Einführung in die Software-Entwicklung

Vom Programmieren zur erfolgreichen Software-Projektarbeit

am Beispiel von Java und C++

- Von der Idee zum Algorithmus
- Funktionen und Datenstrukturen
- Organisation des Quellcodes
- Werte- und Referenzsemantik
- Entwurf von Algorithmen
- Fehlersuche und -behandlung
- Software-Entwicklung im Team
- Abstrakte Datentypen: Einheit von Daten und Funktionalität
- Vielgestaltig (Polymorphie)
- Entwurfsprinzipien für Software

© Hartmut Helmke, Frank Höppner, Rolf Isernhagen

Dr.-Ing. Hartmut Helmke ist im Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR) am Institut für Flugführung in Braunschweig Leiter des Kerngebiets *Flughafen und Flughafennahbereich* und Lehrbeauftragter für Programmierkonzepte an der Fachhochschule Braunschweig/Wolfenbüttel im Fachbereich Informatik.

Prof. Dr.-Ing. Frank Höppner vertritt an der Fachhochschule Braunschweig/Wolfenbüttel im Studiengang Wirtschaftsinformatik die Gebiete Algorithmen und Datenstrukturen, Objektorientierte Programmierung, Informationssysteme und Data Mining.

Prof. i.R. Rolf Isernhagen hat im Fachbereich Informatik an der Fachhochschule Braunschweig/Wolfenbüttel die Gebiete Softwaretechnik, Modellbildung und Simulation zeitdiskreter Systeme vertreten.

Kontakt: http://public.fh-wolfenbuettel.de/ hoeppnef/hanserSWE.html

Text für den Buchrücken

Jeder Programmierer bringt ein 1000-Zeilen-Programm zum Laufen, auch wenn der Entwurf eher kreativ-chaotisch als systematisch erfolgt. Bei der professionellen Software-Entwicklung eines 100000-Zeilen-Programms in einem Team rücken Anforderungen, wie Lesbarkeit, Änderbarkeit und Wartbarkeit in den Vordergrund, die sich auf die gesamte Software-Entwicklung auswirken und die ohne systematische Analyse und systematischen Entwurf schwer zu erfüllen sind.

Bei der Predigt des üblichen top-down-Vorgehens (Analyse, Design, Implementierung) ist dem Anfänger oft nicht ersichtlich, warum dieser Aufwand nötig ist, weil sein 1000-Zeilen-Programmm doch funktioniert. In diesem Buch verfolgen wir daher zunächst einen bottom-up-Ansatz, um die Schwächen davon und die Notwendigkeit zum Handeln aufzuzeigen. Die schrittweise Verbesserung der Lesbarkeit, Änderbarkeit und Wartbarkeit führen uns von der prozeduralen Startlösung bis zur objektorientierten Analyse, Design und Implementierung.

Was erwartet den Leser konkret?

- **Hohe praktische Relevanz:** Das Buch behandelt detailliert Themen wie Quellcode-Organisation (make/Build-File) und Debugging.
- **Niedrige Einstiegshürde:** Wir holen den Leser mit Kenntnissen in prozeduraler Programmierung ab und vermitteln ihm Wissen in objektorientierter Software-Entwicklung.
- **Roter Faden:** Verständlichkeit, Wartbarkeit und Änderbarkeit von Software sind der Motor für eine schrittweise Verbesserung der Programmiertechnik von Kapitel zu Kapitel. Dieser Weg führt in logischer Konsequenz zur objektorientierten Programmierung. Ein einziges durchgehendes Beispiel zieht sich durch alle Kapitel.
- **Objektorientierung:** Herausarbeitung der Fortschritte gegenüber prozeduraler Programmierung, Analyse und Design an einem konkreten Beispiel, bis hin zu Entwurfsmustern

Inhaltsverzeichnis

Kapitel 1

Grundlagen der C++- und der Java-Programmierung

Gemeinsamkeiten, Ähnlichkeiten von C++ und Java

Elementare Datentypen: Typische Deklarationen für elementare Datentypen sind in C++ und Java identisch, z.B.

- `int a, b, c;`
- `double x, y, z;`

In Java ist die Darstellung der elementaren Datentypen festgeschrieben, z.B. 32 Bit für `int`, in C++ ist sie implementierungsabhängig. Darüber hinaus gibt es aber auch weitere syntaktische und semantische Unterschiede.

Operatoren, Ausdrücke und Anweisungen: Bei den Operatoren für elementare Datentypen und bei der Bildung von Ausdrücken gibt es nur sehr geringe Unterschiede; hier zwei Beispiele, die sowohl in C++ als auch Java syntaktisch und semantisch übereinstimmen, vorausgesetzt, die Variablen sind entsprechend deklariert:

- `schaltjahr = jahr%4 == 0 && jahr%100 != 0 || jahr%400 == 0;`
- `n += ++a + b--;`

Steueranweisungen: Auch bei den Steueranweisungen gibt es kaum Unterschiede; hier zwei Beispiele, die sowohl C++- als auch Java-Code sein können:

- `if (x < y) min = x; else min = y;`
- `while (i > n1 && j < n2) { sum = sum + i + j; i--; j++; }`

Unterschiede

Entwurfsphilosophie: Java ist im Wesentlichen eine *orthogonal* konzipierte Sprache, d.h. es gibt in jeder Situation jeweils einen Weg. Objekte elementarer Datentypen werden z.B. als Werte auf dem Programm-Stack verwaltet, selbst definierte Objekte werden per Referenz auf dem Programm-Heap verwaltet.

C++ ist im Gegensatz dazu so konzipiert, dass in jeder Situation zwischen Alternativen gewählt werden kann; z.B. kann jedes Objekt wahlweise im Programm-Stack oder im Programm-Heap verwaltet werden, beim Zugriff hat man zudem häufig die Wahl *Zugriff per Wert*, *Zugriff per Zeiger* oder *Zugriff per Referenz*.

Programmierparadigma: Java ist eine rein objektorientierte Sprache, d.h. als globale Konstrukte eines Programms gibt es nur Klassen, alle Daten sind Attribute von Klassen, alle Vorgänge (Aktionen, Algorithmen) sind Klassen zugeordnet.

