PR3 - Leseaufgabe zur Vorlesung 10

(Stand 2022-07-21 10:17)

Prof. Dr. Holger Peine
Hochschule Hannover
Fakultät IV – Abteilung Informatik
Raum 1H.2.60, Tel. 0511-9296-1830
Holger.Peine@hs-hannover.de

L.10 Klassen, Objekte, Strings, Operatoren, Exceptions, Streams

L.10.1 Klassen und Objekte

In der Vorlesung haben wir eine Klasse zur Repräsentation eines Ortes vorgestellt. Hier wiederholen wir einen Zwischenstand der Implementierung der Klasse und des Hauptprogramms:

// Ort.h

```
#ifndef ORT_H
#define ORT_H
class Ort {
private:
  double breite; // Breitengrad
  double laenge; // Laengengrad
  double hoehe;  // Hoehe ueber NN in m
  char* name;    // Ortsname
  void initName(const char* pname);
public:
  double getBreite();
  double getLaenge();
  double getHoehe();
  const char* getName();
  Ort(double breite, double laenge, double hoehe, const char* name);
  ~Ort();
};
#endif
```

// Ort.cpp

```
#include <cstring>
#include <cstdlib>
#include "Ort.h"

void Ort::initName(const char* pname) {
  name= (char*)malloc(strlen(pname)+1);
  strcpy(name, pname);
}
double Ort::getBreite() { return breite; }
double Ort::getLaenge() { return laenge; }
double Ort::getHoehe() { return hoehe; }
const char* Ort::getName() { return name; }
Ort::Ort(double breite, double laenge, double hoehe, const char* name) {
  this->breite= breite; this->laenge= laenge; this->hoehe= hoehe;
  initName(name);
}
Ort::~Ort() {
  free(name);  name = nullptr;
}
```

// main.cpp

```
#include "Ort.h"
int main() {
  Ort hannover {52.3667, 9.71667, 55.0, "Hannover"};
  return 0;
}
```

L.10.1.1 Initialisierung von Instanzattributen

In Java ist es möglich, Attribute direkt bei ihrer Deklaration zu initialisieren:

Java

```
class Ort {
  private double breite = 0.0;
  private double laenge = 0.0;
  ...
```

Derartiges ist in C++ nicht möglich. Die Initialisierung obliegt einzig und allein den Konstruktoren.

L.10.1.2 Klassenattribute

Wie in Java gibt es auch in C++ Attribute, die nur einmal pro Klasse existieren. Das verwendete Schlüsselwort lautet wie in Java `static`.

Ein Klassenattribut muss außerhalb der Klasse initialisiert werden. Als Beispiel ergänzen wir die Klasse `Ort` um ein Instanzattribut `id` und ein Klassenattribut `last_id`:

```
// Ort.h
...
class Ort {
private:
  static int last_id; // Letzte vergebene ID aller Städte
  int id;             // ID dieser Stadt
  double breite;
  ...
};
```

```
// Ort.cpp
...
#include "Ort.h"

int Ort::last_id = 0; // Zu Beginn noch keine IDs vergeben
...
Ort::Ort(double breite, double laenge, double hoehe, const char* name) {
  last_id++;
  Id = last_id;
  this->breite = breite; this->laenge = laenge; this->hoehe = hoehe;
  initName(name);
}
...
```

Erläuterung:

- Die Header-Datei deklariert die beiden neuen Attribute.
- Im Attribut `last_id` wird jeweils die letzte vergebene ID gespeichert.
- Der Konstruktor erhöht `last_id` jeweils um 1 und übernimmt die neue ID in das Instanzattribut `id`.
- Die Deklaration des Klassenattributs `last_id` muss das Schlüsselwort `static` beinhalten. Bei der Definition in der Quelldatei darf `static` nicht angegeben werden. Die Zeile
  ```
  int Ort::last_id = 0;
  ```
  ist syntaktisch einer globalen Variablendefinition in C ähnlich, nur dass hier die Definition nicht „Programm“-global ist, sondern „`Ort`“-global, was durch das vorangestellte `Ort::` ausgedrückt wird.

L.10.1.3 `const`-Attribute

Attribute mit dem Modifikator `const` können nach ihrer Initialisierung nicht mehr geändert werden. Sie müssen bereits **vor** dem Eintritt in den Konstruktor-Rumpf mit ihrem Wert initialisiert werden. Dazu gibt es in C++ die Initialisierungsliste des Konstruktors, die vor der öffnenden geschweiften Klammer des Konstruktorkörpers angegeben wird.

