return age;}
        double getWeight() {return weight;}
        Animal(){};
        Animal(std::string species, int age, double weight);
};
```

Code

```
Animal::Animal(std::string species, int age, double weight):
    species(species), age(age), weight(weight)
{

}
```

Code

```
{
    std::vector<std::shared_ptr<Animal>> animalVector;
    std::shared_ptr<Animal> cat = std::make_shared<Animal>("Cat", 7, 3.9);
    std::shared_ptr<Animal> dog = std::make_shared<Animal>("Dog", 11, 36.4);
    std::shared_ptr<Animal> turtle = std::make_shared<Animal>("Turtle", 52, 1.5);
    animalVector.push_back(cat);
    animalVector.push_back(dog);
    animalVector.push_back(turtle);

    if(cat){
        std::cout << "Der Pointer `cat` existiert noch mit Inhalt: " << cat->getSpecies() << ", age: " << cat->getAge() << ",
↪   weight: " << cat->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Pointer `cat` ist jetzt nullpointer." << std::endl;
    }

    if(dog){
        std::cout << "Der Pointer `dog` existiert noch mit Inhalt: "  << dog->getSpecies() << ", age: " << dog->getAge() << ",
↪   weight: " << dog->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Pointer `dog` ist jetzt nullpointer." << std::endl;
    }
    if(turtle){
        std::cout << "Der Pointer `turtle` existiert noch mit Inhalt: " << turtle->getSpecies() << ", age: " << turtle->getAge()
↪   << ", weight: " << turtle->getWeight() << std::endl;
    } else {
        std::cout << "Der ursprüngliche Pointer `turtle` ist jetzt nullpointer." << std::endl;
    }


    for(auto& i : animalVector){
        std::cout << "Species: " << i->getSpecies() << ", Age: " << i->getAge() << " years,  Weight: " << i->getWeight() << "
↪   kg" << std::endl;
    }
}
```

weak-pointer

`weak_ptr` sind eine spezielle Art `shared_ptr`: Sie existieren nur, wenn sie mit einem zugehörigen `shared_ptr` verbunden sind. Sie ermöglichen Zugriff auf ein Objekt, erhöhen aber nicht den Referenzzähler des `shared_ptr`; sie haben also keine Besitzrechte an dem Objekt. Man benutzt `weak_ptr`, um sogenannte Besitzzyklen in Datenstrukturen zu vermeiden. Wenn keine `shared_ptr` mehr auf das Objekt zeigen, wird es gelöscht, selbst wenn noch `weak_ptr` darauf zeigen.
Das folgende Beispiel zeigt den sinnvollen Einsatz der `weak_ptr`:

Code

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

Code

```
class Person
{
    public :
        Person() = delete ;
        Person(const std::string& name , std::shared_ptr<Person> father = nullptr , std::shared_ptr<Person> mother = nullptr) ;
        virtual ~Person() ;
        void setSibling(std::weak_ptr<Person> sibling);
    private :
        std::shared_ptr<Person> p_father;
        std::shared_ptr<Person> p_mother;
        const std::string p_sName;
        std::vector<std::weak_ptr<Person>> p_siblings;
};
```

Code

```
Person::Person(const std::string& name, std::shared_ptr<Person> father, std::shared_ptr<Person> mother) :
p_sName(name), p_father(father), p_mother(mother)
{

}
```

Code

```
void Person::setSibling(std::weak_ptr<Person> brosis)
{
    p_siblings.push_back(brosis);
}
```

Code

```
Person::~Person()
{
    p_siblings.clear();
    std::cout << "Geloescht: " << p_sName << std::endl;
}
```

Code

```
void test1 ()
{
    auto m1 = std::make_shared<Person>("Josef");
    auto f1 = std::make_shared<Person>("Maria");
    auto m2 = std::make_shared<Person>("Peter", m1 , f1 );
    auto f2 = std::make_shared<Person>("Birgit", m1 , f1 );
    m2->setSibling(f2);
    f2->setSibling(m2);
}
```

Code

```
{
    std::cout << "Anfang" << std::endl;
    test1();
    std::cout << "Ende" << std::endl;
}
```

Wenn man statt `weak_ptr` im Vektor p_siblings `shared_ptr` speichert, wird der Speicher im Programm nicht automatisch freigegeben, da Peter und Birgit gegenseitig aufeinander zeigen. Sie können gerne an entsprechenden Stellen im Beispiel (in der Funktion `setSibling()` und im Vektor `p_siblings`) die `weak_ptr` durch `shared_ptr` ersetzen und schauen, was passiert. Erst wenn man durch `weak_ptr` den zyklischen Besitz auflöst, werden die Destruktoren wie erwartet aufgerufen.

Beim `weak_ptr` kann man abfragen, ob der zugehörige `shared_ptr` noch definiert ist. Dies geschieht durch die Funktion `lock()`. Sie liefert einen Nullzeiger, wenn das Objekt nicht mehr existiert. Diese Funktion muss eingesetzt werden, wenn man auf den Inhalt des `weak_ptr` zugreifen möchte. Ein `weak_ptr` in der Parameterliste (wie bei `setSibling`) macht deutlich, dass man beim Aufruf `shared_ptr` dort nicht mit `move()` einen Besitzwechsel initiieren sollte.
Ein Beispiel mit der Funktion `lock()`:

Code

```
#include <iostream>
#include <memory>
```

Code

```
void doesWeakExistProperly(std::weak_ptr<int> weak){
    if(weak.lock()){
        std::cout << "Der Pointer wurde erfolgreich gelocked und hat den Inhalt: " << *weak.lock() << std::endl;
    } else {
        std::cout << "Der Pointer konnte nicht gelocked werden." << std::endl;
    }
}
```

Code

```
{
    std::weak_ptr<int> weak;
    doesWeakExistProperly(weak);
    {
        auto shared = std::make_shared<int>(42);
        weak = shared;
        std::cout << "Der Pointer `weak` ist jetzt mit `shared` gepaired." << std::endl;
        doesWeakExistProperly(weak);
    }
    std::cout << "Der Pointer `shared` ist jetzt out of scope." << std::endl;
    doesWeakExistProperly(weak);
}
```

Übungsaufgaben zu diesem Kapitel finden Sie **hier**.

Übungsaufgaben zu Smart Pointer

Sie haben nun gelernt, was Smart Pointer sind und wie sie angewendet werden können. Im Folgenden können Sie das Gelernte mit einigen praktischen Aufgaben und Verständnisfragen überprüfen.

- Praktischen Aufgaben
- Verständnisfragen

Praktische Aufgaben

In diesem Aufgabenteil werden Sie das bereits bekannte `Train`-Beispiel um die Klasse `Station` erweitern. Dem Namen entsprechend soll die neue Klasse die Bahnhöfe simulieren, an denen unsere Züge halten werden.

Außerdem soll die Klasse `Train` um einen Smartpointer `p_pIsAt` um die Funktion `vGoto(...)` erweitert werden. Der Zeiger soll den aktuellen Aufenthaltortes des Zuges in Form eines Smartpointers auf eine Station enthalten. Die Funktion dient dazu, den aktuellen Standort, sofern möglich, auf das übergebene Ziel zu ändern.

Teil 1

Legen Sie die Klasse `Station` an. Sie soll den Namen des Bahnhofs in der Variable `p_sName` speichern und einen Pointer `p_pDestination` besitzen, in dem der Zielbahnhof gespeichert werden soll. Überlegen Sie, welcher der Ihnen nun bekannten Smart Pointer-Arten am besten dafür geeignet ist.

zum Lösungsvorschlag

Teil 2

Im nächsten Schritt sollen einige nützliche Funktionen angelegt werden. Zunächst müssen Sie einen Konstruktor und eine `getName()`-Funktion erstellen.

zum Lösungsvorschlag

Teil 3

Jetzt soll der Zielbahnhof `p_pDestination` in der Funktion `vSetDestination()` gesetzt werden. Dazu übergeben Sie einen Pointer, der diesen beinhaltet. Bedenken Sie weiterhin die Wahl des Smart Pointers! Zusätzlich muss man auch den Inhalt von `p_pDestination` auch wieder auslesen können, deswegen müssen Sie auch eine Funktion `getDestination()` implementieren.

Erweitern Sie die Klasse `Train` um einen Smartpointer `p_pIsAt`. Der Zeiger soll mit Hilfe eines Smart-Pointers den aktuellen Aufenhaltsort (Station) des Zuges verwalten. Implementieren Sie nun die Funktion `vGoTo(std::shared_ptr<Station>)`. Als Paramter soll hier ein `std::shared_ptr<Station>`, der auf das neue Ziel zeigt, verwendet werden. Die Funktion soll den aktuellen Standort des Zuges ändern, sofern eine Verbindung (`getDestination()`)von der aktuellen Station auf die neue Station vorhanden ist. Hierfür muss der Name der Stationen verglichen werden. Sollte keine Verbindung bestehen, soll eine entsprechende Fehlermeldung ausgegeben und der Aufenthaltsort nicht verändert werden. Sollte der Zug noch keinen Standort (eine Station) haben, soll der Zug in jedem Fall auf die übergebene Station gesetzt werden.

zum Lösungsvorschlag

Teil 4

Zum Schluss fügen Sie Ihre in vorherigen Aufgabenteilen geschriebenen Klassen `Train`, sowie deren Unterklassen `PassengerTrain`und `FreightTrain` an passender Stelle ein und führen alle Zellen (auch die Test-Zelle!) nacheinander aus, um die Richtigkeit Ihrer Lösungen zu überprüfen.

Code
```
#include <iostream>
#include <string>
#include <memory>
```

Code
```
//hier: Station.h implementieren
```

Code
```
//Station.cpp
//hier: Konstruktor für Station
```

Code
```
//hier: Funktion Station::getName()
```

Code
```
//hier: Funktion Station::getDestination()
```

Code
```
//hier: Funktion Station::vSetDestination()
```

Code
```
//hier: Train.h einfügen
```

Code
```
//Train.cpp
//hier: Destruktor einfügen
```

Code
```
//hier: Train::vGoTo() einfügen
```

Code
```
//hier: Train::vPrintProperties() einfügen
```

Code
```
//hier: PassengerTrain.h einfügen
```

Code
```
//PassengerTrain.cpp
//hier: Konstruktor einfügen
```

Code
```
//hier: Funktion PassengerTrain::vPrintProperties() einfügen
```

Code
```
//hier: Funktion PassengerTrain::fStation() einfügen
```

Code
```
//hier: FreightTrain.h einfügen
```

Code
```
//FreightTrain.cpp
//hier: Konstruktor einfügen
```

Code
```
//hier: Funktion FreightTrain::vPrintProperties() einfügen
```

Code
```
//hier: Funktion FreightTrain::vLoadCharge() einfügen
```

Code
```
#define OPERATOR operator<<  //Diese Zeile kann in jeder IDE weggelassen werden, sie ist nur für Jupyter notwendig

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR   //Diese Zeile kann in jeder IDE weggelassen werden, sie ist nur für Jupyter notwendig
```

Code

```
// Test

std::unique_ptr<PassengerTrain> aTrain;
aTrain =  std::make_unique<PassengerTrain>();

auto bTrain = std::make_unique<FreightTrain>(2, "tons of wood");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
std::cout << *aTrain;

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
std::cout << *bTrain << std::endl;


    auto berlin = std::make_shared<Station>("Berlin");
    auto hamburg = std::make_shared<Station>("Hamburg");
    auto frankfurt = std::make_shared<Station>("Frankfurt");
    auto koeln = std::make_shared<Station>("Köln");
    auto muenchen = std::make_shared<Station>("München");

//rudimentäre Linie aufbauen (die bekannte Ringbahn Hamburg-Hamburg)
hamburg->vSetDestination(koeln);
koeln->vSetDestination(frankfurt);
frankfurt->vSetDestination(berlin);
berlin->vSetDestination(hamburg);

std::cout << "Guten Tag meine Damen und Herren, herzlich willkommen im Zug der Deutschen Bahn von Hamburg nach Hamburg über Köln
↪  Hbf, Frankfurt Hbf und Berlin Gesundbrunnen!" << std::endl;
std::cout << std::endl;
std::cout << "Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Köln Hauptbahnhof. Wir verabschieden uns von allen
↪  Fahrgästen, die dort aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran,
↪  Ihre persönlichen Wertgegenstände mitzunehmen. Vielen Dank!" << std::endl;
aTrain->vGoTo(koeln);
std::cout << std::endl;
aTrain->vGoTo(muenchen); //Sollte nicht funktionieren, nicht Teil der Ringbahn
std::cout << std::endl;
std::cout << "Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Frankfurt Hauptbahnhof. Wir verabschieden uns von allen
↪  Fahrgästen, die dort aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran,
↪  Ihre persönlichen Wertgegenstände mitzunehmen. Vielen Dank!" << std::endl;
aTrain->vGoTo(frankfurt);
std::cout << std::endl;
std::cout << "Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Berlin Gesundbrunnen. Wir verabschieden uns von allen
↪  Fahrgästen, die dort aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran,
↪  Ihre persönlichen Wertgegenstände mitzunehmen. Vielen Dank!" << std::endl;
aTrain->vGoTo(berlin);
std::cout << std::endl;
std::cout << "Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Hamburg Hauptbahnhof. Unsere Fahrt endet dort, wir bitten
↪  alle Fahrgäste, auszusteigen." << std::endl;
aTrain->vGoTo(hamburg);
std::cout << "Wir bedanken uns für Ihre Reise mit der Deutschen Bahn." << std::endl;
```

Output

```
Eigenschaften 'aTrain':
ID: 6
momentane Verspaetung: 0 Minuten
Anzahl Passagiere: 0

Eigenschaften 'bTrain':
ID: 7
momentane Verspaetung: 0 Minuten
Ladung: 2 tons of wood

Guten Tag meine Damen und Herren, herzlich willkommen im Zug der Deutschen Bahn von Hamburg nach Hamburg über Köln Hbf, Frankfurt
↪  Hbf und Berlin Gesundbrunnen!

Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Köln Hauptbahnhof. Wir verabschieden uns von allen Fahrgästen, die dort
↪  aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran, Ihre persönlichen
↪  Wertgegenstände mitzunehmen. Vielen Dank!
Zug ist nach Köln gefahren.

Der Zug kann nicht nach München fahren: Es gibt keine Verbindung.

Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Frankfurt Hauptbahnhof. Wir verabschieden uns von allen Fahrgästen, die
↪  dort aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran, Ihre persönlichen
↪  Wertgegenstände mitzunehmen. Vielen Dank!
Zug ist nach Frankfurt gefahren.

Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Berlin Gesundbrunnen. Wir verabschieden uns von allen Fahrgästen, die
↪  dort aus- und umsteigen und wünschen eine angenehme Weiterreise. Bitte denken Sie beim Aussteigen daran, Ihre persönlichen
↪  Wertgegenstände mitzunehmen. Vielen Dank!
Zug ist nach Berlin gefahren.

Sehr geehrte Fahrgäste, in wenigen Minuten erreichen wir Hamburg Hauptbahnhof. Unsere Fahrt endet dort, wir bitten alle
↪  Fahrgäste, auszusteigen.
Zug ist nach Hamburg gefahren.
Wir bedanken uns für Ihre Reise mit der Deutschen Bahn.
```

Lösungen

Hier finden Sie eine mögliche Lösung zu den Aufgaben. Bitte beachten Sie, dass es mehrere Möglichkeiten gibt, die Aufgaben zu lösen, hier wird lediglich eine Variante aufgezeigt. Solange Ihre Implementierung funktioniert wie gefordert, ist sie wahrscheinlich richtig!

Code

```
class Station
{
    private:
        std::string p_sName;
        std::shared_ptr<Station> p_pDestination;

    public:
        Station(std::string station);

        void vSetDestination(std::shared_ptr<Station> dest);
        std::string getName() const;
        std::shared_ptr<Station> getDestination() const;
};
```

Code

```
Station::Station(std::string station) : p_sName(station)
{

}
```

Code

```
std::string Station::getName() const
{
    return p_sName;
}
```

Code

```
std::shared_ptr<Station> Station::getDestination() const
{
    return p_pDestination;
}
```

Code

```
void Station::vSetDestination(std::shared_ptr<Station> dest)
{
    p_pDestination = dest;
}
```

Code

```
void Train::vGoTo(std::shared_ptr<Station> to)
{
    if(p_pIsAt==nullptr)
    {
        p_pIsAt = to;
        std::cout << "Zug wurde auf " << p_pIsAt->getName() << " gesetzt." << std::endl;
    }
    else if((p_pIsAt->getDestination())->getName() == to->getName())
    {
        p_pIsAt = to;
        std::cout << "Zug ist nach " << p_pIsAt->getName() << " gefahren." << std::endl;
    {
    else
    {
        std::cout << "Der Zug kann nicht nach " << to->getName() << " fahren: Es gibt keine Verbindung." << std::endl;
    }
}
```

Verständnisfragen

In diesem Teil wird es einige Aufgaben geben, die eher theoretischer Natur sind. Sie werden Codesnippets sehen, bei denen sich Fehler eingeschlichen haben, die Sie korrigieren sollen. Bitte führen Sie zuerst die `include`-Befehle aus, damit keine Zusatzfehler auftreten ;-).

Code

```
#include <iostream>
#include <memory>
```

Aufgabe 1

Im folgenden Abschnitt soll der Inhalt eines `weak_ptr` ausgegeben werden. Allerdings funktioniert die Ausgabefunktion nicht. Finden Sie die Fehler und testen Sie Ihre Lösung mit der unten angegebenen Test-Funktion.

Code

```
void printContent(std::weak_ptr<std::string> weak){
    std::cout << "Der Inhalt des weak-Pointers ist: " << weak << std::endl;
}
```

Code

```
{
    auto test_shared = std::make_shared<std::string>("Test");
    auto test_weak = test_shared;
    printContent(test_weak);
}
```

Aufgabe 2

Es soll der Inhalt eines `unique_ptr`-Arrays umgekehrt in einem anderen gespeichert werden. Allerdings funktioniert das noch nicht so ganz. Finden Sie die Fehler.

Code

```
std::unique_ptr<int[]> reverseArray(std::unique_ptr<int[]> unique_array){
    auto reversed_array = std::make_unique<int[]>(10);
    for(int i = 0; i < 10; i++){
        reversed_array[i] = 9 - unique_array[i];
    }

    return reversed_array;
}
```

Code

```
{
    auto unique_array = std::make_unique<int[]>(10);
    for(int i = 0; i < 10; i++){
        unique_array[i] = i;
    }

    std::cout << "Das ursprüngliche Array:" << std::endl;
    for(int i = 0; i < 10; i++){
        std::cout << "Stelle: " << i << " Inhalt: " << unique_array[i] << std::endl;
    }

    auto reversed_array = reverseArray(unique_array);

    std::cout << "Das umgedrehte neue Array:" << std::endl;

    for(int i = 0; i < 10; i++){
        std::cout << "Stelle: " << i << " Inhalt: " << reversed_array[i] << std::endl;
    }
}
```

Lösungen

Lösung 1

Der Fehler befindet sich in der Ausgabezeile. Auf den Inhalt eines Pointers kann nur mit entsprechender Referenzierung mit dem * -Operator zugegriffen werden. Bei `weak_ptr` muss zusätzlich die Funktion `lock()` genutzt werden, um den Inhalt abrufen zu können. Richtig ist also `std::cout << "Der Inhalt des weak-Pointers ist: " << *weak.lock() << std::endl;`.

Lösung 2

`unique_ptr` können nicht einfach kopiert werden, da es sonst mehr als eine Referenz auf das Objekt gibt. Deswegen **muss** der `move()`-Befehl benutzt werden! Richtig ist also, sowohl bei dem Aufruf von `reverseArray`, als auch beim `return`-Statement diesen Befehl zu verwenden: `auto reversed_Array = reverseArray(move(unique_array));` und `return move(reversed_array);`.

Container Klassen

- Überblick/Einführung
- std::vector
- std::list
- std::array
- std::map
- Iteratoren

Überblick/Einführung

Container sind Klassen zur Aufnahme von Objekten beliebigen Typs mit Funktionen 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. In den Container-Bibliotheken sind häufig gebrauchte Datenstrukturen als Template-Klassen implementiert, die beliebige Datentypen in sich aufnehmen können.

Man unterscheidet folgende grundsätzliche Typen:

- *Sequentielle Container*: Elemente behalten die Reihenfolge bei, in der sie eingefügt wurden. Zugriff erfolgt nur von einem Element zum nächsten bzw. vorigen oder über die genaue Position eines Elementes.
- *Assoziative Container*: Auf einzelne Elemente wird mit einem Schlüssel (Key) zugegriffen. Assoziative Container sind entweder sortiert oder unsortiert. Über sortierte Container kann zusätzlich zum Zugriff über den Schlüssel auch so iteriert werden, wie die Schlüssel geordnet sind. Unsortierte Container bieten einen schnelleren Zugriff.
- *Adapter*: Container mit einer eingeschränkten Schnittstelle für spezielle Anwendungen.

In Tabelle Tabelle 5.1 finden Sie eine Übersicht der zur Verfügung gestellten Container. Im Praktikum verwenden wir nur die Containertypen `vector`, `list`, `array` und `map` und `Iteratoren` um auf die Container zuzugreifen, auf die wir in den weiteren Abschnitten näher eingehen.

Die Benutzung der Container gestaltet sich durch die gemeinsamen Funktionen sehr einfach. Die wichtigsten Funktionen sind:

- `iterator begin()`: Liefert einen Iterator auf das erste Element.
- `iterator end()`: Liefert einen Iterator hinter das letzte Element. Daher darf der von end() zurück gegebene Iterator nicht dereferenziert werden.

  Wenn der Container leer ist, liefern `begin()` und `end()` denselben Iterator.
- `bool empty() const`: Prüft, ob Container leer ist.
- `void clear()`: Entfernt alle Elemente.

Der Zugriff auf die gespeicherten Daten erfolgt meist über Iteratoren. Der Iterator ist ein Zeiger innerhalb der Container und kann mit ++ und -- vor- und rückwärts bewegt werden. Die Funktion `begin()` liefert einen Iterator auf das erste Element der Liste. Das Ende der Liste kann durch die Funktion `end()` abgefragt werden. Mehr dazu im entsprechenden Abschnitt Iteratoren

Container	Beschreibung
Sequentielle Container	
array	Statische, lineare Speicherung Schnelle Iteration
vector	Lineare, benachbarte Speicherung schnelles Einfügen nur am Ende
deque	Lineare, benachbarte Speicherung schnelles Einfügen an Außenstellen
forward_list	Einseitig verkettete Liste schnelles Einfügen an beliebigen Stellen
list	Beidseitig verkettete Liste schnelles Einfügen an beliebigen Stellen
Assoziative 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
Ungeordnete assoziative Container	
unordered_multiset	Datenmenge, schneller assoziativer Zugriff Duplikate zulässig
unordered_set	Wie unordered_multiset, jedoch keine Duplikate zulässig
unordered_multimap	Menge von Schlüssel/Wert-Paaren schneller Zugriff über Schlüssel Duplikate für Schlüssel zulässig
unordered_map	Wie unordered_multimap, jedoch keine Duplikation für Schlüssel zulässig
Adapter	
stack	first-in-last-out (FILO) Datenstruktur (Stapel)
queue	first-in-first-out (FIFO) Datenstruktur (Warteschlange)
priority_queue	Queue, die Elemente mit Priorität speichert und so die Reihenfolge des Zugriffs bestimmt

Tabelle 5.1: Containerklassen in der Übersicht

std::vector

Die Größe eines `vector` kann sich während der Ausführung des Programmes bei Bedarf ändern. Dazu kann z.B. mit `push_back(Element)` ein Element hinten angehängt werden oder mit `pop_back()` das letzte Element abgerufen und aus dem Vektor entfernt werden. Definiert ist der Vektor im Header `<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, die später noch änderbar ist:

```
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);
```

Für den Zugriff auf Elemente stehen einige Methoden in Tabelle 5.2 zur Verfügung. Für eine vollständige Übersicht sei auch hier wieder auf die empfohlene Literatur verwiesen.

Funktionen	**Funktionalität**
size_type size()**const**	Liefert die Anzahl der Elemente
void swap(vector& `vec`)	Tauscht den Inhalt mit dem von `vec`
void push_back(**const** T& `val`)	Fügt `val` am Ende des Vektors ein
void insert(iterator `pos`, **const** T& `val`)	Fügt `val` an der Stelle `pos` innerhalb des Vektors ein
void pop_back()	Löscht das letzte Element im Vektor

Tabelle 5.2: Einige Funktionen von `vector` (Anmerkung: T steht in der Tabelle für beliebige Datentypen)

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

Ein ausführbares und veränderbares Beispiel:

```
#include <vector>
#include <iostream>
{
    std::vector<int> v1 = {0, 1, 2};
    int i;

    std::cout << "Der ursprüngliche Vektor: ";
    for(auto i: v1){
        std::cout << i << ' '; //Ausgabe: 0 1 2
    }
    std::cout << std::endl;

    v1.pop_back();

    std::cout << "Das letzte Element wurde nun gelöscht: ";
    for(auto i: v1){
        std::cout << i << ' '; //Ausgabe 0 1
    }
    std::cout << std::endl;

    v1.push_back(5);
    v1.insert(v1.begin(),6);

    std::cout << "Mit den beiden vorne und hinten angehängten Zahlen sieht unser Vektor jetzt so aus: ";
    for(auto i: v1){
        std::cout << i << ' '; //Ausgabe 6 0 1 5
    }
    std::cout << std::endl;
}
```

std::list

Eine Liste ist ein sequentieller Container, der für Einfügen und Löschen gut geeignet ist. Dafür ist die Iteration langsamer und eine Liste bietet keinen direkten Zugriff üer Index. Sie ist in `<list>` definiert.

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

```
Code
#include <list>
#include <string>

std::list<std::string> meineListe;
```

Eine Liste besitzt die gleichen 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(pos,Element)` und `erase(pos)` zur Verfügung (für eine kurze Übersicht einiger Funktionen, siehe Tabelle 5.3). Diese Funktionen existieren zwar auch für Vektoren, führen dort aber zur Verschiebungen des gesamten Vektors und machen alle Iteratoren ungültig, die sich auf Elemente hinter der Einfüge-/Löschposition beziehen. **Sie sind also dort nur in Sonderfällen einsetzbar, bei der Liste aber sinnvoll.**

Funktionen	**Funktionalität**
size_type size()**const**	Liefert die aktuelle Anzahl der Elemente
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
void pop_front()	Löscht das erste Element
void erase(iterator pos)	Löscht das Element an der Position pos
iterator insert(iterator post, **const** T& value)	Fügt ein Element an der Stelle pos ein

Tabelle 5.3: Einige Funktionen von `list`

Zum umgekehrten Durchlaufen der Liste kann man einen reverse-iterator mit den zugehörigen Funktionen `rbegin()` (letztes Element) und `rend()` (**vor** dem erstem Element) benutzen. Entsprechend gibt es auch die umgekehrten push-/pop-Funktionen: `push_front(Element)` 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 Elemente (in der richtigen Reihenfolge) übereinstimmen.

Beispiel:

Code

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

{
    std::list<int> l1;

    l1.push_back(7);

    std::cout << "Die ursprüngliche Liste: ";
    for(int i: l1){
        std::cout << i << ' '; //Ausgabe: 7
    }
    std::cout << std::endl;

    l1.insert(l1.begin(), 3);
    std::cout << "Nachdem die 3 vorne angehängt wurde, sieht die Liste so aus: ";
    for(int i: l1){
        std::cout << i << ' '; //Ausgabe: 3 7
    }
    std::cout << std::endl;

    l1.insert(l1.end(), 8);
    std::cout << "Nachdem die 8 hinten angehängt wurde, sieht die Liste so aus: ";
    for(int i: l1){
        std::cout << i << ' '; //Ausgabe: 3 7 8
    }
    std::cout << std::endl;

    l1.push_front(6);
    std::cout << "Es gibt verschiedene Möglichkeiten, Zahlen vorne anzuhängen: ";
    for(int i: l1){
        std::cout << i << ' '; //Ausgabe: 6 3 7 8
    }
    std::cout << std::endl;

    l1.pop_front();

    std::cout << "Nach dem Enfernen sieht die Liste nun wieder so aus: ";
    for(int i: l1){
        std::cout << i << ' '; //Ausgabe: 3 7 8
    }
    std::cout << std::endl;
}
```