C++ ist eine hybride Sprache, d.h. als globale Konstrukte eines Programms können nicht nur Klassen, sondern z.B. auch Variablen und Funktionen verwendet werden. Es ist möglich, in C++ ein klassisches, prozedural strukturiertes Programm (z.B. ein C-Programm) zu implementieren oder ein objektorientiertes oder ein Programm, das beide Ebenen nebeneinander verwendet.

Standardbibliothek: Die Java-Bibliothek ist per se rein objektorientiert, die Polymorphie ist dort zentrales Konstruktionselement. Typisches Beispiel ist die polymorphe Container-Bibliothek. Die Java-Bibliothek bietet eine umfangreiche Unterstützung zur Gestaltung grafischer Benutzeroberflächen.

Die C++-Bibliothek ist hybrid aufgebaut, Polymorphie spielt dort eine eher untergeordnete Rolle. Typisches Beispiel ist die *Standard Template Library (STL)*, bei der die Verwendung von Templates im Vodergrund steht. Die C++-Bibliothek enthält im Standard keine Unterstützung zur Gestaltung grafischer Benutzeroberflächen, sondern überlässt das je nach Betriebssystem individuellen Bibliotheken.

1.1 Ein kleines Beispiel

Listing 1.1 zeigt ein kleines Beispiel, das links als C++-Programm und rechts als Java-Programm dargestellt ist. Startpunkt für die Ausführung ist in beiden Fällen die Funktion `main`. Das Programm bestimmt jeweils die Summe $S = \sum_{i=n1}^{n2} i$ für zwei eingelesene Werte $n1$ und $n2$ mit $n1 \leq n2$ und führt die entsprechende Berechnung dazu in der Funktion (C++) bzw. in der Methode (Java) `summe` durch.

Das *Java-Programm* ist – wie von der Sprache vorgegeben – *rein objektorientiert*, `summe` und `main` sind *Methoden*, d.h. Funktionen, die Bestandteil der Klasse `Summe` sind. Das *C++-Programm* ist – wahlweise absichtlich – *nicht objektorientiert*, sondern *rein klassisch prozedural* formuliert, `summe` und `main` sind *„normale“ Funktionen*. Die Programme werden wie folgt compiliert und aufgerufen (GNU C++-Compiler 3.4.2, SUN JDK 1.5):

C++/Java 1.1: Zahlensumme berechnen

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

int summe(int a, int b) {
  int s = 0;
  for (int i=a; i<=b; i++) s += i;
  return s;
}

int main() {
  int n1, n2, sum;
  do {
    cout << "n1, n2: ";
    cin >> n1 >> n2;
    sum = summe(n1, n2);
    cout << "n1 + n1+1 + ... + n2 = "
         << sum << endl;
  }
  while (n1 <= n2);
}
```

(Basics/anfang/summe.cpp)

```java
import java.util.Scanner;


class Summe {
  static int summe(int a, int b) {
    int s = 0;
    for (int i=a; i<=b; i++) s += i;
    return s;
  }

  public static void main(String[] args) {
    int n1, n2, sum;
    Scanner keyb = new Scanner(System.in);
    do {
      System.out.println("n1, n2: ");
      n1 = keyb.nextInt();
      n2 = keyb.nextInt();
      sum = summe(n1, n2);
      System.out.println("n1 + n1+1 + "+
        "... + n2 = "+sum);
    }
    while (n1 <= n2);
  }
}
```

(Basics/anfang/summe.java)

```
g++ summe.cpp -o summe
summe
```

```
javac Summe.java
java Summe
```

In C++ sind `cin` (mit dem Operator <<) und `cout` (mit dem Operator >>) die Standardmedien für die Ein- bzw. Ausgabe (Tastatur bzw. Bildschirm). Sie sind in dem Bibliotheksmodul `iostream` definiert. Die vollständigen Namen für die beiden Objekte sind `std::cin` bzw. `std::cout`, weil sie im Namensbereich `std` (für *standard*) liegen. Mit `using namespace std;` werden die entsprechenden Namen automatisch expandiert.

In Java heißen die Tastatur-Eingabe- und Bildschirm-Ausgabe-Ströme `System.in` und `System.out`. Um die Arbeit mit dem Eingabe-Strom zu vereinfachen, wird die Hilfsvariable `keyb` vom Typ `Scanner` benutzt, die den Eingabe-Strom als Datenquelle zugewiesen bekommt. Diese Hilfsklasse wird über die `import`-Anweisung verfügbar. Über `keyb.nextInt()` wird von der Tastatur eine ganze Zahl eingelesen.

Wie schon erwähnt, werden in Java alle Funktionen und Variablen Klassen (bzw. Objekten) zugeordnet. Es geht zunächst um die klassische, prozedurale Programmierung. Wir *simulieren* die prozedurale Programmierung daher in Java, indem wir alle Funktionen und Variablen innerhalb einer Klasse mit dem Zusatz `static` versehen. Dann verhalten sich C++ und Java nahezu identisch.

1.2 Trennzeichen (White Spaces) und Kommentare

Trennzeichen (white spaces) der Sprache sind die Zeichen *Zwischenraum (space)*, *horizontaler Tabulator*, *neue Zeile*, *vertikaler Tabulator* und *Seitenvorschub (form feed)*. Die Syntax von C++ und Java erlaubt (außer in Variablennamen) Leerzeichen und Zeilenumbrüche an beliebiger Stelle.

Kommentare sind beliebige Zeichenfolgen, die mit den beiden Zeichen `/*` beginnen und mit `*/` enden bzw. durch die Zeichen `//` bis zum Zeilenende eingeleitet werden:

- Kommentar: `/* . . . */`
- Inline-Kommentar: `// ...` Dieser kann in einem normalen Kommentar geschachtelt sein.

Kommentare werden vom Compiler überlesen und dienen *nur* dazu, Quelltexte für den Menschen besser lesbar zu machen. Kommentare erstrecken sich gegebenenfalls auch über mehr als eine Zeile, können aber nicht geschachtelt werden, d.h. ein Kommentar endet beim ersten Auftreten der Zeichenfolge `*/`, unabhängig davon, wie oft die Zeichenfolge `/*` innerhalb des Kommentars aufgetreten ist. Inline-Kommentare enden jeweils am Zeilenende. Innerhalb der Kommentarzeichen `/*` und `*/` können beliebig viele Inline-Kommentare auftreten.

1.3 Daten, Operatoren, Ausdrücke, Anweisungen

1.3.1 Namen für Variablen (und für andere Sprachkonstrukte)

Für die Wahl von Variablennamen gilt folgende Regel:

- Ein Name besteht aus beliebigen Folgen von Buchstaben, Ziffern und Unterstrichen. Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein. In Java sind zusätzlich das Dollarzeichen (`$`) und Umlaute an jeder Stelle im Variablennamen erlaubt.
- Bzgl. der Länge kann es implementierungsabhängige Einschränkungen geben.
- Es wird zwischen Klein- und Großschreibung unterschieden.
- Ein Name darf kein Schlüsselwort (z.B. `break, true, false, char, double, int, class, catch, try,` ...) sein.

Die Regel für die Bildung von Variablennamen gilt auch für die Bildung von Namen (identifier) für andere Sprachkonstrukte, wie z.B. für Typen, Konstanten und Funktionen. Einige Beispiele für korrekte und falsche C++- und Java-Bezeichnernamen zeigt die folgende Aufstellung:

```
int a;
int _pointer;
int ganz_langer_name_24_undNochLaenger;
int name, Name;       // Groß– und Kleinschreibung wird unterschieden
```

```
int 34Name;           // Fehler: Ziffern am Anfang nicht erlaubt
int Strassen Name;    // Leerzeichen nicht erlaubt: Strassen_Name
int C&A;              // Fehler; Sonderzeichen verboten
int $Name, Mein$Name; // nur in Java erlaubt
```

1.3.2 Deklarationen

Jede Variable muss vor ihrer Benutzung vereinbart werden, z.B. in folgender Form:

```
int i, j, k;
double x, y, z;
char ch;
```

```
int i, j, k;
double x, y, z;
char ch;
```

Eine Variable kann bei ihrer Vereinbarung auch gleich initialisiert werden, z.B.:

```
int i = 0, j = 100, k;
double x = 0.0, z=123.456, max=1.0e20;
char ch = 'A';
bool end = true;
double x1(14.6);
double j1(44);
bool anf(false);
```

```
int i = 0, j = 100, k;
double x = 0.0, z=123.456, max=1.0e20;
char ch = 'A';
boolean end = true;
// double x1(14.6); // Sytaxfehler
// double j1(44);    // Sytaxfehler
// boolean anf(false); // Sytaxfehler
```

In C++ ist die Funktionsschreibweise `double z(123.456)` zusätzlich zur Initialisierung mit dem Gleichheitszeichen `double z = 123.456` möglich. Im Zusammenhang mit Objekten können manche Variablen von benutzerdefinierten Datentypen dann nur über diese Funktionsschreibweise initialisiert werden.

Wird einer Variablen kein initialer Wert zugewiesen, hat sie einen zufälligen Wert. Wenn also eine Initialisierung vergessen wird, kann das zu unerwartetem Verhalten führen. In Java überprüft daher der Compiler, ob einer Variablen vor der Benutzung auch ein Wert zugewiesen wurde, andernfalls gibt es eine Fehlermeldung (*definite assignment*).

1.3.3 Datentypen

In den Tabellen 1.1 und 1.2 sind die elementaren, vordefinierten Datentypen von Java und C++ zusammengestellt. Die Länge in Bytes der einzelnen Typen und die Wertebereiche sind in C++ implementierungsabhängig, in der Tabelle 1.2 sind beispielhafte Werte für 32-Bit-Umgebungen angegeben. In Java sind die Bytelängen und damit die Wertebereiche dagegen plattformunabhängig standardisiert.

Die *Character-Typen* dienen dazu, Zeichen (Buchstaben, Ziffern, Sonderzeichen und Steuerzeichen), wie z.B. Zeichen des ASCII-Zeichensatzes, darzustellen, sie können aber auch – stilistisch unschön – zur Darstellung kleiner Zahlen verwendet werden, d.h. der Zeichencode lässt sich auch als Zahl interpretieren, und es kann mit Character-Typen gerechnet werden. In C++ wird in der Regel der ASCII-Code (1 Byte) zugrunde gelegt, in Java hingegen der Unicode (2 Byte). Durch die zusätzli-

Tabelle 1.1: Elementare Datentypen in Java

Datentyp	Länge in Byte	Wertebereich	Genauigkeit in Dezimalstellen
Character			
`char`	2	$-2^15 \ldots +2^15-1$	4
Integer			
`byte`	1	$-2^7 \ldots +2^7-1$	2
`short`	2	$-2^{15} \ldots +2^{15}-1$	4
`int`	4	$-2^{31} \ldots +2^{31}-1$	9
`long`	8	$-2^{63} \ldots +2^{63}-1$	19
Boolean			
`boolean`		`false, true`	
Aufzählungstypen			
{ *list* }	4	wie `int`	
Gleitkommatypen			
`float`	4	$\sim -10^{38} \ldots \sim +10^{38}$	7
`double`	8	$\sim -10^{308} \ldots \sim +10^{308}$	15

chen Attribute *signed* und *unsigned* kann in C++ festgelegt werden, ob der Bereich positive und negative oder nur vorzeichenlose Werte umfasst.

Zum Arbeiten mit Zeichenketten gibt es in C++ den Datentyp `string` und in Java den Typ `String`. Bei beiden handelt es sich um Bibliothekstypen, die als Klassen definiert sind, worauf wir an dieser Stelle noch nicht näher eingehen wollen. In C++ muss zum Arbeiten mit Strings das Modul `string` eingefügt werden (`#include <string>`).