Als Beispiel wollen wir das eben eingeführte Instanzattribut `id` konstant machen, da sich die ID eines Ortes eigentlich nie ändern soll.

```
// Ort.h
...
class Ort {
private:
  static int last_id; // Letzte vergebene ID
  const int id;       // ID dieser Stadt
  double breite;
  ...
};
```

In dieser Situation ist nun das Quellmodul `Ort.cpp` nicht mehr übersetzbar. Wir erhalten die Fehlermeldung

```
Ort.cpp: In constructor 'Ort::Ort(double, double, double, const char*)':
Ort.cpp:__: error: uninitialized member 'Ort::id' with 'const' type 'const int'
Ort.cpp:__: error: assignment of read-only data-member 'Ort::id'
```

Wir müssen die Initialisierung des Attributes in der Initialisierungsliste vornehmen:

```
// Ort.cpp
...
#include "Ort.h"

int Ort::last_id {0}; // Zu Beginn noch keine IDs vergeben
...
Ort::Ort(double breite, double laenge, double hoehe, const char* name)
  : id {++last_id} {
  this->breite = breite; this->laenge = laenge; this->hoehe = hoehe;
  initName(name);
}
...
```

Wir haben hier den Prefix-Inkrementoperator gewählt, weil wir das Ergebnis des Inkrements als Wert für `id` verwenden wollen[1].

L.10.1.4 Klassenmethoden

Wie in Java gibt es auch in C++ Klassenmethoden, die mit dem Schlüsselwort `static` gekennzeichnet werden. Wir schreiben zur Illustration eine Methode, die die nächste zu vergebende ID berechnet und zurück gibt:

```
// Ort.h
class Ort {
private:
  static int last_id;     // Letzte vergebene ID aller Städte
  const int id;           // ID dieser Stadt
  static int getNextId(); // Liefert die naechste zu vergebende ID
  double breite;
  ...
};
```

[1] Der Code `:id {last_id++}` würde `id` mit dem Wert von `last_id` belegen und *danach* `last_id` um 1 erhöhen.

Die Definition der Methode folgt im Quellmodul ohne das `static`-Schlüsselwort:

```
// Ort.cpp
...
#include "Ort.h"

int Ort::last_id {0}; // Zu Beginn noch keine IDs vergeben

int Ort::getNextId() {
  last_id++;
  return last_id;
}
...
Ort::Ort(double breite, double laenge, double hoehe, const char* name)
  : id {getNextId()} {
  this->breite = breite; this->laenge = laenge; this->hoehe = hoehe;
  initName(name);
}
...
```

Einen Aufruf haben wir im Konstruktor in der Initialisierung von `id` programmiert.

L.10.1.5 Objekte als Parameter

Objekte können wie C-structs als Parameter an eine Funktion übergeben werden. Dabei werden die Objekte wie in C als Kopie (call by value) übergeben. "Kopie" bedeutet in C++, dass das Parameter-Objekt aus dem Argument-Objekt durch Aufruf des Copy-Konstruktors erzeugt wird. Auch als Rückgabewert sind Objekte in Wertkopie möglich.

Für die call-by-Reference-Semantik gibt es in C++ Referenz-Typen. Parameter als Referenztypen sind natürlich einerseits sinnvoll, wenn die Methode den Inhalt des übergebenen Objekts ändern will. Referenz-Typen werden in C++ jedoch sehr häufig auch bei nicht zu verändernden Parametern verwendet, dann allerdings in Verbindung mit dem Schlüsselwort `const`. Der Vorteil: Eine Referenz als Parameter vermeidet das zeitaufwändige Kopieren des übergebenen Objekts.

Beispiel:

```
bool hoeher(const Ort& a, const Ort& b) {
  return(a.getHoehe() > b.getHoehe());
}
...
Ort hannover {52.3667, 9.71667, 55.0, "Hannover"};
Ort rio {-22.9, -43.23333, 32.0, "Rio de Janeiro"};
bool b {hoeher(rio, hannover)};
```

Die Übergabe der beiden Parameter an die Funktion `hoeher` ist so effizient wie die Übergabe zweier Zeiger. Die Syntax und Handhabung beim Aufruf und bei der Verarbeitung in der aufgerufenen Funktion ist jedoch einfacher, weil wir nicht ständig selbst an die Dereferenzierung denken müssen.