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

Code

```
void insert(iterator pos, const_iterator first, const_iterator last)
```

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

Beispiel:

Code

```
#include <list>
#include <iostream>
{
    std::list<std::string> list1, list2;

    list1.push_back("gelb");
    list1.push_back("blau");
    list1.push_back("rot");

    std::cout << "Der Inhalt von list1 sieht so aus: " << std::endl;
    for(const auto& i: list1){
        std::cout << i << ' '; //Ausgabe: gelb blau rot
    }
    std::cout << std::endl;
    std::cout << std::endl;

    list2.push_front("weiß");
    list2.push_front("lila");
    list2.push_front("schwarz");

    std::cout << "Der Inhalt von list2 sieht so aus: " << std::endl;
    for(const auto& i: list2){
        std::cout << i << ' '; //Ausgabe: schwarz lila weiß
    }
    std::cout << std::endl;
    std::cout << std::endl;

    auto start = (list2.begin())++;
    list1.insert(list1.begin(), start, list2.end());

    std::cout << "Die zusammengefügte Liste sieht nun so aus: " << std::endl;
    for(const auto& i: list1){
        std::cout << i << ' '; //Ausgabe: schwarz lila weiß gelb blau rot
    }
    std::cout << std::endl;
    std::cout << std::endl;

    list1.erase(list1.begin());
    std::cout << "Der Inhalt von List1 sieht so aus: " << std::endl;
    for(const auto& i: list1){
        std::cout << i << ' ';
    }
    std::cout << std::endl;
}
```

std::array

Das in der Container-Bibliothek implentierte `array` ist eine bessere Implementierung der Ihnen eventuell aus C bekannten Felder. Dementsprechend ist `std::array` den C-Arrays vorzuziehen. Definiert ist `array` im Header `<array>` und ist wie folgt deklariert:

Code

```
template<class T, std::size_t N>
class array;
```

Dabei ist T der Typ der Elemente und N die Größe. Die Größe von `array` **muss** zur Compile-Zeit vorliegen. Deswegen können können Elemente weder hinzugefügt, noch entfernt werden.

Deklariert wird ein Array z.B. so:

Code

```
array<int, 42> feld; //ist ein Array namens feld, welches 42 int-Werte speichern kann.
```

Bei der Initialisierung können Werte gesetzt werden:

Code

```
array<int, 3> feld = {1, 2, 3}; //ist ein Array, das bereits mit Werten initialisiert wurde.
```

Für ein `array` ist sichergestellt, dass alle Elemente hintereinander im Speicher liegen. Dementsprechend kann auf die einzelnen Elemente mit dem []-Operator zugegriffen werden, beginnend für das erste Element mit 0 (bei dem vorherigen Beispiel würde feld[2] den Wert 3 zurückliefern). **Zugriffe auf ein Array außerhalb dieser Indizies ist ein Programmierfehler mit undefinierten Auswirkungen**. Die Funktion `at` empfängt die gleichen Argumente wie der []-Operator bei den klassischen C-Arrays, überprüft aber den Wertebereich und wirft im Fehlerfall eine Exception. Auf die Elemente kann auch über Iteratoren zugegriffen werden und mithilfe von `size_type size()const` kann die Anzahl der Elemente ausgeben werden. Die Funktion `void swap(array& ar)` tauscht den Inhalt des Arrays mit dem von `ar`.

Beispiel:

Code

```
#include <iostream>
#include <array>

{
    std::array<int, 4> arr = {0, 1, 2, 3};
    std::cout << "Inhalt von arr: ";
    for(auto it: arr){
        std::cout << it << " ";  //Ausgabe: 0 1 2 3
    }
    std::cout << std::endl;

    std::array<int, 4> swaparray = {0, 0, 0, 0};
    std::cout << "Inhalt von swaparray: ";
    for(auto it : swaparray){
        std::cout << it << " ";  //Ausgabe: 0 0 0 0
    }
    std::cout << std::endl;
    swaparray.swap(arr);

    std::cout << "Der Inhalt von swaparray ist nun: ";
    for(auto it: swaparray){
        std::cout << it << " ";  //Ausgabe: 0 1 2 3
    }
    std::cout << std::endl;

    std::cout << "Und der Inhalt von arr ist jetzt: ";
    for(auto it: arr){
        std::cout << it << " ";  //Ausgabe: 0 0 0 0
    }
    std::cout << std::endl;
}
```

std::map

Maps sind assoziative Container zur Speicherung von Schlüssel/Wert-Paaren. Sie ermöglichen schnellen Zugriff über den Schlüssel, können aber auch sequentiell durchlaufen werden. Zum schnelleren Zugriff kann man auch unsortierte Maps benutzen. In diesem Zusammenhang bedeutet unsortiert, dass die Schlüssel nicht sortiert abgespeichert werden. Zur einfacheren Darstellung benutzen wir im Praktikum sortierte Maps. Maps sind im Header `<map>` definiert.

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

Code

```
map<[Key], [Value]>;
```

also z.B.:

Code

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

Diese Map enthält int-Zahlen, auf die über strings als Schlüssel zugegriffen wird.

Code

```
map<long int, Studi> MapStudi;
```

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

Über den Subskript-Operator `operator[]` erfolgt der Zugriff mit dem Schlüssel. Wenn noch kein Objekt mit diesem Schlüssel existiert, wird ein leeres Elemtn über den Default-Kontrukstor erstellt (bei einem Pointer ist dies der `nullptr`) und zurückgegeben. Der Operator gibt eine Referenz auf das Objekt zurück. Auch für die Map gibt es die Alternative `at`, die, wenn kein Objekt gefunden wurde, eine Exception wirft. die Funktionen `begin()`, `end()`, `empty()` und `size()` sind analog zu den übrigen Containern auch für Maps definiert.

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

Eine sehr häufige Anwendung der Maps ist die Zuordnung zwischen externen Textschlüsseln auf interne Werte, wie im folgenden Beispiel die Zuordnung von Studierenden zu ihren Matrikelnummern: Beispiel:

Code

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

{
    std::map<long, std::string> studi_map;

    studi_map[123123] = "Schmachtenberg";
    studi_map[123456] = "Ruediger";
    studi_map[123321] = "Ulrich";

    std::cout << "Die Studierenden mit ihren Matrikelnummern sind: " << std::endl;
    for(const auto& it: studi_map){
        std::cout << it.first << " ";
        std::cout << it.second << std::endl;
    }
}
```

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

Code

```
Iterator find(Key);
```

die einen entsprechenden Iterator liefert. Falls dieses Element nicht gefunden wurde, wird ein Iterator auf `end()` zurückgegeben.

Beispiel:

Code

```
{
    std::map<std::string, int> numbers;

    numbers["eins"] = 1;
    numbers["zwei"] = 2;
    numbers["drei"] = 3;

    std::cout << "So sieht die map numbers aus: " << std::endl;
    for(const auto& it: numbers){
        std::cout << it.first << ": " << it.second << std::endl;
    }

    auto lookUp1 = numbers.find("eins");

    if(lookUp1 != numbers.end()){
        numbers.erase(lookUp1);
    }

    std::cout << std::endl;
    std::cout << "Nachdem wir die 1 gelöscht haben, sieht die map so aus: " << std::endl;
    for(const auto& it: numbers){
        std::cout << it.first << ": " << it.second << std::endl;
    }
}
```

Iteratoren

Iteratoren sind Objekte, die es gestatten, Elemente eines Containers (oder Streams) zu durchlaufen. Definiert sind sie im Header `<iterator>`. Ein Iterator hat zu jedem Zeitpunkt eine bestimmte Position innerhalb eines Containers, die er so lange beibehält, bis er durch einen neuen Befehlsaufruf versetzt wird. Die Standartbibliothek 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: kann sich vorwärts und rückwärts bewegen, ebenfall eine Kombination von Input- und Output-Iterator
- Random-Access: wie Bidirectional, kann jedoch auch springen

Eine mögliche Durchlaufreihenfolge einer Liste wird an folgendem Bild verdeutlicht:

Output

Bild 5.1: Durchlauf einer Liste

Die grundlegenden Operationen eines Iterators sind:

- die Dereferenzierung (* und -> Operatoren), die ein Element des Containers liefert
- die pre- und post-Inkrement-Operatoren (++ bzw. --)
- der Test auf (Un-)Gleichheit (== 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 +, +=, -, -=
- die Vergleichsoperatoren (<, <=, >, >=)

definiert.

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

Code

```
vector<int>::iterator it1;
```

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

Code

```
list<int>::const_iterator it2;
```

Die Konzepte Datentypableitung (`auto`) und Range-basierte `for`-Schleifen sind für Container-Zugriffe mittels Iteratoren ideal geeignet. In den folgenden Beispielen wird dies ersichtlich.

Beispiel (Berechnen einer Summe):

Code

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

{
    vector<int> v (3,1); //v: 1 1 1
    v.push_back(7); //v: 1 1 1 7
    int sum = 0;
    cout << "Summe(";

    for(auto it: v){ //Elementtyp des Vektors ist int, auto wird durch int ersetzt
        cout << " " << it;
        sum += it;  //Ausgabe: 1 1 1 7
    }
    cout << " ) = " << sum << endl; //Ausgabe: 10
}
```

Soll der Container rückwärts durchlaufen werden, gibt es verschiedene Möglichkeiten. Die einfachste ist wohl der `reverse_iterator`. Er ist ebenfalls Teil der `<iterator>`-Bibliothek und kann beispielsweise so deklariert werden:

Code

```
vector<int>::reverse_iterator revIt;
```

`reverse_iterator` erzeugt einen Iterator, der vom Ende eines Containers zu dem Anfang durch Inkrementierung wandert. Man erhält ihn auch als Ergebnis einer entsprechenden Funktion (`rbegin()`oder `rend()`).

Beispiel:

Code

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

{
    std::vector<int> v;
    for(int i = 0; i < 10; ++i){
        v.push_back(i);
    }

    std::vector<int>::iterator it;
    std::vector<int>::reverse_iterator revIt;

    std::cout << "v wird vorwärts durchlaufen: ";
    for(it = v.begin(); it < v.end(); it++){
        std::cout << *it << " ";  //Ausgabe: 0 1 2 3 4 5 6 7 8 9
    }
    std::cout << std::endl;

    std::cout << "v wird rückwärts durchlaufen: ";
    for(revIt = v.rbegin(); revIt != v.rend(); revIt++){
        std::cout << *revIt << " ";  //Ausgabe: 9 8 7 6 5 4 3 2 1 0
    }
    std::cout << std::endl;
}
```

Übungsaufgaben zu diesem Kapitel finden Sie **hier**.

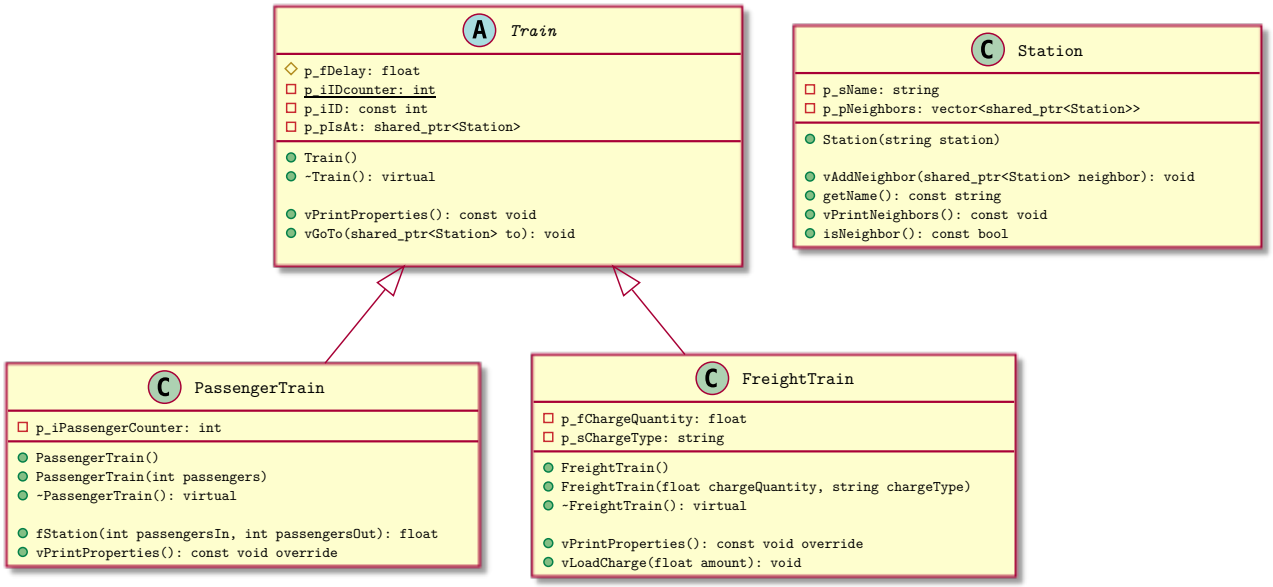
Übungsaufgaben zu Container

In diesem Kapitel haben Sie gelernt, was Container-Klassen sind und wie sie angewendet werden können. Im Folgenden gibt es einige praktische Übungen und Verständnisfragen, mit denen Sie überprüfen können, ob Sie das Gelernte korrekt anwenden können.

- Praktische Aufgabe
- Verständnisfragen

Praktische Aufgaben

In diesem Teil soll das Gelernte praktisch anwendet und somit das `Train`-Beispiel erweitert werden. Konkret werden Sie die Klasse `Station` aus den vorangegangen Aufgaben bearbeiten, sodass verschiedene Bahnhöfe miteinander verbunden werden.



Teil 1

Legen Sie in `Station` einen Vector `p_pNeighbors` vom Typ shared_ptr an, in dem benachbarte Stationen gespeichert werden.

Teil 2

Jetzt sollen verschiedene Funktionen implementiert werden. Schreiben Sie eine Funktion `vAddNeighbor`, die den Returntyp `void` hat, einen Nachbar zu dem aktuellen Bahnhof übergeben bekommt und ihn in dem Vector `p_pNeighbors` speichert. Um zu kontrollieren, dass Ihre Implementierung richtig ist, legen Sie ebenfalls eine Funktion `vPrintNeighbors` an, die den Vector ausgibt.

Teil 3

Zum Schluss legen Sie eine Funktion `isNeighbor` an, die überprüft, ob ein anderer Bahnhof mit dem aktuellen verbunden ist. Die Funktion soll 1 zurückgeben, wenn die Stationen miteinander verbunden sind und 0, wenn sie nicht verbunden sind. Passen Sie die zuvor implementierte `vGoTo(...)` so an, dass hier die neue Funktion `isNeighbor` zu Überprüfung der Verbindung verwendet wird.

Teil 4

Fügen Sie Ihre Implementierungen von `Train`, `PassengerTrain` und `FreightTrain` aus den vorherigen Aufgaben an passender Stelle ein. Die Funktion `vSetStation` ist für diese Aufgabe nicht relevant, kann also weggelassen werden. Compilieren Sie anschließend alles und führen Sie die `main()`-Funktion aus, um zu überprüfen, ob alles korrekt gemacht wurde.

zum Lösungsvorschlag

Code
```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
```

Code
```
//Station.h
class Station{
    //hier Ihre Erweiterungen einfügen
    private:
    std::string p_sName;

    public:
    Station(std::string station);
    std::string getName() const;
}
```

Code
```
//Station.cpp
//Konstruktor für Station
Station::Station(std::string station)
    : p_sName(station){