```
string s1;
string s2 = "hallo";
string s3 = " Welt";
s1 = s2 + s3 + "\n";
cout << s1;
```

```
String s1;
String s2 = "hallo";
String s3 = " Welt";
s1 = s2 + s3 + "\n";
System.out.print(s1);
```

Tabelle 1.2: Elementare Datentypen in C++

Die angegebenen Werte stellen Beispiele für 32-Bit-Umgebungen dar, sie sind implementierungsabhängig!			
Datentyp	Länge in Byte	Wertebereich	Genauigkeit in Dezimalstellen
Character			
`char` ≡ `signed char`	1	$-2^7 \ldots +2^7-1$	2
`unsigned char`	1	$0 \ldots 2^8-1$	2
Integer			
`int` ≡ `signed int`	4	$-2^{31} \ldots +2^{31}-1$	9
`unsigned int`	4	$0 \ldots 2^{32}-1$	9
`short int` ≡ `signed short int`	2	$-2^{15} \ldots +2^{15}-1$	4
`unsigned short int`	2	$0 \ldots 2^{16}-1$	4
`long int` ≡ `signed long int`	4	$-2^{31} \ldots +2^{31}-1$	9
`unsigned long int`	4	$0 \ldots 2^{32}-1$	9
Boolean			
`bool`	1	`false, true`	
Wide Character			
`wchar_t`	2		
Aufzählungstypen			
`enum` *id* ≡ `enum` *id* `{` *list* `}` ≡ `enum {` *list* `}`	4	wie int	
Gleitkommatypen			
`float`	4	$\sim -10^{38} \ldots \sim +10^{38}$	7
`double`	8	$\sim -10^{308} \ldots \sim +10^{308}$	15
`long double`	10	$\sim -10^{4932} \ldots \sim +10^{4932}$	19
Typisierte und nichttypisierte Zeiger			
type`*`	4	$0 \ldots 2^{32}-1$	
`void*`	4	$0 \ldots 2^{32}-1$	
Referenztypen			
type`&`	4	$0 \ldots 2^{32}-1$	

Die *Integer-Typen* dienen zur Darstellung ganzer Zahlen, `int` und `signed int` können in C++ synonym verwendet werden; der Zahlenbereich umfasst dann jeweils beide Vorzeichen. `unsigned int` dient der Darstellung vorzeichenloser ganzer (natürlicher) Zahlen. Aus historischen Gründen und zur Erweiterung des Bereichs um z.B. `EOF` (end of file) mit Wert `-1` werden in C++ Integer-Typen auch zur Darstellung von Zeichen verwendet, verschiedene C-Bibliotheksfunktionen für das Arbeiten mit Zeichen verwenden in dem Zusammenhang bei der Parameterübergabe und als Rückgabetyp den Typ `int`.

Die `short int`- und die `long int`-Typen sind speziell für kleine bzw. große Zahlenbereiche vorgesehen. Bezüglich der Implementierung ist in C++ vorgeschrieben, dass der Wertebereich für `short`-Typen kleiner oder gleich und der für *long*-Typen größer oder gleich dem der `int`-Typen ist:

$$\texttt{short int} \subseteq \texttt{int} \subseteq \texttt{long int}$$

Die Aufzählungstypen `enum` werden in Abschn. 1.8.3 behandelt.

Zur Darstellung von Gleitkommazahlen (Real-Zahlen, Float-Zahlen) stehen die Typen `float` sowie `double` und in C++ zusätzlich `long double` zur Verfügung, wobei die höhere Zahl von Bytes für die Implementierung jeweils genutzt wird, um sowohl den Wertebereich als auch die Genauigkeit zu erhöhen; siehe dazu C++-Bibliotheksmodul `cfloat`.

Beim Rechnen mit Gleitkommazahlen sollte man sich bewusst sein, dass sie die rationalen und die reellen Zahlen aus der Mathematik nur ungenau darstellen können. Daraus folgt unter anderem, dass es nicht sinnvoll sein kann, errechnete Gleitkommawerte auf Gleichheit zu testen!

Die für eine konkrete Implementierung jeweils gültigen Maximal- und Minimalwerte für die ganzzahligen Typen stehen im Bibliotheksmodul `climits` und die Grenzwerte für die Gleitkommatypen im entsprechenden C++-Bibliotheksmodul `cfloat`. In Java gibt es dafür die Konstanten `Integer.MAX_VALUE`, `Double.MAX_VALUE` etc.

```
#include<iostream>
using namespace std;
#include<climits>
#include<cfloat>
int main() {
  cout << " INT_MIN:" << INT_MIN;
  cout << " INT_MAX:" << INT_MAX;
  cout << "\nDBL_MIN:" << DBL_MIN;
  cout << " DBL_MAX:" << DBL_MAX;
  cout << endl;
}
```

```
public class Limits {

public static void main(String[] args){
  System.out.println(Integer.MAX_VALUE);
  System.out.println(Integer.MIN_VALUE);
  System.out.println(Double.MAX_VALUE);
  System.out.println(Double.MIN_VALUE);
}
}
```

1.3.4 Symbolische Konstanten

Mit dem Attribut **const** bzw. **final** kann angegeben werden, dass sich der entsprechende Wert nicht ändern soll, z.B.:

```
const double pi = 3.14159;
const int MaxLen = 1000000;
const char space = ' ';
```