Auch als Rückgabewert sind `const`-Referenzen einsetzbar:

```
const Ort& hoehererOrt(const Ort& a, const Ort& b) {
  if (a.getHoehe() > b.getHoehe()) return a;
  return b;
}
...
cout << hoehererOrt(rio, hannover).getName() << endl;
```

Alle vorstehenden Codezeilen funktionieren nur dann, wenn die get-Methoden als Beobachtermethoden (also `const`) deklariert wurden. Mehr dazu in den Vorlesungsfolien.

Zum Nachdenken ein kleines Beispiel:

```
class A {
public:
  int x, y;
  A(int px, int py) : x {px}, y {py} {}
  A(const A& a) : x(a.x), y(55) { }
  void print() const { cout << x << " " << y << endl; }
};

void f1(const A &a) {
  a.print();
}
void f2(const A a) {
  a.print();
}

int main() {
  A a {1, 2};
  f1(a);
  f2(a);
  return 0;
}
```

Wie lautet hier die Ausgabe?
Bitte erst nachdenken und dann zur nächsten Seite umblättern…

Antwort:

- Beim Aufruf von `f1` wird der Übergabeparameter nicht kopiert (da Refererenz-Parameter). In diesem Fall werden also die Originaldatenelemente von `a` bei der Weiterleitung an `print` verwendet.
- Beim Aufruf von `f2` wird der Übergabeparameter kopiert (Call-by-value, keine Referenz). Falls ein Typ einen Copy-Konstruktor definiert, dann wird der hier aufgerufen. Der Copy-Konstruktor jedoch setzt das Datenelement `y` stets auf 55. Bei der Weiterleitung an `print` wird also nur das Datenelement `a.x` im Original weiter gereicht. `a.y` hat hier immer den Wert 55.
- Die Ausgabe lautet daher:

```
1 2
1 55
```

L.10.1.6 Konstruktion und Destruktion eines temporären Objekts

Manchmal erzeugt man temporäre Objekte, die nur für die Dauer existieren, in der ein Ausdruck ausgewertet wird. Beispiel:

```
void f() {
  bool b {hoeher( Ort {52.3667, 9.71667, 55.0, "Hannover"},
                  Ort {-22.9, -43.23333, 32.0, "Rio de Janeiro"});
  cout << b << endl;
}
```

Hier passiert zur Laufzeit folgendes:

- Zwei Aufrufe des `Ort`-Konstruktors, jeweils für unterschiedliche Objekte (die Reihenfolge der Konstruktoraufrufe ist nicht spezifiziert),
- Übergabe der Objekte per Call-by-reference an die `hoeher`-Funktion,
- Speicherung des Ergebnisse nach Rückkehr von `hoeher` in der Variable `b`,
- Destruktor-Aufruf für die beiden Objekte (Reihenfolge nicht spezifiziert),
- Ausgabe des Wertes von `b`.

Die beiden `Ort`-Objekte in diesem Beispiel nennt man übrigens "anonyme Objekte", weil sie in keiner Variable (die ja einen Namen hätte) gespeichert werden, sondern nur für die Dauer der Auswertung des Ausdrucks existieren, dessen Teil sie sind (hier: Argumentübergabe an `hoeher`). Die beiden `Ort`-Objekte werden bereits vor Ende des Blocks wieder zerstört („destruiert“).

Es gibt auch temporäre Objekte, deren Konstruktion gar nicht im Quelltext sichtbar ist (bei den beiden anonymen Objekten in unserem Beispiel ist ja ein Konstruktoraufruf sichtbar). Falls etwa `hoeher` als Parametertyp nicht `Ort&` (bzw. `const Ort&`) hätte, sondern einfach `Ort` (call-by-value), dann würde bei der Parameterübergabe ein temporäres `Ort`-Objekt mittels Copy-Konstruktor erzeugt, das dann in `hoeher` über den Parameter angesprochen wird und nach Ende von `hoeher` wieder zerstört wird.