    }
```

Code
```
//Getter-Funktion für p_sName
std::string Station::getName() const {
    return p_sName;
}
```

Code
```
//hier: Funktion Station::vAddNeighbor
```

Code
```
//hier: Funktion Station::vPrintNeighbors
```

Code
```
//hier: Funktion Station::isNeighbor
```

Code
```
//hier: Train.h einfügen
```

Code
```
//hier: Train.cpp
//Destruktor einfügen
```

Code
```
//hier: Funktion Train::vGoto einfügen
```

Code
```
//hier: Funktion Train::vPrintProperties einfügen
```

Code
```
//hier: PassengerTrain.h einfügen
```

Code
```
//hier: PassengerTrain.cpp
//Konstruktor einfügen
```

Code
```
//hier: PassengerTrain::vPrintProperties einfügen
```

Code
```
//hier: Funktion PassengerTrain::fStation einfügen
```

Code
```
//hier: FreightTrain.h einfügen
```

Code
```
//hier: FreightTrain.cpp
//Konstruktor einfügen
```

Code
```
//hier: Funktion FreightTrain::vPrintProperties einfügen
```

Code

```
//hier: Funktion FreightTrain::vLoadCharge einfügen
```

Code

```
#define OPERATOR operator<< //Diese Zeile kann in jeder IDE weggelassen werden, sie ist nur für Jupyter notwendig

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR //Diese Zeile kann in jeder IDE weggelassen werden, sie ist nur für Jupyter notwendig
```

Code

```
// Test

std::unique_ptr<PassengerTrain> aTrain;
aTrain =  std::make_unique<PassengerTrain>();

auto bTrain = std::make_unique<FreightTrain>(2, "tons of wood");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
std::cout << *aTrain;

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
std::cout << *bTrain;

auto berlin = std::make_shared<Station>("Berlin");
auto hamburg = std::make_shared<Station>("Hamburg");
auto frankfurt = std::make_shared<Station>("Frankfurt");
auto koeln = std::make_shared<Station>("Koeln");
auto muenchen = std::make_shared<Station>("Muenchen");

berlin->vAddNeighbor(hamburg);
berlin->vAddNeighbor(frankfurt);

hamburg->vAddNeighbor(berlin);
hamburg->vAddNeighbor(koeln);

// Strecke nur in eine Richtung
koeln->vAddNeighbor(frankfurt);

frankfurt->vAddNeighbor(koeln);
frankfurt->vAddNeighbor(berlin);
frankfurt->vAddNeighbor(muenchen);

muenchen->vAddNeighbor(frankfurt);

// Zeige Liste der bekannten Nachbarn
berlin->vPrintNeighbors();
hamburg->vPrintNeighbors();
koeln->vPrintNeighbors();
frankfurt->vPrintNeighbors();
muenchen->vPrintNeighbors();




std::cout<< "\n\n\nZuege fahren lassen: \n\n";

aTrain->vGoTo(berlin);
aTrain->vGoTo(hamburg);
aTrain->vGoTo(koeln);
aTrain->vGoTo(hamburg); // Fehlermeldung: keine Verbindung
```

Lösungen

zur Aufgabenstellung

Hier finden Sie die Lösungen zu den Aufgaben zu `Station`. Bitte beachten Sie, dass es mehrere Möglichkeiten gibt, die Aufgaben zu lösen, hier wird lediglich eine Variante aufgezeigt. Solange Ihre Implementierung funktioniert wie gefordert, ist sie wahrscheinlich richtig!

Code

```
class Station
{
    private:
        std::string p_sName;
        std::vector<std::shared_ptr<Station>> p_pNeighbors;

    public:
        Station(std::string station);

        void vAddNeighbor(std::shared_ptr<Station> neighbor);
        std::string getName() const;
        void vPrintNeighbors() const;
        bool isNeighbor(std::shared_ptr<Station> request) const;
}
```

Code

```
Station::Station(std::string station) : p_sName(station)
{}
```

Code

```
bool Station::isNeighbor(std::shared_ptr<Station> request) const
{
    if(std::find(p_pNeighbors.begin(), p_pNeighbors.end(), request) != p_pNeighbors.end()) {
        return 1;
    }

    return 0;
}
```

Code

```
void Station::vPrintNeighbors() const
{
    std::cout << std::endl << "Folgende Bahnhoefe koennen von " << p_sName << " aus angefahren werden: ";
    for (auto n : p_pNeighbors) {
        std::cout << n->getName() << ", ";
    }
}
```

Code

```
std::string Station::getName() const
{
    return p_sName;
}
```

Code

```
void Station::vAddNeighbor(std::shared_ptr<Station> neighbor)
{
    p_pNeighbors.push_back(neighbor);
}
```

Code

```
void Train::vGoTo(std::shared_ptr<Station> to)
{
    if((p_pIsAt==nullptr) || (p_pIsAt->isNeighbor(to)))
    {
        p_pIsAt = to;
        std::cout << "Zug ist nach " << p_pIsAt->getName() << " gefahren." << std::endl;
    }
    else
    {
        std::cout << "Der Zug kann nicht nach " << to->getName() << " fahren: Es gibt keine Verbindung." << std::endl;
    }
}
```

Verständnisfragen

Dieser Teil ist eher theoretischer Natur. Er besteht aus 3 kleinen Aufgaben, bei denen Sie auf Fehlersuche in kleinen Codesnippets gehen sollen. Bitte führen Sie die unten aufgeführten `include`-Befehle als erstes aus, damit es nicht zu Zusatzfehlern kommt ;-).

Code

```
#include <iostream>
#include <vector>
#include <map>
#include <array>
#include <string>
#include <iterator>
```

Aufgabe 1

Folgender vector wurde angelegt. Er soll die Zahlen 7, 7, 7, 256, 4 in dieser Reihenfolge beinhalten. Finden Sie die Fehler.

Code

```
{
    vector<int> vec(3,7);
    vec.push_back(256);
    vec.insert(4,4);
    for(auto it: vec){
        cout << it << " ";
    }
}
```

Aufgabe 2

Es wurde ein Array angelegt mit 4 Elementen. An der vierten Stelle soll der Wert verändert werden. Warum funktioniert das nicht? Finden Sie den Fehler.

Code

```
{
    std::array<float, 4> arr = {1.1, 2.2, 3.3};
    arr.at(4) = 4.4;
    for(auto i: arr){
        std::cout << i << " ";
    }
}
```

Aufgabe 3

In der Folgenden Map soll das gefundene Element ausgegeben werden. Wenn die gesuchte Person (hier: Ernst Schmachtenberg) nicht in dieser Map gespeichert ist, soll eine Meldung erscheinen. Finden Sie die Fehler.
Zusatz: Warum ist eine Map nicht geeignet, um Vor- und Nachnamen zueinander zu speichern?

Code

```
{
    std::map<std::string, std::string> people;
    people["Schmachtenberg"] = "Ernst";
    people["Rüdiger"] = "Ulrich";
    people["Lankes"] = "Stefan";
    people["Sauer"] = "Dirk";
    people["Monti"] = "Antonello";

    auto wanted = people.find("Ernst");

    if(wanted != people.end()){
        std::cout << wanted->first << ", " << wanted->second;
    } else {
        std::cout << "Person not found.";
    }
}
```

Output

```
Person not found.
```

Lösungen

Lösung 1

Die erste 4 in dem insert-Befehl ist kein Iterator, dementsprechend wird nicht die passende Sstelle zurück geliefert. Mögliche Lösungen wären, entweder `vec.insert(vec.end(), 4)` oder `vec.push_back(4)` zu benutzen. Alternativ kann man den ganzen vector auch von vornherein mit den korrekten Werten initialisieren. Zusätzlich fehlt das 'std::' vor `vector<int>` bei der Initialisierung und vor `cout`. Diesen Fehler könnte man auch durch die Zeile `using namespace std` vor den geschweiften Klammern beheben, dies ist aber keine gängige Praxis (mehr).

Lösung 2

Hier sollten Sie sich an den Satz "Array index starts at 0" erinnern (Dieser Satz ist i.A. gültig, es gibt aber einige wenige Ausnahmen, wie Sie vielleicht aus Matlab etc. wissen). Um auf die vierte Stelle des Arrays zuzugreifen, wird der Index 3 benutzt. Korrekt würde der Befehl also lauten `arr.at(3) = 4.4`.

Lösung 3

Wenn `find()` den Iterator `end()` liefert, heißt das, dass kein Eintrag mit diesem Schlüssel in der Map existiert. Der erste Fehler ist also, dass es in der if-Bedingung `!=` statt `==` heißen müsste (oder die beiden Anweisungen getauscht werden müssten). Der zweite Fehler ist, dass Key (deutsch: Schlüssel) und Value (deutsch: Wert) in der Funktion verwechselt wurden. Es wurde also nach dem Wert gesucht und nicht nach dem Schlüssel. Richtig ist also `auto wanted = people.find("Schmachtenberg");`.

Exception Handling (Ausnahmebehandlung)

Unter Exceptions (Ausnahmen) versteht man Fehler oder Sondersituationen, die bei der Programmausführung in einer Bibliothek auftreten können und direkt an das aufrufende Programm gemeldet werden sollen. Beispiele sind falsche Eingaben, arithmetische Fehler (z.B. Divisionen durch Null) und Bereichsüberschreitungen bei Vektoren.

Beim Entwurf von Bibliotheken und Klassen ist anfangs nicht klar, wie und vor allem an welcher Stelle Fehler geeignet zu behandeln sind. Insbesondere soll der Anwender der Bibliothek entscheiden, wie bei einem Fehler verfahren wird: Falls es dazu kommt, dass der Arbeitsspeicher nicht für eine Speicheranforderung ausreicht, ist es für viele Anwendungen akzeptabel, sie geordnet zu beenden. Andere müssen auch in diesem Fall weiter ausgeführt werden. Anstatt den Fehler durch alle Funktionen explizit durchzureichen und dafür alle Definitionen zu ändern, wird bei Exceptions der Fehler beim Auftreten geworfen (throw) und in der Anwendung durch den zugehörigen Exception Handler gefangen (catch).

Die Syntax für den throw-Ausdruck lautet:

Code
```
throw expression ;
```

Der zu überwachende Code steht in einem try-Block, der von mindestens einem Exception- Handler überwacht wird. Es können beliebig viele catch-Blöcke zu einem try-Block angegeben werden. Sie werden der Reihenfolge nach berücksichtigt.

Code
```
try {
    // Dieser Teil des Codes wird überwacht // hier wird eine Exception geworfen
}
catch (const Exceptionclass1& aExc) {
    // hier werden Ausnahmen der Exceptionclass1
    // und aller Unterklassen bearbeitet
}
catch (const Exceptionclass2& aExc) {
    // hier werden Ausnahmen der Exceptionclass2
    // und aller Unterklassen bearbeitet
}
catch (...) {
    // hier werden alle übrigen Ausnahmen behandelt
}
```

Wird eine Exception im Programm nicht gefangen, erfolgt eine Systemfehlermeldung mit Programmabbruch. Daher gibt es den (...)-catch-Block, der alle Exceptions fängt. Nach diesem Block können keine anderen Exceptions mehr gefangen werden. Dieser allgemeine Exception-Handler sollte nur sehr vorsichtig benutzt werden. Man beachte, dass auch Unterklassen gefangen werden:

Code
```
catch(Base&) {
    // fängt auch Derived -Objekte
}
catch (Derived&) {
    // hat keinen Effekt
}
```

Bei einer Hierarchie von Exceptions sollte die Ausnahme mittels einer polymorphen Funktion bearbeitet werden:

Code
```
catch(Base& e) {
    // fängt alle Base-Objekte und ruft die richtige Bearbeitungsfunktion auf
    e.vTreat();
}
```

Die Exception ist dabei ein Objekt, das beim Werfen erstellt wird, und dem Handler weitergegeben wird. Für diesen Zweck werden eigene Klassen verwendet. Sie sollten von std::exception oder anderen Exception-Typen aus der Standardbibliothek erben:

Code
```
#include <exception>
#include <stdexcept>
using namespace std;