```
final double pi = 3.14159;
final int MaxLen = 1000000;
final char space = ' ';
```

Variablen und Konstanten müssen vor ihrer Benutzung deklariert sein. Allerdings müssen die Vereinbarungen nicht, wie noch bei C, allen Anweisungen voranstehen, sondern Vereinbarungen und Anweisungen können sich in beliebiger Reihenfolge abwechseln. Natürlich muss jeder Name vor seiner ersten Verwendung bekannt sein.

1.4 Operatoren für elementare Datentypen

Die Tabellen 1.3 und 1.4 geben eine Übersicht über die in C++ und Java verfügbaren Operatoren und ihre jeweilige Bedeutung. Operatoren, die nur in C++ oder Java erlaubt sind, sind in den Tabellen jeweils grau unterlegt.

1.4.1 Binäre Operatoren

Arithmetik: `+ - * / %` Die ersten vier Operatoren betreffen die aus der Mathematik bekannten Grundrechenarten für Zahlen und sind auf alle Integer-Typen, auf Character-Typen und auf Gleitkommatypen anwendbar. Der Operator `%` ist nicht auf Gleitkommatypen anwendbar und steht für die Modulo-Operation, d.h. er liefert den Rest einer ganzzahligen Division (`5%2` $\equiv$ `1`, `27%10` $\equiv$ `7`, `35%5` $\equiv$ `0`). Die Division zweier ganzzahliger Operanden liefert ein ganzzahliges Ergebnis (`5/2` $\equiv$ `2`). Eine Operation mit einem ganzzahligen und einem Gleitkommatyp wird im Gleitkommabereich durchgeführt (`5.0/2` $\equiv$ `2.5`). Die Operatoren `+` und `-` können in C++ auch in begrenztem Umfang auf Adressen und Zeiger angewendet werden.

Vergleich: `< <= == != >= >` Die Vergleichsoperatoren sind im Wesentlichen auf alle elementaren Datentypen anwendbar, auf C++-Zeiger allerdings nur der Test auf Gleichheit sowie auf Ungleichheit. Das Ergebnis einer Vergleichsoperation ist vom Typ **bool** bzw. **boolean** (`false` bzw. `true`). Die Vergleichsoperanden sollten möglichst dem gleichen Datentyp angehören, es können aber auch verwandte Datentypen – wie z.B. `int` und `long` – verglichen werden.

Bitoperationen: `& | ~ << >> >>>` Die Operatoren `& | ~` beziehen sich auf die binäre Darstellung ganzzahliger Typen und führen jeweils bitweise UND-, ODER- bzw. EXKLUSIV-ODER-Verknüpfungen durch. `b >> n` schiebt die Bits in `b` um `n`

[1] Fußnote zu den Tabellen 1.3 und 1.4: Darüber hinaus sind einige arithmetische Operatoren – mit Einschränkungen – auch in Verbindung mit Adressen und Zeigern verwendbar; siehe dazu Abschn. 4.2.2.

Tabelle 1.3: C++- und Java-Operatoren für elementare Datentypen, Teil I

Operator	Bedeutung	Art	anwendbar auf
Binäre Operatoren			
`+`	Addition	Arithmetik	Zahlen[1]
`-`	Subtraktion		
`*`	Multiplikation		
`/`	Division		
`%`	Divisionsrest		nur ganze Z.
`<`	Vergl. auf *kleiner*	Vergleich	alle Typen
`<=`	Vergl. auf *kleiner oder gleich*		
`==`	Vergl. auf *gleich*		
`!=`	Vergl. auf *ungleich*		
`>=`	Vergl. auf *größer oder gleich*		
`>`	Vergl. auf *größer*		
`&`	bitw. *UND*-Verknüpfung	Bitoperation	ganzz. Typen
`|`	bitw. *ODER*-Verknüpfung		
`^`	bitw. *EXKL.-ODER*-Verkn.		
`<<`	bitw. Linksschieben		
`>>`	bitw. Rechtsschieben		
`>>>`	bitw. Rechtsschieben, nur Java		
`&&`	log. *UND*-Verkn. in Ausdr.	Logische Verknüpfung	Boolesche Ausdrücke
`||`	log. *ODER*-Verkn. in Ausdr.		
Unäre Operatoren			
`&`	Adresse von, nur C++	Referenzier.	alle Typen
`*`	Inhalt von, nur C++	Dereferenz.	Adressen
`+`	pos. Vorzeichen	Arithmetik	Zahlen
`-`	neg. Vorzeichen		
`~`	bitw. Invertieren	Bitoperation	ganzz. Typen
`!`	log. Invertieren	Log. Verkn.	Bool. Ausdr.
(*type*)	Typkonvertierung	Typisierung	alle Typen
type()	Typkonvertierung, nur C++	Typisierung	alle Typen
`x_cast`<*type*>()[1]	Typkonvertierung, nur C++	Typisierung	alle Typen
`sizeof`	**sizeof** *expr.*: nur C++	Speicherbedarf	Ausdrücke
`sizeof`	**sizeof**(*type*): nur C++	Speicherbedarf	Typen
Postfix- und Präfix-Operatoren			
`++`	Inkrementierung	Arithmetik	ganzz. Typen und Zeiger
`--`	Dekrementierung		

Tabelle 1.4: C++- und Java-Operatoren für elementare Datentypen, Teil II

Operator	Bedeutung	Art	anwendbar auf
Zuweisungsoperatoren			
=	Wertzuweisung	Zuweisung	alle Typen
+= -= *= /= %=	Addition Subtraktion Multiplikation Division Divisionsrest	Arithmetik und Zuweisung	Zahlen[1] nur ganze Z.
&= |= ~= <<= >>= >>>=	bitw. *UND*-Verknüpfung bitw. *ODER*-Verknüpfung bitw. *EXKL.-ODER*-Verkn. bitw. Linksschieben bitw. Rechtsschieben bitw. Rechtsschieben, nur Java	Bitoperation und Zuweisung	ganzz. Typ.
Sonstige Operatoren			
?: ,	Formulierung bed. Ausdrücke Aufzählung in Klammerausdr.		Ausdrücke Ausdrücke

Stellen nach rechts und füllt jeweils mit dem Vorzeichenbit auf (`0` bei `unsigned` und positiven sowie `1` bei negativen Werten von `b`). Dagegen bedeutet `b >>> n` in Java ein Schieben von b um n Stellen nach rechts, wobei vorzeichenunabhängig von b mit Nullbits aufgefüllt wird. Eine semantische Entsprechung in C++ gibt es nicht. Beim Linksschieben werden von rechts jeweils binäre Nullen nachgeschoben. Eine Linksverschiebung um n Stellen entspricht damit einer Multiplikation mit 2^n, eine Rechtsverschiebung um n Stellen (und Auffüllen mit binären Nullen) entsprechend einer Division durch 2^n; siehe auch die Beispiele in den Listings 1.2 und 1.3.

C++ 1.2: Bitweises Links- und Rechts-Shiften (Basics/Shift/main.cpp)

```
// b1: 101 000 ... 0000 = -1610612736: b2 = 011 00..00
int b1 = (128 + 32) * 256 * 65536 ;
int b2 = (64 + 32) * 256 * 65536 ;
cout << "b1: " << b1 << ", b2: " << b2 << endl;

// 101 ganz nach links schieben, Rest vorne auf 1 setzen : -3: 5 bei unsigned int b1
cout << "b1>>29: " << (b1 >> 29) << endl;
// 011 ganz nach links schieben, Rest vorne auf 0 setzen : 3:
cout << "b2>>29: " << (b2 >> 29) << endl;

// 101 00 ..00 um 2 Stellen rechts ergibt 100.. 00 : -2147483648
cout << "b1<<2: " << (b1 << 2) << endl;