Wir betrachten dazu eine kleine Übung, die Sie selbst nachvollziehen sollten. Gegeben ist eine Klasse `X`:

```
class X {
public:
  int i;
  X()            { i = 0;   cout << " S" << i; }
  X(int pi)      { i = pi;  cout << " +" << i; }
  X(const X& x)  { i = x.i; cout << " C" << i; }
  ~X()           {          cout << " -" << i; }
};
```

Welche Ausgabe produziert das folgende Programmfragment?

```
      void aufgabe() {
        X f;
        f = X {5};
        cout << " |";
      }
```

Die Antwort finden Sie am Ende dieses Dokumentes in Abschnitt L.10.5. Bitte vor dem Blättern zunächst selbst nachdenken!

Seit C++11 gibt es eine Möglichkeit, die Kosten der Konstruktion von kurzlebigen temporären Objekten fast auf Null zu drücken, und zwar durch die "Move-Semantik", die mit "r-value references" (Syntax: `T&&` für einen Typ `T`) arbeitet. Dies wird aus Zeitgründen nicht in dieser Vorlesung behandelt.

L.10.2 Operatoren

L.10.2.1 Zuweisungsoperatoren

Wenn eine Klasse nicht flach kopiert werden darf, sondern eine tiefe Kopie benötigt, muss man dies nicht nur im Copy-Konstruktor implementieren, sondern die Klasse benötigt auch einen eigenen Zuweisungsoperator, der eine tiefe Kopie implementiert (da der vom Compiler erzeugte Default-Zuweisungsoperator nur eine flache Kopie durchführt). Außerdem muss ein Zuweisungsoperator ggf. zuvor vom zu überschreibenden Objekt belegte Ressourcen freigeben, z.B. so:

```
// Vektor.cpp
#include " Vektor.h"
…
Vektor& operator = (const Vektor& v) {
   if (this ==  &v) return *this;  // Muss (und darf!) nichts tun
   delete[] elems;   // Ressourcen freigeben
   // Ab hier die gleiche Funktionalität wie ein Copy-Konstruktor:
   elems = new double[N];
   for (int i = 0; i < N; ++i)
     elems[i] = v.elems[i];
   return *this;
}
```

Meistens benötigen solche Klassen auch einen Destruktor (der ebenfalls die belegten Ressourcen freigibt), weshalb die "Dreierregel" gilt:

> *Wenn eine Klasse einen Copy-Konstruktor oder einen Destruktor oder einen Zuweisungsoperator benötigt, dann benötigt sie alle drei von diesen.*

Jeder `operator = (…)` sollte zuallerst prüfen, ob das zugewiesene Objekt identisch mit dem aktuellen Objekt ist (hier: `this ==  &v`) und in dem Fall sofort enden. Dies spart nicht nur unnötige Arbeit, sondern eine "Zuweisung an sich selbst" würde sogar meistens zu falschen Ergebnissen führen. In der obigen Klasse Vektor würden z.B. `delete[] elems` in diesem Fall auch `v.elems` löschen, so dass die Elemente verloren wären.

Um Code-Redundanz zu vermeiden, ruft der Copy-Konstruktor oft den Zuweisungsoperator auf, z.B. so:

```
// Vektor.cpp
#include " Vektor.h"
…
// Copy-Konstruktor
Vektor(const Vektor& v) {
   elems = nullptr;  // Undefinierten Wert überschreiben, damit das
                     // delete[] elems in operator=() nicht abstuerzt
```

```
                    // (delete nullptr oder delete[] nullptr hat keine Wirkung)
   *this = v;  // Zuweisungsoperator nutzen vermeidet redundanten Code
}
```

L.10.2.2 Mehrfache Überladung eines Operators

Ein Operator kann mehrfach überladen werden, wenn die Signaturen unterschiedlich sind (also wie ganz normale Funktionen). Wir zeigen ein Beispiel für den `operator+=`. Die erste im Folgenden gezeigte Variante stammt aus den Vorlesungsfolien, die zweite, fett dargestellte Variante ist neu:

```
class Vektor {
  ...
  Vektor& operator+=(const Vektor & z2);
  Vektor& operator+=(const double d[]);
};
```

```
// Vektor.cpp
#include "Vektor.h"
…
Vektor& operator += (const Vektor& v) {
   for (int i = 0; i < N; ++i)
     elems[i] += v.elems[i];
   return *this;
}
Vektor& Vektor::operator+=(const double d[]) {
  for (int i = 0; i < N; ++i)
     elems[i] += d[i];
  return *this;
}
```

Wir können den zweiten Operator wie folgt verwenden:

```
  double values1[N] { 1, 1 };
  double values2[N] { 2, 2 };
  Vektor v {values1};  // Siehe nächsten Abschnitt für diesen Konstruktor
  v += values2;
  cout << v << endl; // (3.0, 3.0)
```