class MyException : public logic_error {
    // ...
};
```

Die meisten von std::exception geerbten Klassen der Standardbibliothek fordern einen Beschreibungsstring beim Konstruieren (allerdings nicht exception selbst), der an den Konstruktor der Standardexception weitergegeben wird. Weitere Datenelemente für eigene Informationen können natürlich hinzugefügt werden.

Code
```
class MyException : public logic_error {
    int p_i;
public:
    MyException (const string& what, int i) : logic_error(what), p_i(i) {}
};
```

Zur Verdeutlichung diene zunächst folgendes Beispiel: Bei der Array-Klasse aus dem vorigem Kapitel soll at(int) eine Referenz auf das Element an einem Index zurückgeben. Falls der Index aber negativ oder außerhalb der Länge des Arrays ist, ist das nicht möglich und eine Exception vom Typ std::out_of_range wird geworfen:

Code
```
template <typename T, int N> T& Array<T, N>::at(int i)
{
    if (0 <= i && i < N)
        return p_array[i];
    else
        throw std::out_of_range("Fehler: Array Index außerhalb des gültigen Bereichs.");
}
```

Im Fehlerfall wird der Code nach dem throw-Statement nicht mehr ausgeführt und es wird direkt zu einem passenden Handler gesprungen. Um diese Fehlersituation zu berücksichtigen, muss der Aufruf der Funktion at() in einem try- Block stehen, der einen Exception-Handler für out_of_range enthält:

Code
```
try {
    // Dieser Block wird vom Exception-Handler überwacht.
    /* ... */
    x = array.at(3);
    // Ausführung springt zum Handler
    /* ... */
    }
catch (const out_of_range& e) { // Handler
    cout << e.what() << endl;
}
```

Beim Wurf der Exception wird mit dem Code des entsprechenden Handlers fortgesetzt. Der entsprechende try-Block wird nur bis zum Aufruf ausgeführt. Direkt danach wird also die Beschreibung auf cout ausgegeben. Das geworfene Objekt wird an die Variable des catch-Blocks (hier e) übergeben und kann so bearbeitet werden.

Da die Exception-Objekte der Standardbibliothek sehr allgemein sind, sollten bei größeren Projekten eigene Exception-Klassen eingeführt werden, ggf. in einer Hierarchie. Dadurch können Fehler genauer abgebildet werden und es werden Kollision mit Ausnahmen anderer Programmteile vermieden.

Exceptions sollen aber nicht nur ausgebbar sein. Oft erfordern Fehler eine Behandlung. Daher empfiehlt sich eine (polymorphe) Bearbeitungsfunktion (z.B. vTreat()), die dann im catch-Block aufgerufen werden kann. Die Exception-Klassen müssen dann Daten- elemente für die Daten haben, die die Ausnahme beschreiben. Diese werden beim throw durch Aufruf entsprechender Konstruktoren gefüllt.

Eine Hierarchie für mathematische Fehler könnte dann so aussehen:

Code

```
class MathError : std::exception {
    // Fehlerbeschreibung
    string what;

public:
    // Konstruktor mit Fehlerbeschreibung
    MathError(const string& s) : what(s){}

    virtual void vTreat () const {
        // Ausgabe der Beschreibung
        cout << what << endl;
    }
};

class NegRoot: public MathError {
    // Fehlerhaftes (negatives) Argument
    double p_arg;

public:
    // Konstruktor mit Argument
    NegRoot(double arg)
    // Beschreibung konstant
    : MathError("Fehler: Wurzel aus negativer Zahl"), p_arg(arg){}

    // Fehlerbehandlung: Ausgabe der Beschreibung mit → Argument
    void behandeln () const override {
        MathError::vTreat();
        cout << " Argument: " << p_arg << endl;
    }
};
```

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

Code

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

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

Code

```
try {
    /* ... */
}
catch (NegRoot& e) {
    // behandle NegRoot
}
catch (MathError& e) {
    // behandle alle anderen MathError
}
```

Im folgenden Beispiel soll, wenn ein Fehler auftritt auch der Name der geöffneten Datei ausgegeben werden:

Code

```
void f(string file) {
    ifstream fin(file);
    try {
        // ...
    }
    catch (const exception& e) {
        cout << "Fehler: " << e.what() << "bei " << file << endl;
    }
}
```

Allerdings kann in dieser Funktion nicht alles notwendige ausgeführt werden (z.B. Abbruch) und der Fehler soll an das Hauptprogramm weitergegeben werden. Im obigen Beispiel geschieht das nicht. In diesen Situationen kann die aktuelle Exception im catch-Block für den aufrufenden Code noch einmal geworfen werden. Dies geschieht durch ein throw ohne Argumente:

Code

```
catch (const exception& e) {
    cout << "Fehler: " << e.what() << "bei " << file << endl;
    // aktuelle Exception noch einmal werfen
    throw;
}
```

Exceptions sollten nicht für jede Situation verwendet werden. Bei der Methode find eines Containers ist es möglich, dass ein gesuchtes Element nicht existiert. Es ist sinnvoll, dass im Rückgabetyp abzubilden, zum Beispiel mit einem End-Iterator. Bei anderen Funktionen ist ein Scheitern nicht normal, zum Beispiel bei Index-operationen. Diese geben normalerweise Referenzen zurück. Wenn allerdings der Index außerhalb des validen

Bereichs ist, wird eine Exception geworfen. Insbesondere wenn ein Konstruktor ein Objekt nicht konstruieren kann, sind Exceptions der einzige Weg, die Konstruktion abzubrechen.

Als Referenz sind hier die wichtiges Arten von Exceptions aus der Exception-Hierarchie in der Standardbibliothek angegeben:

- logic_error
  - invalid_argument – domain_error
  - length_error
  - out_of_range
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - system_error
    - ios_base::failure
- bad_cast
- bad_alloc

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:

```
<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 Funktions- deklaration 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.

Übungsaufgaben zu diesem Kapitel finden Sie **hier**.

Übungsaufgaben zu Exception

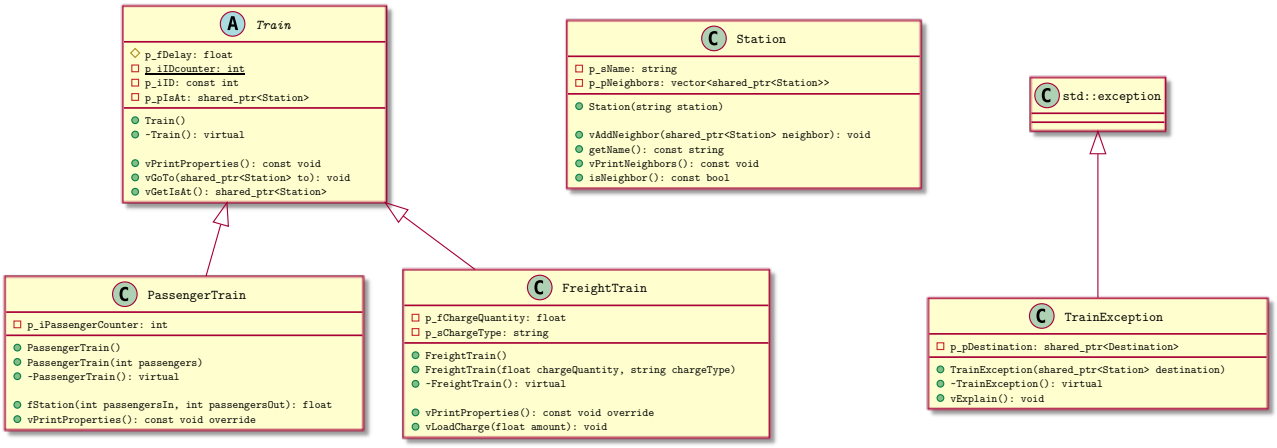
- Praktische Aufgabe
- Verständnisfragen

Praktische Aufgabe

Im Notebook zu Exceptions wurde das Konzept von Exceptions in C++ erläutert: Diese dienen dazu, bei Ausnahmesituationen den normalen Programmablauf zu unterbrechen und optional eine Fehlerbehandlung zu starten.

Dafür muss zunächst definiert werden, welche Art von Fehlermeldung geworfen werden soll. Hierfür ist es möglich, von `std::exception` erbende Klassen zu erstellen, es existieren aber auch schon einige standardmäßig in C++. Daraufhin wird im Abschnitt, bei dem ein Fehler auftreten könnte, mittels `throw` eine Instanz jener Exception erzeugt und an die aufrufende Funktion übergeben. Bei vielen bereits vorhandenen Klassen wie beispielsweise den elementaren Datentypen sind diese Maßnahmen schon implementiert, sodass nur noch auf die korrekte Behandlung dieser eventuell auftretenden Exceptions geachtet werden muss.

Dafür wird, falls vorhanden, der den Funktionsaufruf umschließende `catch`-Block mit dem passenden Typ einer Exception aufgerufen. Wenn der Fehler behebbar ist, sollten entsprechende Schritte eingeleitet werden, andernfalls ist oft eine Ausgabe der Fehlermeldung mit `what` hilfreich. Existiert kein passendes `catch`, bricht die Ausführung des Programms ebenfalls mit der Ausgabe einer Fehlermeldung ab.



Unten finden Sie die Codebasis auf dem Stand der letzten Übung zu Containerklassen, die nun am Ende erweitert wird. Es soll eine Exception geworfen werden, falls zwischen gewünschtem Start und Ziel keine Verbindung besteht. In diesem Fall wird aktuell in `Train::vGoTo(...)` eine einfache Fehlermeldung über die Konsole ausgegeben. Diese Aufgabe soll im Folgenden eine Exception übernehmen:

Erstellen einer `TrainException` Klasse

Um das oben beschriebene Verhalten zu erreichen, soll die Klasse `TrainException` erstellt werden, welche von `std::exception` erbt. Zunächst müssen dafür die beiden Libraries `<exception>` und `<stdlib.h>` inkludiert werden. Außerdem verfügt diese Klasse als Membervariable über einen Shared Pointer auf den Bahnhof, der angefahren werden soll, um im weiteren Verlauf dessen Namen auszugeben.

Ansonsten wenden Sie dieselben Konzepte an, die Sie bisher beim Erstellen von Klassen und Vererbungsstrukturen erlernt haben; insbesondere also Konstruktor, Destruktor und passende Zugriffsspezifizierer.

Ausgabe der Fehlermeldung

Zur Realisierung der Ausgabe implementieren Sie die Methode `TrainException::vExplain()`, die auf der Konsole ausgibt, welchen Bahnhof der Zug anfahren wollte und dass das nicht möglich ist, da keine Verbindung existiert

Integration der Exception in den Programmablauf

Werfen Sie nun anstelle der einfachen Fehlermeldung im `else`-Zweig von `Train::vGoTo(...)` eine `TrainException`, der der Shared Pointer im Konstruktor übergeben wird. Führen Sie nun die Testaufrufe innnerhalb eines `try-catch` Blockes durch, der im Fehlerfall `TrainException::vExplain()` der geworfenen Exception aufruft.

zum Lösungsvorschlag

Code

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
```

Code

```
class Station
{
    private:
        std::string p_sName;
        std::vector<std::shared_ptr<Station>> p_pNeighbors;

    public:
        Station(std::string station);

        void vAddNeighbor(std::shared_ptr<Station> neighbor);
        std::string getName() const;
        void vPrintNeighbors() const;
        bool isNeighbor(std::shared_ptr<Station> request) const;
};
```

Code

```
Station::Station(std::string station) : p_sName(station)
{}
```

Code

```
// falls request ein bekannter Nachbar ist: returnt 1, sonst 0
bool Station::isNeighbor(std::shared_ptr<Station> request) const
{
    // falls request in p_pNeighbors:
    if(std::find(p_pNeighbors.begin(), p_pNeighbors.end(), request) != p_pNeighbors.end()) {
        return 1;
    }

    // sonst:
    return 0;
}
```

Code

```
void Station::vPrintNeighbors() const
{
    std::cout << std::endl << "Folgende Bahnhoefe koennen von " << p_sName << " aus angefahren werden: ";
    for (auto n : p_pNeighbors) {
        std::cout << n->getName() << ", ";
    }
}
```

Code

```
std::string Station::getName() const
{
    return p_sName;
}
```

Code

```
void Station::vAddNeighbor(std::shared_ptr<Station> neighbor)
{
    p_pNeighbors.push_back(neighbor);
}
```

Code

```
class Train
{
    private:
        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;

        std::shared_ptr<Station> p_pIsAt = nullptr;
    protected:
        float p_fDelay = 0;

    public:
        Train() = default;
        virtual ~Train() = 0;

        virtual void vPrintProperties(std::ostream& ausgabe) const;

        void vGoTo(std::shared_ptr<Station> to);
};
```

Code

```
Train::~Train()
{
    std::cout << "Aufruf des rein virtuellen Destruktors" << std::endl;
}
```

Code

```
void Train::vPrintProperties(std::ostream& ausgabe) const
{
    ausgabe << "ID: " << p_iID << std::endl;

    //ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl; -> jetzt in PassengerTrain
    ausgabe << "momentane Verspaetung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
class PassengerTrain : public Train
{
    public:
        PassengerTrain() = default;
        PassengerTrain(int p_iPassengerCounter);
        virtual ~PassengerTrain() = default;

        void vPrintProperties(std::ostream& ausgabe) const override;
        float fStation(int passengersIn, int passengersOut);

    private:
        int p_iPassengerCounter = 0;

};
```

Code

```
PassengerTrain::PassengerTrain(int passengers) : Train(), p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
void PassengerTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/


float PassengerTrain::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Sekunden, Umrechnung in Minuten spaeter
    float secondsChange = 120 - passengersIn * 10 - passengersOut * 5;
    p_fDelay = (p_fDelay - secondsChange < 0) ? 0 : (p_fDelay - secondsChange)/60;

    return secondsChange/60;
}
```

Code

```
class FreightTrain : public Train
{
    private:
        float p_fChargeQuantity = 0.0;
        std::string p_sChargeType = "default";

    public:
        FreightTrain() = default;
        FreightTrain(float chargeQuantity, std::string chargeType);