// 101 um 28 Stellen nach links und Rest vorne auf 0 setzen: 00 .. 00 1010 = 10
// nichts Entsprechendes (Einfaches) in C++ vorhanden
```

Java 1.3: Bitweises Links- und Rechts-Shiften (Basics/Shift/Shift.java)

```
public static void main(String[] args){
 // b1: 101 000 ... 0000 = −1610612736: b2 = 011 00..00
  int b1 = (128 + 32) * 256 * 65536 ;
  int b2 = (64 + 32) * 256 * 65536 ;
  System.out.println("b1: " + b1 + ", b2: " + b2 );

  // 101 ganz nach links schieben, Rest vorne auf 1 (Vz) setzen : −3
  System.out.println("b1>>29: " + (b1 >> 29) );
  // 011 ganz nach links schieben, Rest vorne auf 0 (Vz) setzen : 3
  System.out.println("b2>>29: " + (b2 >> 29) );
  // 101 00 ..00 um 2 Stellen rechts ergibt 100.. 00 : −2147483648
  System.out.println("b1<<2: " + (b1 << 2) );

  // 101 um 28 Stellen nach links und Rest vorne auf 0 setzen: 00 .. 00 1010 = 10
  System.out.println("b1>>>a: " + (b1>>>28) );
  // 01100..0 um 28 Stellen nach links und Rest vorne auf 0 setzen: 0..0 0110 = 6
  System.out.println("b2>>>a: " + (b2>>>28) );
```

Logische Verknüpfung: `&& ||` Die Operatoren `&&` und `||` verknüpfen *logische Werte und Ausdrücke* durch ein *logisches UND* bzw. durch ein *logisches ODER*, z.B.:

```
bool okay, notOkay;
double x=44;
/* . . . */
okay = (x >= 0) && (x < 100);
notOkay = (x < 0) || (x >= 100);
if (okay && notOkay)
  cout << "Fehler!" << endl;
```

```
boolean okay, notOkay;
double x=44;
/* . . . */
okay = (x >= 0) && (x < 100);
notOkay = (x < 0) || (x >= 100);
if (okay && notOkay)
  System.out.println("Fehler!");
```

1.4.2 Unäre Operatoren

C++-Referenzierung/Dereferenzierung: `& *` Der Operator `&` liefert in C++ die Adresse im Arbeitsspeicher, unter der das durch den Operanden bezeichnete Datum abgelegt ist, entsprechend liefert der C++-Operator `*` das Datum, das unter der durch den Operanden bezeichneten Adresse abgelegt ist. Ein Zeiger ist typgebunden und wird z.B. wie folgt vereinbart:

```
double* x;
```

Das bedeutet: *Der Inhalt von x ist vom Typ* `double`*, somit ist x ein Zeiger, der auf einen Speicherbereich zeigt, in dem ein* `double`*-Wert steht.* Das folgende kleine Beispiel demonstriert den prinzipiellen Umgang mit den beiden Operatoren `&` und `*`. Ausführlicher wird das Thema in Abschn. 4.2.2 behandelt.

```
double x = 12.34;
double* adresse;
adresse = &x;
/* Ausgabe: 12.34 12.34 */
cout << x << " " << *adresse;
```

```
/*
kein direkter Umgang
mit Speicheradressen
(Zeigern) in Java
*/
```

Arithmetik: `+ -` Die beiden unären Operatoren werden als positives bzw. als negatives Vorzeichen für Zahlen verwendet.

Bitoperation: `~` Der unäre Operator ~ gehört inhaltlich zu den oben behandelten Bitoperationen; er bewirkt, dass in der binären Darstellung des entsprechenden ganzzahligen Datentyps alle Bits invertiert werden.

Logische Verknüpfung: `!` Der unäre Operator `!` gehört inhaltlich zu den oben behandelten, logischen Verknüpfungen; er invertiert einen Booleschen Ausdruck, z.B. `!((x<0) || (x>=100))` ≡ `!(x<0) && !(x>=100)` ≡ `(x>=0) && (x<100)`.

Typkonvertierung: (*type*), *type*( ), *x*`_cast<`*type*`>`( ) Siehe Abschn. 1.6.

Speicherbedarf: `sizeof` Der C++-Operator `sizeof` liefert den Speicherbedarf eines konkreten Objektes oder eines Datentyps in Byte, z.B.:

```
double x;
const int xLength = sizeof x;
const int doubleLength = sizeof (double);
```

Wenn, wie im zweiten Fall, der Operand ein Typ ist, muss er geklammert werden.

1.4.3 Postfix- und Präfix-Operatoren

Inkrementierung/Dekrementierung: `++ --` Durch Voranstellen (Präfix) oder Anhängen (Postfix) der Operatoren ++ oder -- wird der zugehörige Ausdruck jeweils um den Wert 1 erhöht bzw. erniedrigt, dabei muss der Operand einem ganzzahligen Typ angehören und ein L-Wert (siehe Seite 15) sein (d.h. eine konkrete Variable (ein konkretes Objekt) repräsentieren). Wird der Operator in einer isolierten Anweisung, wie im folgenden Beispiel, verwendet, so ist die Wirkung von Postfix- und Präfix-Operation identisch, d.h. `j++` entspricht `++j`. Entsprechendes gilt für das Erniedrigen mit --.

```
int i=0, j=0;
// i, j erhalten beide den Wert 1
i++; ++j;
```

```
int i=0, j=0;
// i, j erhalten beide den Wert 1
i++; ++j;
```

Sind Inkrement- bzw. Dekrementoperator jedoch Teil eines komplexeren Ausdrucks, in dem das Ergebnis der Operation sofort weiterverwendet wird, so hat die Entscheidung, Präfix oder Postfix zu verwenden, entscheidenden Einfluss auf das Ergebnis des Ausdrucks. Bei Verwendung der Präfix-Notation wird der Wert der Variablen erst erhöht bzw. erniedrigt und dann der Ausdruck ausgewertet. Entsprechend wird bei Postfix-Notation erst der Ausdruck ausgewertet und dann erst die Variable erhöht bzw. erniedrigt.