L.10.2.3 `operator+` als globale Funktion

In den Folien haben wir den `operator+` als Element-Funktion umgesetzt. Wie würde die Umsetzung als globale Funktion aussehen?

```
// Vektor.h
class Vektor {
  …
};
Vektor operator+(const Vektor& z1, const Vektor &z2) const;
```

```
// Vektor.cpp
#include " Vektor.h"
…
Vektor::Vektor(const double values[]) { // Konstruktor nicht auf den Folien!
  elems = new double[N];
  for (int i = 0; i < N; ++i)
    elems[i] = values[i];
}

Vektor operator + (const Vektor& z1, const Vektor& z2) const {
  Vektor result {};
```

```
   for (int i = 0; i < N; ++i)
     result.elems[i] = z1.elems[i] + z2.elems[i];
   return result;
}
```

Leider erhalten wir eine Fehlermeldung des Compilers. Der Zugriff auf das private Attribut `elems` ist aus der globalen Funktion nicht erlaubt. Zur Abhilfe kann man (wie Sie es auch aus Java kennen) eine get-Methode für dieses Attribut schreiben (Achtung: eine solche Methode sollte eine frische Kopie des Arrays zurückgeben); das offenbart aber die Implementierung nicht nur für `operator+()`, sondern für *jeden* Code außerhalb der Klasse. Eine bessere Möglichkeit beschreibt der folgende Abschnitt.

L.10.2.4 Friends

In C++ gibt es eine Möglichkeit, Zugriff auf private Daten einer Klasse ganz gezielt zu vergeben: `friend`'s. Eine Klasse kann bestimmte andere Programmteile zu ihren "Freunden" erklären. Freunde haben Zugriff auf private Daten und Methoden.

Natürlich sollte man dieses Mittel mit Vorsicht einsetzen, da es die Datenkapselung der Klasse verletzt. Bei Operatoren, die als globale Funktion realisiert werden, kann man dieses Konzept jedoch sinnvoll einsetzen. Operatoren kann man als zur Klasse gehörig betrachten. Z. B. war die in der Vorlesung vorgestellte Umsetzung des `operator<<` als globale Funktion nicht konzeptionell motiviert, sondern durch die begrenzten programmiersprachlichen Mittel begründet (denn es war nicht möglich, die Ausgabeoperation als Methode der feststehenden Bibliotheksklasse `ostream` hinzuzufügen).

Wir wollen hier als Beispiel den `operator+` als Freund der Klasse deklarieren. Dazu deklarieren wir in der Klasse mit dem Schlüsselwort `friend` einen Verweis auf die anderweitig definierte, globale Funktion:

```
// Vektor.h
class Vektor { …
  friend Vektor operator+(const Vektor& z1, const Vektor& z2);
};
```

Die ursprünglich unter der Klasse angegebene Deklaration der globalen Funktion kann nun entfallen. Es reicht die Deklaration als `friend`. Beachten Sie, dass eine globale Funktion (anders als eine Methode) nicht `const` sein kann, da es gar kein Objekt gibt, dass sie unverändert lassen könnte.

Man kann das `friend` auch benutzen, um eine ganze Klasse zum Freund zu deklarieren:

```
  friend class OtherClass;
```

Nun können alle `OtherClass`-Methoden auf private Elemente der Klasse `Vektor` zugreifen.

Oder man gewährt gezielt einzelnen Methoden einer anderen Klasse Zugriff. Dann ist die Syntax wie folgt:

```
  friend int OtherClass::method1(…);
```

L.10.2.5 Vergleichs-Operatoren

Der Vergleichsoperator `operator==` sollte als Elementfunktion realisiert werden:

```
// Vektor.h
class Vektor {
  …
```

```
  bool operator==(const Vektor& v) const;
};
```

```
// Vektor.cpp
#include " Vektor.h"
…
bool Vektor::operator==(const Vektor& v) const {
  for (int i = 0; i < N; ++i)
    if (elems[i] != v.elems[i])
       return false;
  return true;
}
```

Mit diesen Änderungen kann man den Vergleich wie folgt durchführen:

```
  double values1[N] { 1, 1 };
  double values2[N] { 1, 1 };
  Vektor v1 {values1};
  Vektor v2 {values2};
  cout << boolalpha << (v1 == v2) << endl; // Ausgabe: true
```