        //void vPrintProperties() const;
        void vPrintProperties(std::ostream& ausgabe) const override;

        void vLoadCharge(float amount);
};
```

Code

```
FreightTrain::FreightTrain(float chargeQuantity, std::string chargeType) : Train(), p_fChargeQuantity(chargeQuantity),
↪  p_sChargeType(chargeType)
{}
```

Code

```
void FreightTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Ladung: " << p_fChargeQuantity << " " << p_sChargeType << std::endl;
}
```

Code

```
// abladen: amount < 0
void FreightTrain::vLoadCharge(float amount)
{
    if (p_fChargeQuantity + amount < 0)
    {
        std::cout << "Es kann nicht mehr abgeladen werden als geladen ist. Abbruch." << std::endl;
        return;
    }

    p_fChargeQuantity += amount;
    std::cout << "Es wurden " << amount << " " << p_sChargeType;


    if (amount > 0)
    {
        std::cout << " aufgeladen.";
    }
    else
    {
        std::cout << " abgeladen.";
    }

    std::cout << " Es befinden sich nun " << p_fChargeQuantity << " " << p_sChargeType << " auf dem Zug." << std::endl;
}
```

Code

```
#define OPERATOR operator<<

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR
```

Code

```
//TrainException.h
//hier: Klasse TrainException implementieren
```

Code

```
//TrainException.cpp
//hier: Konstruktor implementieren
```

Code

```
//TrainException.cpp
//hier: Destruktor implementieren
```

Code

```
//TrainException.cpp
//hier: vExplain(...) implementieren
```

Code

```
void Train::vGoTo(std::shared_ptr<Station> to)
{
    if((p_pIsAt==nullptr) || (p_pIsAt->isNeighbor(to)))
    {
        p_pIsAt = to;
        std::cout << "Zug ist nach " << p_pIsAt->getName() << " gefahren." << std::endl;
    }
    else
    {
        std::cout << "Der Zug kann nicht nach " << to->getName() << " fahren: Es gibt keine Verbindung." << std::endl;
    }

    /*
    // Fahre nur, wenn Zug im Anfangszustand, oder das Ziel und die aktuelle Haltestelle verbunden sind:
    if((p_sIsAt==nullptr) || (p_sIsAt->getDestination() == to))
    {
        p_sIsAt = to;
        std::cout << "Zug ist nach " << p_sIsAt->getName() << " gefahren." << std::endl;
    }
    else
    {
        std::cout << "Der Zug kann nicht nach " << to->getName() << " fahren: Es gibt keine Verbindung." << std::endl;
    }
    */
}
```

Code

```
// Test

std::unique_ptr<PassengerTrain> aTrain;
aTrain =  std::make_unique<PassengerTrain>();

auto bTrain = std::make_unique<FreightTrain>(2, "tons of wood");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
std::cout << *aTrain;

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
std::cout << *bTrain;

// alter Test: gleiches Verhalten wie in 04_smartpointer
/*
auto berlin = std::make_shared<Station>("Berlin");
auto hamburg = std::make_shared<Station>("Hamburg");
auto frankfurt = std::make_shared<Station>("Frankfurt");

berlin->vAddNeighbor(hamburg);
hamburg->vAddNeighbor(berlin);

aTrain->vGoTo(berlin);
aTrain->vGoTo(hamburg);
aTrain->vGoTo(frankfurt);
*/

// neu: Strecken"netz"
auto berlin = std::make_shared<Station>("Berlin");
auto hamburg = std::make_shared<Station>("Hamburg");
auto frankfurt = std::make_shared<Station>("Frankfurt");
auto koeln = std::make_shared<Station>("Koeln");
auto muenchen = std::make_shared<Station>("Muenchen");

berlin->vAddNeighbor(hamburg);
berlin->vAddNeighbor(frankfurt);

hamburg->vAddNeighbor(berlin);
hamburg->vAddNeighbor(koeln);

// Strecke nur in eine Richtung
koeln->vAddNeighbor(frankfurt);

frankfurt->vAddNeighbor(koeln);
frankfurt->vAddNeighbor(berlin);
frankfurt->vAddNeighbor(muenchen);

muenchen->vAddNeighbor(frankfurt);

// Zeige Liste der bekannten Nachbarn
berlin->vPrintNeighbors();
hamburg->vPrintNeighbors();
koeln->vPrintNeighbors();
frankfurt->vPrintNeighbors();
muenchen->vPrintNeighbors();




std::cout<< "\n\n\nZuege fahren lassen: \n\n";

aTrain->vGoTo(berlin);
aTrain->vGoTo(hamburg);
aTrain->vGoTo(koeln);
aTrain->vGoTo(hamburg); // Fehlermeldung: keine Verbindung
```

Output

```
Eigenschaften 'aTrain':
ID: 0
momentane Verspaetung: 0 Minuten
Anzahl Passagiere: 0

Eigenschaften 'bTrain':
ID: 1
momentane Verspaetung: 0 Minuten
Ladung: 2 tons of wood

Folgende Bahnhoefe koennen von Berlin aus angefahren werden: Hamburg, Frankfurt,
Folgende Bahnhoefe koennen von Hamburg aus angefahren werden: Berlin, Koeln,
Folgende Bahnhoefe koennen von Koeln aus angefahren werden: Frankfurt,
Folgende Bahnhoefe koennen von Frankfurt aus angefahren werden: Koeln, Berlin, Muenchen,
Folgende Bahnhoefe koennen von Muenchen aus angefahren werden: Frankfurt,


Zuege fahren lassen:

Zug ist nach Berlin gefahren.
Zug ist nach Hamburg gefahren.
Zug ist nach Koeln gefahren.
Der Zug kann nicht nach Hamburg fahren: Es gibt keine Verbindung.
```

Lösungsvorschlag zur Implementierung

zur Aufgabenstellung

Code

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
#include <exception>
#include <stdlib.h>
```

Code

```
class Station
{
    private:
        std::string p_sName;
        std::vector<std::shared_ptr<Station>> p_pNeighbors;

    public:
        Station(std::string station);
        virtual ~Station() = default;
        void vAddNeighbor(std::shared_ptr<Station> neighbor);
        std::string getName() const;
        void vPrintNeighbors() const;
        bool isNeighbor(std::shared_ptr<Station> request) const;
};
```

Code

```
Station::Station(std::string station) : p_sName(station)
{}
```

Code

```
// falls request ein bekannter Nachbar ist: returnt 1, sonst 0
bool Station::isNeighbor(std::shared_ptr<Station> request) const
{
    // falls request in p_pNeighbors:
    if(std::find(p_pNeighbors.begin(), p_pNeighbors.end(), request) != p_pNeighbors.end()) {
        return 1;
    }

    // sonst:
    return 0;
}
```

Code

```
void Station::vPrintNeighbors() const
{
    std::cout << std::endl << "Folgende Bahnhoefe koennen von " << p_sName << " aus angefahren werden: ";
    for (auto n : p_pNeighbors) {
        std::cout << n->getName() << ", ";
    }
}
```

Code

```
std::string Station::getName() const
{
    return p_sName;
}
```

Code

```
void Station::vAddNeighbor(std::shared_ptr<Station> neighbor)
{
    p_pNeighbors.push_back(neighbor);
}
```

Code

```
class Train
{
    private:
        static inline int p_iIDCounter = 0;
        const int p_iID = p_iIDCounter++;

        std::shared_ptr<Station> p_pIsAt = nullptr;
    protected:
        float p_fDelay = 0;

    public:
        Train() = default;
        virtual ~Train() = 0;

        virtual void vPrintProperties(std::ostream& ausgabe) const;
        void vGoTo(std::shared_ptr<Station> to);
        std::shared_ptr<Station> getIsAt();
};
```

Code

```
std::shared_ptr<Station> Train::getIsAt()
{
    return p_pIsAt;
}
```

Code

```
Train::~Train()
{
    std::cout << "Aufruf des rein virtuellen Destruktors" << std::endl;
}
```

Code

```
void Train::vPrintProperties(std::ostream& ausgabe) const
{
    ausgabe << "ID: " << p_iID << std::endl;

    //ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl; -> jetzt in PassengerTrain
    ausgabe << "momentane Verspaetung: " << p_fDelay << " Minuten"<< std::endl;
}
```

Code

```
class PassengerTrain : public Train
{
    public:
        PassengerTrain() = default;
        PassengerTrain(int p_iPassengerCounter);
        virtual ~PassengerTrain() = default;

        void vPrintProperties(std::ostream& ausgabe) const override;
        float fStation(int passengersIn, int passengersOut);

    private:
        int p_iPassengerCounter = 0;

};
```

Code

```
PassengerTrain::PassengerTrain(int passengers) : Train(), p_iPassengerCounter(passengers)
{
    std::cout << "Aufruf des Nicht-Standardkonstruktors" << std::endl;
}
```

Code

```
void PassengerTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Anzahl Passagiere: " << p_iPassengerCounter << std::endl;
}
```

Code

```
// gibt Änderung der Verspätung zurück und Verändert `delay` (aber delay >= 0)

/* Annahmen / Funktion
bei Haltestelle: 2 Minuten Zeit.
    Pro einsteigender Passagier: 10 Sekunden
    Pro aussteigender Passagier: 5 Sekunden
    nicht parallel
alles >2 Minuten: neue Verspätung
*/


float PassengerTrain::fStation(int passengersIn, int passengersOut)
{
    // mehr Passagiere als Aussteigende? -> fehlerhafte Einhabe -> mache nichts
    if (p_iPassengerCounter < passengersOut)
    {
        std::cout << "Es koennen nicht mehr Personen aussteigen als sich im Zug befinden" << std::endl;
        return 0;
    }

    p_iPassengerCounter = p_iPassengerCounter + passengersIn - passengersOut;


    // Rechnung in Sekunden, Umrechnung in Minuten spaeter
    float secondsChange = 120 - passengersIn * 10 - passengersOut * 5;
    p_fDelay = (p_fDelay - secondsChange < 0) ? 0 : (p_fDelay - secondsChange)/60;

    return secondsChange/60;
}
```

Code

```
class FreightTrain : public Train
{
    private:
        float p_fChargeQuantity = 0.0;
        std::string p_sChargeType = "default";

    public:
        FreightTrain() = default;
        FreightTrain(float chargeQuantity, std::string chargeType);

        //void vPrintProperties() const;
        void vPrintProperties(std::ostream& ausgabe) const override;

        void vLoadCharge(float amount);
};
```

Code

```
FreightTrain::FreightTrain(float chargeQuantity, std::string chargeType) : Train(), p_fChargeQuantity(chargeQuantity),
↪  p_sChargeType(chargeType)
{}
```

Code

```
void FreightTrain::vPrintProperties(std::ostream& ausgabe) const
{
    Train::vPrintProperties(ausgabe);
    ausgabe << "Ladung: " << p_fChargeQuantity << " " << p_sChargeType << std::endl;
}
```

Code

```
// abladen: amount < 0
void FreightTrain::vLoadCharge(float amount)
{
    if (p_fChargeQuantity + amount < 0)
    {
        std::cout << "Es kann nicht mehr abgeladen werden als geladen ist. Abbruch." << std::endl;
        return;
    }

    p_fChargeQuantity += amount;
    std::cout << "Es wurden " << amount << " " << p_sChargeType;


    if (amount > 0)
    {
        std::cout << " aufgeladen.";
    }
    else
    {
        std::cout << " abgeladen.";
    }

    std::cout << " Es befinden sich nun " << p_fChargeQuantity << " " << p_sChargeType << " auf dem Zug." << std::endl;
}
```

Code

```
#define OPERATOR operator<<

std::ostream & OPERATOR(std::ostream& out, const Train& train)
{
    train.vPrintProperties(out);
    return out;
}

#undef OPERATOR
```

Code

```
class TrainException : std::exception
{
    private:
        std::shared_ptr<Station> p_pDestination;

    public:
        TrainException(std::shared_ptr<Station> destination);
        virtual ~TrainException();
        void vExplain(); // ganz unten
};
```

Code

```
TrainException::TrainException(std::shared_ptr<Station> destination) : p_pDestination(destination)
{}
```

Code

```
TrainException::~TrainException()
{}
```

Code

```
void TrainException::vExplain()
{
    std::cout << "Der Zug versuchte nach " << p_pDestination->getName() << " zu fahren. Nicht moeglich, da keine Verbindung
↪   vorhanden. Ende." << std::endl;
}
```

Code

```
void Train::vGoTo(std::shared_ptr<Station> to)
{
    if((p_pIsAt==nullptr) || (p_pIsAt->isNeighbor(to)))
    {
        p_pIsAt = to;
        std::cout << "Zug ist nach " << p_pIsAt->getName() << " gefahren." << std::endl;
    }
    else
    {
        throw TrainException(to);
        std::cout << "Der Zug kann nicht nach " << to->getName() << " fahren: Es gibt keine Verbindung." << std::endl;
    }
}
```

Code

```
// Test

std::shared_ptr<PassengerTrain> aTrain;
aTrain =  std::make_shared<PassengerTrain>();

auto bTrain = std::make_shared<FreightTrain>(2, "tons of wood");

std::cout << std::endl << "Eigenschaften 'aTrain':" << std::endl;
std::cout << *aTrain;

std::cout << std::endl << "Eigenschaften 'bTrain':" << std::endl;
std::cout << *bTrain;

// neu: Strecken"netz"
auto berlin = std::make_shared<Station>("Berlin");
auto hamburg = std::make_shared<Station>("Hamburg");
auto frankfurt = std::make_shared<Station>("Frankfurt");
auto koeln = std::make_shared<Station>("Koeln");
auto muenchen = std::make_shared<Station>("Muenchen");


berlin->vAddNeighbor(hamburg);
berlin->vAddNeighbor(frankfurt);

hamburg->vAddNeighbor(berlin);
hamburg->vAddNeighbor(koeln);

// Strecke nur in eine Richtung
koeln->vAddNeighbor(frankfurt);

frankfurt->vAddNeighbor(koeln);
frankfurt->vAddNeighbor(berlin);
frankfurt->vAddNeighbor(muenchen);

muenchen->vAddNeighbor(frankfurt);

// Zeige Liste der bekannten Nachbarn
berlin->vPrintNeighbors();
hamburg->vPrintNeighbors();
koeln->vPrintNeighbors();
frankfurt->vPrintNeighbors();
muenchen->vPrintNeighbors();
```

Output

```
Eigenschaften 'aTrain':
ID: 0
momentane Verspaetung: 0 Minuten
Anzahl Passagiere: 0

Eigenschaften 'bTrain':
ID: 1
momentane Verspaetung: 0 Minuten
Ladung: 2 tons of wood

Folgende Bahnhoefe koennen von Berlin aus angefahren werden: Hamburg, Frankfurt,
Folgende Bahnhoefe koennen von Hamburg aus angefahren werden: Berlin, Koeln,
Folgende Bahnhoefe koennen von Koeln aus angefahren werden: Frankfurt,
Folgende Bahnhoefe koennen von Frankfurt aus angefahren werden: Koeln, Berlin, Muenchen,
Folgende Bahnhoefe koennen von Muenchen aus angefahren werden: Frankfurt,
```

Code

```
try
{
    std::cout<< "\n\n\nZuege fahren lassen: \n\n";

    aTrain->vGoTo(berlin);
    aTrain->vGoTo(hamburg);
    aTrain->vGoTo(koeln);
    aTrain->vGoTo(hamburg); // Fehlermeldung: keine Verbindung
}

catch(TrainException &e)
{
    e.vExplain();
}
```

Output

```


Zuege fahren lassen:

Zug ist nach Berlin gefahren.
Zug ist nach Hamburg gefahren.
Zug ist nach Koeln gefahren.
```

Output

```
Error:
```

Verständnisfragen

Unterschiedliche Ausgabe

Warum wird mit der Implementierung von `Train::vGoTo()` im Lösungsvorschlag bei einem fehlerhaften Aufruf der Inhalt der Exception ausgegeben, nicht aber die normale Ausgabe in der Zeile darunter (die in der vorherigen Implementierung ausgeführt wurde)?