Wir wollen den Vergleichsoperator mit einer naheliegenden Version überladen, die einen Vektor mit einem Array aus reellen Zahlen (`double`) vergleicht:

```
// Vektor.h
class Vektor {
  …
  bool operator==(const double d[]) const;
};
```

```
// Vektor.cpp
#include " Vektor.h"
…
bool operator == (const double d[]) const {
  for (int i = 0; i < N; ++i)
    if (elems[i] != d[i])
        return false;
  return true;
}
```

```
  double values[N] {1, 2};
  Vektor v {values};
  cout << boolalpha << (v == values) << endl; // Ausgabe: true
  cout << boolalpha << (values == v) << endl; // Ausgabe: ???
```

Die erste Ausgabezeile des Testcodes ruft die Methode `v.operator==(values)` auf. Was aber ruft die zweite Zeile auf? `values.operator==(v)`? Da `values` kein Objekt ist (sondern ein `double[]`), kann der Compiler diesen Aufruf nicht übersetzen und bricht mit der Fehlermeldung

```
  error: no match for 'operator==' in 'values == v'
```

ab. Es gibt in C++ keinen Automatismus, der die beiden Operanden einfach vertauscht. Wir müssen die Variante „`double == Vektor`“ selbst implementieren. Dabei greifen wir (da der erste Parameter kein Objekt ist) auf den Mechanismus der globalen Funktion zurück:

```
// Vektor.h
class Vektor {
  …
  bool operator==(const Vektor& v) const;
  bool operator==(const double d[]) const;
  friend bool operator==(const double d[], const Vektor& c);
};
```

```
// Vektor.cpp
#include "Vektor.h"
…
bool Vektor::operator==(const Vektor& v) const {
  return *this == v.elems;  // Ueberladenen Vektor::operator== aufrufen
}
bool Vektor::operator==(const double d[]) const {
  for (int i = 0; i < N; ++i)
    if (elems[i] != d[i])
        return false;
  return true;
}
bool operator==(const double[] d, const Vektor& c) {
  return (c==d); // Vektor::operator== aufrufen
}
```

```
  double values1[N] { 1, 1 };
  double values2[N] { 1, 1 };
  double values3[N] { 3, 3 };
  Vektor v1 {values1};
  Vektor v2 {values2};
  Vektor v3 {values3};
  cout << boolalpha << (v1 == v2) << endl; // Ausgabe: true
  cout << boolalpha << (v3 == values3) << endl; // Ausgabe: true
  cout << boolalpha << (values3 == v3) << endl; // Ausgabe: true
```

Es ist grundsätzlich guter C++-Stil, kommutative Operatoren stets symmetrisch zu programmieren. Außerdem kann ein Nutzer unserer Klasse `Vektor` nun erwarten, ebenfalls einen `operator!=` aufrufen zu können, dessen Implementierung Ihnen zur Übung empfohlen sei.

L.10.3 Exceptions

Der Array-Zugriff mit `[]`-Klammern kann für beliebige Objekte überschrieben werden – als sog. Indexoperator. Der Parametertyp ist dabei nicht auf positive ganze Zahlen beschränkt:

```
// Vektor.h
#include <stdexcept>
class Vektor {
  …
  double operator[](int index) const throw(invalid_argument);
};
```

```
// Vektor.cpp
#include " Vektor.h"
#include <stdexcept>
…
double& Vektor::operator[](int index) const throw(invalid_argument) {
  if (index < 0 || index >= N)
     throw invalid_argument("Vektor::operator[] illegal index " + index);
  return elems[index];
}
```

Im vorstehenden Quelltext sehen wir, wie man in C++ eine Exception erzeugt. Eine Exception ist üblicherweise in C++ genauso wie in Java ein Objekt, allerdings benutzt man kein auf dem Heap angelegtes Objekt (denn das müsste jemand später wieder löschen), sondern ein

(anonymes) Objekt auf dem Stack als Wert-Kopie. Wir erzeugen ein Objekt des Typs `invalid_argument`, welcher in `<stdexcept>` als Exception-Typ deklariert ist.

```
double values[N] { 3, 3 };
Vektor v {values};
cout << v[0] << ", " << v[1] << endl;  // Ausgabe: 3, 3
cout << v[5];                          // Exception
```

Die letzte Programmzeile liefert erwartungsgemäß zur Laufzeit folgende Ausgabe:

```
terminate called after throwing an instance of 'std::invalid_argument'
  what():  Vektor::operator[] illegal index 5
Aborted (core dumped)
```

Zur Demonstration, wie man diese fängt:

```
try {
  double values[N] { 3, 3 };
  Vektor v {values};
  cout << v[0] << ", " << v[1] << endl;  // Ausgabe: 3, 3
  cout << v[5];                      // Exception
} catch(invalid_argument e) {
  cerr << "He he, was soll das denn für ein Index sein?" << endl
       << e.what() << endl;
}
```

Weitere `catch`-Blöcke könnte man syntaktisch anhängen, was hier jedoch keinen Sinn machen würde, weil ja nur der eine Exception-Typ geworfen werden kann. Ein `finally` ist nicht möglich.

Die Ausgabe erfolgt in dem Beispiel im Fehlerfall übrigens auf die Standardfehlerausgabe. Hierfür gibt es in C++ (neben den schon bekannten Strömen `cin` für die Standardeingabe und `cout` für die Standardausgabe) den Strom `cerr`.

L.10.4 Streams

L.10.4.1 Hexadezimale Ausgabe

Streams wie `cout` können durch sog. Manipulatoren in ihrem Verhalten verändert werden. Ein Manipulator ist ein Objekt, das in den Ausgabestrom eingefügt wird, aber nicht zu einer Ausgabe führt, sondern zu einer Zustandsänderung des Streams, die sich dann auf spätere Ausgaben auswirkt.

Beispiel:

```
#include <iomanip>
#include <iostream>
...
int x {16384};
cout<<setw(15)<<left<<"dezimal:"<<setw(10)<<right<<dec<<x<<endl
    <<setw(15)<<left<<"hex:"    <<setw(10)<<right<<hex<<x<<endl;
```

Die Ausgabe lautet:

```
dezimal:           16384
hex:                4000
```

Zur Verwendung von Manipulatoren muss der Header `<iomanip>` inkludiert werden.

Noch ein Beispiel:

```
bool b {5>3};
cout << boolalpha << b << endl;
```

Die Ausgabe lautet:

```
true
```

Mehr Details zu Manipulatoren finden Sie z. B. hier: http://www.cplusplus.com/reference/iostream/ .

L.10.4.2 String-Streams

String-Streams sind Speicherbereiche, die sich wie Streams verhalten, deren Daten in einem `string`-Objekt gespeichert werde. Man könnte einen Eingabe-String-Stream verwenden, um die Anzahl der Wörter in einem Text zu ermitteln:

```
#include <sstream>
...
int countwords(const string &s) {
  istringstream is {s};
  int count {0};
  string w;
  while (is >> w) count++;
  return count;
}
```

Ausgabe-String-Streams werden häufig für die Zusammenstellung einer Meldung aus mehreren Komponenten benutzt:

```
  const int max= 100;
  int i;
  do {
    cin >> i;
    ostringstream msg;
    if (i>=0 && i<=max) break;

    msg << "Die Eingabe " << i
        << " liegt nicht im gewünschten Bereich [0," << max << "]";
    cerr << msg.str() << endl;
  } while(true);
```

L.10.5 Lösung der Übung aus Abschnitt L.10.1.6

Die Ausgabe lautet:

```
S0 +5 -5 | -5
```

Erläuterung:

```
1  void aufgabe() {
2    X f;
3    f = X {5};
4    cout << " |";
5  }
```

- *In Zeile 2 wird der Konstruktor ohne Parameter aufgerufen (Ausgabe `S0`).*
- *Zeile 3 erzeugt zunächst ein temporäres Objekt durch Aufruf des Konstrukors `X {5}` mit `int`-Parameter (Ausgabe `+5`). Die Zuweisung des temporären Objekts an `f` erzeugt keine Ausgabe. Am Ende von Zeile 3 wird der Destruktor des temporären Objekts aufgerufen (Ausgabe `-5`), weil dieses Objekt (nicht verwechseln mit `f`!) nun nicht mehr gebraucht wird.*
- *Nun folgt in Zeile 4 die Ausgabe von `|`.*
- *Schließlich wird am Ende des Blocks (Zeile 5) der Destruktor der automatischen Variable `f` aufgerufen, weil `f` außerhalb der Funktion `aufgabe()` nicht mehr exisitiert (Ausgabe `-5`).*