Abbruch

Welches Verhalten erwarten Sie von den unten aufgeführten Methodenaufrufen? Wie könnte erreicht werden, dass auch bei einzelnen fehlerhaften Aufrufen von `Train::vGoTo()` die darauf folgenden Anweisungen ausgeführt werden?

Code

```
try
{
    std::cout<< "\n\n\nZuege fahren lassen: \n\n";

    aTrain->vGoTo(berlin);
    aTrain->vGoTo(hamburg);
    aTrain->vGoTo(frankfurt);
    aTrain->vGoTo(koeln);
    aTrain->vGoTo(frankfurt);
    aTrain->vGoTo(hamburg);
    aTrain->vGoTo(berlin);
}

catch(TrainException &e)
{
    e.vExplain();
}
```

Lösungsvorschläge zu den Verständnisfragen

Unterschiedliche Ausgaben

Nach dem Werfen einer Exception mittels `throw` wird die fehlerhafte Funktion verlassen, bis die Exception durch ein passendes `catch` abgefangen wird. Bei einander aufrufenden Funktionen geschieht das so lange, wie die Exception unbehandelt bleibt, d.h. es können durch eine Exception auch mehrere Funktionen verlassen werden. Wenn schließlich die eigentliche `main`-Methode des Programms verlassen werden soll, bricht die Ausführung des Programms mit der Ausgabe dieser Exception ab.

Abbruch

Da keine direkte Verbindung von Hamburg nach Frankfurt besteht, würde die Ausführung bei der aktuellen Implementierung an dieser Stelle abbrechen, da eine Exception geworfen und der `catch`-Block ausgeführt wird. Um dieses Verhalten zu verhindern, könnte jeder Aufruf von `Train::vGoTo()` in einem separaten `try-catch` Block stattfinden, was allerdings schnell unübersichtlich wird:

Code

```
std::cout<< "\n\n\nZuege fahren lassen: \n\n";

try{aTrain->vGoTo(berlin); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(hamburg); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(frankfurt); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(koeln); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(frankfurt); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(hamburg); }
catch(TrainException &e) { e.vExplain();}

try{aTrain->vGoTo(berlin); }
catch(TrainException &e) { e.vExplain();}
```

Templates

- Funktionentemplates
- Klassentemplates

Oft unterscheiden sich Lösungen verschiedener 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. Die Implementierung dieser ist häufig nur vom zu speichernden Datentyp abhängig und kann daher gleich implementiert werden. 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++ kann man mit einem *Template* die Funktionalität einmal definieren und dann die Datentypen einfach in die Implementierung einfügen.

Ein Template ist eine Variable, 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. Erst beim Compilieren wird für jeden benutzten Datentyp der entsprechende Code generiert. Dafür muss die gesamte Template-Definition sichtbar sein. Also werden Templates ausschließlich in Headerdateien definiert.

Template-Deklarationen haben die folgende Syntax:

Code
```
template<{parameter-list}> {declaration}
```

Es wird entweder ein Funktions- oder ein Klassentemplate deklariert. Ein einzelner Parameter kann ein Datentyp (`typename T` oder `class T`) oder ein Integer-Wert (`int N`) sein. Mehrere Parameter werden durch Kommata getrennt:

Code
```
template<typename Key, typename Value> ...;
template<typename T, int Size> ...;
template<unsigned N, char M> ...;
```

Parameter für Templates müssen zur Compile-Zeit bekannt sein. Daher führt folgendes Beispiel zu einem Fehler:

Code
```
int n = 100;
// std::array<double, n> ds;  // Fehler: n nicht zur Compile-Zeit bekannt
```

Werte, die zur Compile-Zeit feststehen, können mit `const` markiert werden:

Code
```
const int n = 100;
std::array<double, n> ds;  // Ok. n ist constant expression
```

Funktionentemplates

Funktionentemplates werden immer dann sinnvoll eingesetzt, wenn Algorithmen unabhängig vom Datentyp auf bestimmte Eigenschaften dieser Datentypen zurückgeführt werden können.

Ein einfaches Beispiel ist eine Maximumsfunktion:

Code
```
template<typename T>
T& max(T& left, T& right) {
    return  right < left ? left : right;
}
```

Wie man sieht, entspricht diese Definition einer normalen Funktionsdefinition. Nur die `template`-Zeile wurde hinzugefügt und anstatt eines konkreten Datentyps (z.B. `double`) wurde der Parameter `T` verwendet. Damit

dieses Template für einen Datentyp verwendet werden kann, muss der Operator `<` zwischen zwei Objekten vom Typ `T` definiert sein.

Ein Funktions-Template kann grundsätzlich für alle Variablentypen genutzt werden. Einzige Voraussetzung hierfür ist, dass für diesen Datentyp alle Operatoren definiert sind, die in der Funktion verwendet werden. Der Compiler erzeugt zur Übersetzungszeit den benötigten Code für alle Funktionsaufrufe. In diesem Fall ist die Funktion also mehrmals vorhanden.

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 bereits vorgefertigten Templates in der Standardbibliothek (s.).

Im folgenden Beispiel wird ein Klassen-Template `Array` definiert. Es ist generisch über dem Datentyp `T` und der Array-Größe `N`. Die Parameter können in der gesamten Template-Definition als Datentyp bzw. Wert benutzt werden.

Code
```
template<typename T, int N>
class Array {
    private:
        T p_array[N];
    public:
        T* begin() {
            return p_array;
        }
        T* end() {
            return p_array + N;
        }
        T& at(int i);
};
```

Der Name einer Klasse, die aus solch einem Template generiert wird, ist der Name des Klassen-Templates gefolgt von den konkreten Parametern in spitzen Klammern. Template-Instanziierungen mit unterschiedlichen Parametern sind unterschiedliche Typen.

Code
```
Array<int, 15> a;
Array<int, 100> b;
// Typen von a und b sind verschieden:
// a == b ist nicht definiert

// mit weiteren Typen
Array<float, 1> f;

struct S {};
Array<S, 20> f;
```

Überall, wo in der Definition `T` und `N` verwendet werden, ersetzt der Compiler sie durch den konkreten Typ bzw. Wert. Das heißt, für ein Integer-Array der Länge 15 wird folgender Code generiert:

Code
```
template<>
class Array<int, 15> {
    private:
        int p_array[15];
    public:
        int* begin() {
            return p_array;
        }
        int* end() {
            return p_array + 15;
        }
        int& at(int i);
};
```

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

Code
```
template<typename T, int N>
T* Array<T, N>::end() {
    return p_array + N;
}
```

Ein- und Ausgabe

- Ausgabe
- Formatierte Ausgabe
- Eingabe
- Formatierte Eingabe
- Ein- und Ausgabe mit Dateien
  - Schreiben von Daten in eine Datei
  - Lesen von Daten aus einer Datei
  - Abfangen von Fehlern
  - Datei-Flags

- Strings
  - String-Streams

Für die Ein- und Ausgabe sollten *nicht* die `stdio`-Funktionen wie `putchar(), printf()` etc. genutzt werden sondern die im folgenden beschriebene Bibliothek.

Für Ein- und Ausgabe bietet C++ die sogenannten Streams an. 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 für 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

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

zur Verfügung. Um die IO-Streams zu benutzen, müssen Sie die Bibliothek über

Code
```
#include <iostream>
```

einbinden (ohne Endung `.h`).

Ausgabe

Eine Ausgabe erfolgt durch die Anweisung:

Code
```
std::cout << object;
```

Mehrere Ausgaben können auch direkt miteinander verbunden werden:

Code
```
std::cout << object << object << ... << object;
```

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

Beispiel:

Code

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

Code

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

std::cout << "Ausgabe im Terminal:\n";

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

`std::endl` steht für neue Zeile, stattdessen kann auch `\n` innerhalb eines Strings verwendet werden. Bei Verwendung von `\n` werden Ausgaben erst angezeigt, wenn das Programm endet oder `std::flush` aufgerufen wird.

Formatierte Ausgabe

Die Stream-Klassen bieten folgende Methoden zur Manipulation der Ausgabe:

Methode	Verwendung
`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

Beachte: Das Setzen der Feldbreite hat nur Einfluss auf das nächste Ausgabeelement. Ohne Angabe eines Parameters liefern `width(), fill(), flags()` und `precision()` jeweils den aktuellen Wert.

Für die Ausgabeformatierung stehen folgende Flags zu Verfügung:

Flag	Verwendung
`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: `1.2345e67`
`ios::fixed`	Darstellung: `1234.56`
`ios::unitbuf`	Ausgabe nach jeder Operation
`ios::stdio`	Ausgabe nach jedem Zeichen

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

Flag	Verwendung
`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;
std::cout.width(6);
std::cout.precision(3);
std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout << x << std::endl;
std::cout.width(10);
std::cout.precision(5);
std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout << x << std::endl;
std::cout.width(10);
std::cout << i << std::endl;
std::cout.width(6);
std::cout << i << std::endl;
```

Um das Ganze etwas zu vereinfachen, existieren die sogenannten IO-Manipulatoren, die das Setzen der Ausgabeparameter direkt als Ausgabeelemente über `<<` ermöglichen. Sie sind in `iomanip` definiert. Um sie zu verwenden, muss

```
#include <iomanip>
```

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

Flag	Verwendung
`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 IO-Methoden. Am Beispiel ist nochmals die obige Ausgabe realisiert:

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

Die Parameter sind voneinander unabhängig . 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.

Eingabe

Möchte man den Wert einer Variablen 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` beim ersten Zeichen das keine Ziffer ist, bei `string` bei einem 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 werden und dann konvertiert werden. Mit einem `static_cast` kann der Wert ohne Überprüfung konvertiert werden. Besser ist es, mit einem `switch` die Eingabe zu prüfen:

Code
```
enum class Farbe { Rot = 1, Blau = 2 };
std::cout << "Farbe? (1=Rot, 2=Blau, sonst=Rot)" << std::endl;
int i;
std::cin >> i;
Farbe f;
// oder ungeprüft: f = static_cast<Farbe>(i);
switch (i) {
    case int(Farbe::Blau):
        f = Farbe::Blau;
        break;
    case int(Farbe::Rot):
    default:
        f = Farbe::Rot;
        break;
}
std::cout << "Farbe: " << int(f) << std::endl;
```

Formatierte Eingabe

Analog zur Ausgabe, lässt sich auch die Eingabe durch Flags beeinflussen.

Flag	Verwendung
`ios::skipws`	führende Leerzeichen beim Einlesen ignorieren (default)
`ios::noskipws`	führende Leerzeichen beim Einlesen nicht ignorieren

Ein- und Ausgabe mit Dateien

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

Mit sequentiellen Dateien kann auf drei Arten interagiert werden: Daten einlesen, in eine neue Datei schreiben oder an eine existierende Datei anhängen. Bei einem sequentiellen File können die Daten nicht umgeordnet werden, ohne das ganze File zu löschen.

Die Standardbibliothek zum Lesen und Schreiben von Dateien kann mit

Code
```
#include <fstream>
```

eingebunden werden.

Schreiben von Daten in eine Datei

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

Code
```
// Testvariablen
std::string name = "Arthur";
int semester = 42;
double schnitt = 4.2;
const std::string& dateiname = "Ausgabedatei.dat";

// Öffnen des Files für die Ausgabe
std::ofstream outfile(dateiname);

// Schreibe die Daten in das File
outfile << "Name: " << name << std::endl;
outfile << "Semester: " << semester << std::endl;
outfile << "Notenschnitt: " << std::setprecision(1)
        << std::setiosflags(std::ios::fixed) << schnitt << std::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. 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 beim Aufruf des Konstruktors automatisch geöffnet wird und der Destruktor das File wieder schließt, wenn der Gültigkeitsbereich des Objektes verlassen wird.

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

Output

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

Lesen von Daten aus einer Datei

Das folgende Programm liest die Daten aus der Datei `Ausgabedatei.dat` wieder ein und gibt sie auf dem Bildschirm aus:

Code

```
        std::string name;
        int semester;
        double schnitt;
        std::string info[3];
        const std::string& dateiname = "Ausgabedatei.dat";

        // Öffnen des Files zum Lesen
        std::ifstream infile(dateiname);

        // Lese die Daten aus dem File
        infile >> info[0] >> name;
        // info[0] = "Name:"; name = "Arthur";

        infile >> info[1] >> semester;
        // info[1] = "Semester:"; semester = 42;

        infile >> info[2] >> schnitt;
        // info[2] = "Notenschnitt:"; schnitt = 4.2;

        // Schreiben auf den Bildschirm
        std::cout << info[0] << name << std::endl;
        std::cout << info[1] << semester << std::endl;
        std::cout << info[2] << schnitt << std::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. 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:

Output

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

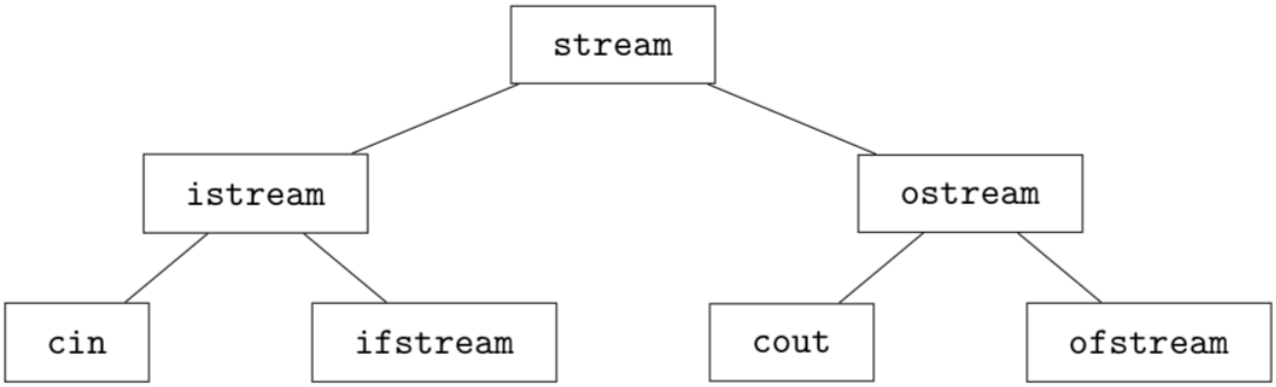
Ist die Menge der einzulesenden Daten nicht von vornherein bekannt, so muss es eine Möglichkeit geben zu erkennen, wann das Dateiende erreicht ist. Dies kann mittels der Funktion `eof()` getestet werden.

Code

```
// Variable zum Einlesen
int zahl;
// Öffnen der Datei
std::ifstream fin("Testdatei.dat"); // Lies , bis Dateiende erreicht fin >> zahl;
fin >> zahl; // Einlesen vor erstem eof Test
while (!fin.eof())
{
    std::cout << "Gelesene Zahl : " << zahl << std::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 hier noch einmal die Klassenstruktur der IO dargestellt.



Abfangen von Fehlern

Beim Lesen von Files und Schreiben in Files können Fehler auftreten, die abgefangen werden müssen. Beispielsweise könnte man versuchen, aus einem nichtexistierenden File zu lesen, das [usb]{acronym-label=”usb” acronym-form=”singular+short”}-Laufwerk könnte nicht verfügbar sein oder man könnte für die Datei nicht genügend Rechte haben. Ein `ifstream`- oder `ofstream`-Objekt besitzt vier Funktionen, über die der Zustand abgefragt werden kann: `good()` (Operation erfolgreich), `eof()` (Ende der Datei erreicht), `fail()` (Fehler z.B. Formatfehler) und `bad()` (Gegenteil von `good()`). Dieser Zustand bezieht sich immer auf die letzte ausgeführte Leseoperation.

Fehler bei Streams können auf zwei Weisen behandelt werden. Die eine Möglichkeit ist, nach jeder Operation abzufragen, ob der Stream noch `good()` ist. Falls ein Fehlerzustand vorliegt, kann der Fehler durch eine entsprechende Aktion bearbeitet werden oder eine Exception geworfen werden.

Die Stream-Klassen unterstützen das Werfen der Exception auch direkt: Mit der `exceptions`-Methode wird aktiviert, dass der Stream Exceptions wirft, wenn eine Operation fehlschlägt. Diese Methode nimmt eine Bitmaske entgegen und wird wie folgt aufgerufen:

Code
```
ifstream f("dateiname");
f.exceptions(ios_base::eofbit | ios_base::failbit | ios_base::badbit);
```

`ios_base` ist dabei die Basisklasse aller Eingabe- und Ausgabe-Streams. Die Zustände, die zu Exceptions führen sollen, werden mit einem bitweisen Oder (`|`) verknüpft. Bei der Ausführung werden dann bei einem Fehler Ausnahmen vom Typ `ios::failure` -- einer Unterklasse von `std::exception` -- geworfen. Falls schon ein Fehler existiert, wenn diese Methode aufgerufen wird, wird sofort eine Ausnahme geworfen.

Code
```
int liesFile(std::string dateiname) {
    // Öffne die Datei zum Lesen
    std::ifstream fin(dateiname);
    fin.exceptions(std::ios::eofbit | std::ios::failbit | std::ios::badbit);
    // Exceptions definieren.

    // Lies den Wert ein
    int i;
    fin >> i;
    // Exception geworfen, falls Einlesen fehlschlug
    return i;
}
```

Der Unterschied zwischen den Zuständen *fail* und *bad* ist klein, aber durchaus von Interesse: *fail* 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, kommt der Stream in diesen Zustand. *bad* wird gesetzt, wenn der Stream prinzipiell nicht mehr in Ordnung ist oder Zeichen verlorengegangen sind. Dann ist der Stream nicht mehr benutzbar. Dieses Flag wird z.B. beim Positionieren vor einen Dateianfang gesetzt.

Datei-Flags

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

Flag	Verwendung
`ios_base::in`	Lesen (Default bei `ifstream`)

Flag	Verwendung
`ios_base::out`	Schreiben (Default bei `ofstream`)
`ios_base::app`	Anhängen
`ios_base::ate`	ans Ende Positionieren (”`at end`”)
`ios_base::trunc`	alten Dateiinhalt löschen
`ios_base::nocreate`	Datei muss existieren
`ios_base::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:

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

Strings

Mit einem String der Standardbibliothek von wird die Bearbeitung von Zeichenketten gegenüber C-Strings (`char[]`) wesentlich erleichtert. Voraussetzung für die Benutzung ist die Einbindung der Bibliothek `<string>` durch

Code
```
#include <string>
```

Es können dann Elemente der 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 ist natürlich eine Initialisierung mit Leerzeichen.
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 entsprechenden `<<` bzw. `>>`-Operatoren.

Code
```
std::string s1(10, 'A');
std::string s2("Dies ist ein String.");
std::string s3;

std::cout << s1 << std::endl;
std::cout << s2 << std::endl;
std::cout << "Geben Sie einen String ein:" << std::endl;
std::cin >> s3;
std::cout << "Der String " << s3  << " wurde eingegeben." << std::endl;
```

Auch die Vergleichsoperatoren, sowie der `+` - und der `+=` -Operator sind intuitiv für Strings definiert. Die Vergleichsoperatoren liefern das Ergebnis des zeichenweisen Vergleichs entsprechend der zugrundeliegenden Codierung. Da die Codierung von Buchstaben alphabetisch ist, sind auch die Vergleichsoperatoren für Strings alphabetisch. Die `+` -Operatoren dienen zur Aneinanderreihung (Konkatenation) zweier Strings.

Beispiel:

Code
```
    std::string s1("ABC");
    std::string s2;
    std::string s3;

    s2 = "AC";
    s3 = s1 + s2;
    std::cout << s3 << std::endl;
    std::cout << "Maximum von: " << s1 << " und " << s2 << ": ";
    if (s1 > s2)
        std::cout << s1;
    else
        std::cout << s2;
```

Der Zugriff auf die einzelnen Zeichen eines Strings erfolgt mit dem `[]`-Operator. Auch hier wird mit dem Index 0 auf das erste Zeichen zugegriffen. Im Gegensatz zu den C-Strings wird `string` nicht durch ein Null-Byte (`\0`) beendet. Das Ende wird durch die aktuelle Länge des Strings, die mitgespeichert wird, bestimmt. Die Länge kann durch die Methode `length()` abgefragt werden.

Beispiel:

Code

```
std::string s1("ABC");

// [] fängt bei 0 an
std::cout << "Zweites Zeichen: " << s1[1] << std::endl;
// length() gibt die Anzahl der Zeichen an
std::cout << "Länge: " << s1.length() << std::endl;
```

Beachte: `s1[s1.length() - 1]` ist das letzte Zeichen des Strings, `s1[s1.length()]` erzeugt einen Zugriffsfehler.

Neben diesem direkten Zugriff auf die einzelnen Elemente kann auch sequentiell auf die Zeichen eines Strings zugegriffen werden.

Beispiel:

Code

```
std::string s1("ABCD");

for (auto c : s1)
    std::cout << c << ' ';
```

Für die komfortable Bearbeitung von Stringobjekten stehen diverse Methoden zur Verfügung. Die in 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 Online-Hilfe).

Methode	Funktionalität
`length(), size()`	Liefert die aktuelle Länge des Strings
`empty()`	Liefert `true`, falls leer
`insert(pos, str)`	Fügt `str`vor `pos` im Strin 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`, oder -1 falls nicht vorhanden
`rfind(str, pos)`	Liefert die Position des letzten Auftretens von `str` im String ab `pos`, oder -1 falls nicht vorhanden
`find_first_of(str, pos)`	Liefert die Position des ersten Auftretens eines Zeichens von `str` im String ab `pos`, oder -1 falls nicht vorhanden
`find_last_of(str, pos)`	Liefert die Position des letzten Auftretens eines Zeichens von `str` im String ab `pos`, oder -1 falls nicht vorhanden
`find_first_not_of(str, pos)`	Liefert die erste Position des Zeichens im String, das nicht in `str` vorkommt, oder -1 falls alle vorkommen
`find_first_not_of(str, pos)`	Liefert die letzte Position des Zeichens im String, das nicht in `str` vorkommt, oder -1 falls alle 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 Vergleichsstrings `(pos1, anz1)` eingeschränkt werden.

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()`

String-Streams

Mit den String-Stream-Klassen kann man unter Verwendung der Formatierungsmöglichkeiten der IO-Streams 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 gelesen 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:

Code

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

string namen[3];
for (int i = 0; i < 3; i++) {
    // Beachte: stringstream lokal, keine Datei
    stringstream s;
    // Ausgabe "Objekt" + Nummer, z.B. "Objekt1"
    s << "Objekt" << i + 1;
    namen[i] = s.str();
}

// Kontrollausgabe
for (const auto& name : namen){
    cout << name << endl;
}
```

Typumwandlung

- Implizite (automatische) Typumwandlung
- Explizite (erzwungene) 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 Datentyp gebracht werden. Dazu werden die Werte, wenn möglich, automatisch so umgewandelt, dass keine Informationen verloren gehen. Es besteht aber für den Programmierer durch Angabe des Typs die Möglichkeit, eine Typumwandlung zu erzwingen. Um dies zu erreichen sollte in C++ die funktionale Typumwandlung genutzt werden. Dazu wird vor dem, in Klammern stehenden, Objekt der gewünschte Typ geschrieben:

Code

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

d = i + f;            // i wird zu float, i+f dann zu double.
i = int(f) + int(d);  // f, d werden zu int. 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.

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.

Code

```
class Bruch {
    int p_x = 0, p_y = 1;  // Zähler und Nenner
    public:
        Bruch() = default;
        Bruch(int x)  // implizite Konvertierung
            : p_x(x)
        {}
        Bruch(int x, int y)
            : p_x(x), p_y(y)
        {}
        Bruch operator*(const Bruch& r)
        {
            Bruch res;
            res.p_x = p_x * r.p_x;
            res.p_y = p_y * r.p_y;
            return res;
        }
};

int main()
{
    Bruch x(5, 2);
    Bruch y = x * 15;  // implizit: 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.

Code

```
class Bruch {
        // ...
    public:
        explicit Bruch(int x)  // jetzt explizit
            : p_x(x)
        {}
        operator double() const
        {
            return double(p_x) / double(p_y);
        }
};

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.

Explizite (erzwungene) Typumwandlung

Ähnlich wie bei der automatischen Typumwandlung gibt es auch bei der expliziten Typumwandlung mehr als eine Art. Zum einen die Funktionale Notation, zum anderen die Cast-Notation.

Funktionale Notation

Die Typumwandlung wird durch Aufruf eines entsprechenden Konstruktors erreicht. Dabei kann auch eine Liste von Argumenten übergeben werden:

Code

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

Cast-Notation

Da Typumwandlungen verschiedenen Zwecken dienen können, existieren zur genauer definierten Typumwandlung vier Operatoren: `static_cast, dynamic_cast, const_cast, reinterpret_cast`.

Die beiden gängigsten sind die ersten beiden. Sie werden im Folgenden näher 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.

Code

```
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. 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 bei Zeigern ein `nullptr` zurückgegeben und bei Referenzen eine `bad_cast`-Exception geworfen.

Code

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

void f() {
    D d;
    B& b_ref = d;

    D* d_ptr = dynamic_cast<D*>(&b_ref);  // OK, b_ref ist Instanz von D.
    D& d_ref = dynamic_cast<D&>(b_ref);   // OK, analog.

    E* e_ptr = dynamic_cast<E*>(&b_ref);  // e_ptr ist nullptr. b_ref ist keine Instanz von E.
    E& e_ref = dynamic_cast<E&>(b_ref);   // Exception. b_ref ist keine Instanz von E.
}
```

- Für das Casting von `shared_ptr` gibt es entsprechende Cast-Operatoren:
  - `shared_ptr<T> static_pointer_cast (const shared_ptr<U>& sp)` und
  - `shared_ptr<T> dynamic_pointer_cast (const shared_ptr<U>& sp)`

  Diese liefern jeweils eine Kopie des übergebenen Parameters mit denselben Bedingungen und Ergebnissen wie für normale Pointer beschrieben. Es handelt sich dabei im Prinzip nur um eine Kurzschreibweise für z.B. `static_cast<T*>(sp.get())`.