```
int i=4, j=4, n=4, p=4;
int k = i++; // k=4, i=5
int m = ++j; // m=5, j=5
if (n++ == 4) {/* ist true */}
if (++p == 4) {/* ist false */}
cout << i << " " << k << "\n";
cout << j << " " << m << "\n";
```

```
int i=4, j=4, n=4, p=4;
int k = i++; // k=4, i=5
int m = ++j; // m=5, j=5
if (n++ == 4) {/* ist true */}
if (++p == 4) {/* ist false */}
System.out.println(i+" "+k);
System.out.println(j+" "+m);
```

Die Ausgabe des Programmfragments ist dann:

```
5 4
5 5
```

1.4.4 Zuweisungsoperatoren

Zuweisung: `=` Der Zuweisungsoperator = weist einer Variablen einen neuen Wert zu. Er wird meist in der Form

- *lvalue* = *expression*;
  lvalue : siehe Seite 15

im Rahmen einer Anweisung verwendet. Dabei wird der Ausdruck auf der rechten Seite berechnet und dem Objekt der linken Seite zugeordnet (d.h. der Wert des Ausdrucks wird in den zum Objekt gehörenden Speicherbereich kopiert).

Verknüpfung und Zuweisung: `+= -= *= /= %= &= |= ~= <<= >>= >>>=` Diese Operatoren dienen wieder der verkürzten Schreibweise, und zwar gilt allgemein

- *lvalue* **op**= *expression* ≡
  lvalue = *lvalue* **op** *expression*;
  lvalue : siehe Seite 15

wobei **op** für einen dieser Operatoren steht.

1.5 Ausdrücke

Ausdrücke (expressions) sind Folgen von Operanden und Operatoren, die dazu dienen, Verarbeitungsvorgänge zu formulieren. Die entsprechenden Möglichkeiten in C++ und Java sind sehr vielfältig, der syntaktische Rahmen ist entsprechend komplex, deshalb beschränken wir uns an dieser Stelle auf einige Beispiele und Hinweise:

1.5.1 Beispiele

- $\frac{a-b}{a+b} \Longrightarrow$ `(a-b)/(a+b)`
- $a^2 + b^2 \Longrightarrow$ `a*a + b*b` oder `pow(a,2)+pow(b,2)`
- $x^n - y^n \Longrightarrow$ `pow(x,n) - pow(y,n)`
- $x^3 + 3x^2y + 3xy^2 + y^3 \Longrightarrow$
  `pow(x,3) + 3*pow(x,2)*y + 3*x*pow(y,2) + pow(y,3)`

- $\sqrt[2]{a^2+b^2} \Longrightarrow$ `sqrt(a*a + b*b)`
- $\sqrt[n]{\frac{x^n-y^n}{1+u^{2n}}} \equiv (\frac{x^n-y^n}{1+u^{2n}})^{\frac{1}{n}} \Longrightarrow$
  `pow( ((pow(x,n) - pow(y,n))/(1 + pow(u, 2*n)), 1.0/(double)n)`
- $\sin(x+\frac{\pi}{2}) \Longrightarrow$ `sin(x + pi/2.0)`

Die Beispiele sind sowohl in Java als auch in C++ syntaktisch korrekt. Das C++-Programm muss die Include-Anweisung `#include <cmath>` enthalten; siehe auch Abschn. 3.1. Das entsprechende Java-Programm muss ab Java Version 5.0 die Anweisung `import static java.lang.Math.*;` enthalten. In früheren Java-Versionen muss den mathematischen Funktionen noch jeweils der Klassenname `Math` vorangestellt werden, d.h. statt einfach `pow(x,n) - pow(y,n)` muss es heißen `Math.pow(x,n) - Math.pow(y,n)`.

1.5.2 Objekte und L-Werte

Ein Objekt ist ein Bereich im Arbeitsspeicher, der einem Namen zugeordnet ist, d.h. eine Variable. Ein L-Wert (lvalue = *location for a value*) ist ein Ausdruck, der sich auf ein solches Objekt bezieht, z.B. ein Variablenname. In der Zuweisung `a1 = a2` muss der linke Ausdruck a1 ein L-Wert – also z.B. ein Variablenname – sein. Entsprechendes gilt auch bei der Verwendung der übrigen Zuweisungsoperatoren `+=`, ..., `||=`.

Die folgenden Beispiele sind syntaktisch korrekt:

```
int a, b=44, *p=&a;
a = 27;  // a ist ein L−Wert
b += 27; // b ist ein L−Wert
*p++ = 27; //*p ist ein L−Wert
```

```
int a, b=44;
a = 27;  // a ist ein L−Wert
b += 27; // b ist ein L−Wert
// keine Zeiger in Java
```

Die folgenden Beispiele sind dagegen alle syntaktisch falsch:

```
b++ = 27; // b++ ist kein L−Wert !
a + b = 27; // a + b ist kein L−Wert
(p++)++; // p++ ist kein L−Wert
```

```
b++ = 27; // b++ ist kein L−Wert !
a + b = 27; // a + b ist kein L−Wert
// keine Zeiger in Java
```

1.5.3 Hinweise

- Die Reihenfolge der Auswertungen wird durch die *Prioritäten der Operatoren* und durch die *Gruppierung der Unterausdrücke* bestimmt, darüber hinaus können aber Freiheitsgrade bleiben (siehe weiter unten).
- Im Zweifelsfall und zur Erhöhung der Lesbarkeit sollten redundante Klammerpaare verwendet werden. **Tipp**
- Als Folge begrenzter Genauigkeit der Rechnerarithmetik können – auch bei gleichrangigen Operatoren – verschiedene Reihenfolgen der Auswertung zu unterschiedlichen Ergebnissen führen. Dies bedeutet, dass auch das *Kommutativ-*, *Distributiv-*

und *Assoziativ-Gesetz* nicht immer anwendbar sind (z.B. bei *Überlauf oder Unterlauf*); siehe Abschn. 1.8.

- Bei einem Ausdruck oder Unterausdruck mit gleichrangigen Operatoren ist die Reihenfolge der Auswertung in C++ undefiniert und damit implementierungsabhängig, hier können Klammern ggf. helfen, die Gruppierung – und damit die Reihenfolge der Auswertung – zu definieren. In Java dagegen ist die Auswertungsreihenfolge bei mehreren zweistelligen Operationen gleicher Priorität – außer bei Zuweisungsoperatoren – stets von links nach rechts.
- Das Verhalten bei Überlauf und Unterlauf ist implementierungsabhängig, üblicherweise wird in den Fällen sowohl in C++ als auch in Java **kein** Fehler angezeigt.
- Das Verhalten bei Division durch *Null* ist in C++ ebenfalls implementierungsabhängig, in Java wird eine Ausnahme geworfen (vgl. Abschn. **??**).
- Die Auswertung logischer Ausdrücke ist nicht kommutativ, d.h. die Auswertung einer UND-Verknüpfung wird abgebrochen, wenn der erste Operand den Wert `false` ergibt; entsprechend, wenn in einer ODER-Verknüpfung der erste Ausdruck `true` ergibt.

1.6 Explizite und implizite Typkonvertierungen

Typkonvertierungen werden in manchen Fällen automatisch (*implizit*) durchgeführt (siehe Abschn. 1.8), können aber auch *explizit* erzwungen werden, wie das folgende Beispiel veranschaulicht:

```
int m, n;
double x = 12.34;
m = x;     // implizit erlaubt
n = (int) x; // explizit
```

```
int m, n;
double x = 12.34;
m = x; // implizit verboten
n = (int) x;  // explizit
```

In beiden Fällen wird der Nachkommateil abgeschnitten, und sowohl `m` als auch `n` erhalten den Wert `12`. In diesem Fall ergibt sich somit jeweils das Gleiche. Trotzdem kann die *explizite Konvertierung* aus Gründen der Klarheit angebracht sein! Java verlangt sie hier sogar.

Es gibt aber auch viele Fälle, in denen auch in C++ eine *explizite Konvertierung* unbedingt erforderlich ist, z.B.:

```
int a = 5, b = 2;
double x, y, z;
  /* x erhaelt den Wert 2 ! */
x = a / b;
  /* y erhaelt den Wert 2.5 ! */
y = ((double) a) / ((double) b);
  /* z erhaelt den Wert 2.5 ! */
z = ((double) a) / b;
```

```
int a = 5, b = 2;
double x, y, z;
  /* x erhaelt den Wert 2 ! */
x = a / b;
  /* y erhaelt den Wert 2.5 ! */
y = ((double) a) / ((double) b);
  /* z erhaelt den Wert 2.5 ! */
z = ((double) a) / b;
```

Explizite Typkonvertierungen (**type casts**) werden in C++ und Java durch Voranstellen des *Zieltyps* entsprechend folgender Form erreicht:

Allgemeine Form: (*Zieltyp*) *Ausdruck*

In C++ ist die folgende in Java nicht zulässige Schreibweise zu bevorzugen:
Zieltyp (Ausdruck), z.B. `m = int (x);`

Empfehlenswert ist in C++ jedoch die Verwendung des `static_cast`-Operators:
`static_cast<`*Zieltyp*`>(`*Ausdruck*`)`, z.B. `m = static_cast<int>(x)`

Neben `static_cast` gibt es in C++ die Operatoren `const_cast`, `reinterpret_cast` und `dynamic_cast`. Diese C++-Typkonvertierungsoperatoren erleichtern es, den Zweck der Typkonvertierung zu erkennen, siehe hierzu Abschn. 7.3.4. Entsprechendes ist in Java nicht vorhanden. Hier gibt es lediglich die schon aus C bekannte Form der Allzweckkonvertierung `(`*Zieltyp*`)`*Ausdruck*.

***mixed mode*-Arithmetik:** Verschiedene elementare Datentypen können in C++ und in Java in Ausdrücken gemischt werden. Bei deren Auswertung werden dann implizit Konvertierungen nach bestimmten Regeln durchgeführt, die hier im Detail nicht erörtert werden. Als grober Anhaltspunkt gilt:

- Bei der Verknüpfung zweier Operanden unterschiedlichen Typs wird der einfachere in den umfassenderen umgewandelt.

Implizite und explizite Typkonvertierungen werden, wie oben gezeigt, auch dann durchgeführt, wenn dabei Information verloren geht, insbesondere ist in C++ auch die Konvertierung zwischen `signed`- und `unsigned`-Typen sehr fehleranfällig. Sie sollten daher vermieden werden.

Tipp

Standardmäßig sollten die Typen `int` für ganze Zahlen und `double` für Gleitkommazahlen verwendet werden, um unnötige (unbeabsichtigte) Konvertierungen zu vermeiden.

1.7 Prioritäten von Operatoren

Bezüglich der Prioritäten aller Operatoren gilt die in Tabelle 1.5 angegebene Reihenfolge. Operatoren, die nur in C++ oder Java erlaubt sind, sind in den Tabellen jeweils grau unterlegt.

Die höchste Priorität hat der in Zeile (0) aufgeführte Operator, die zweithöchste die in Zeile (1) notierten Operatoren. Die drittniedrigste Priorität haben die in Zeile (15) angegebenen Zuweisungsoperatoren. Die Abstufungen sind mit Bedacht so festgelegt worden, damit in vielen Fällen auf Klammerungen verzichtet werden kann. Die folgenden Beispiele zeigen äquivalente Ausdrücke mit und ohne zusätzliche Klammern. Allerdings erhöht die Verwendung von Klammern häufig die Les- und Wartbarkeit ganz erheblich.

ohne zus. Klammern

```
int a, b, c, d, y1, y2, y3;
y1 = a < b && !(c == d);
```

mit zus. Klammern

```
int a, b, c, d, y1, y2, y3;
y1 = (a < b) && (!(c == d));
```

```
y2 = a & ~b | ~(c | d);          y2 = (a & (~b)) | (~(c | d));
y3 = a << b & c >> a;            y3 = (a << b) & (c >> a);
```

Tabelle 1.5: Prioritäten von Operatoren für elementare Datentypen

Priorität	Operatoren	Bemerkung
0	`::`	C++-Bereichsauflösung

1	`++ -- () [] -> .`
	`static_cast, dynamic_cast const_cast`
	`reinterpret_cast`
		postfix, ...,
		C++-Typumwandlung
		C++-Typumwandlung

2	`++ -- ! ~ + -` (*type*) `new new[]`
	`* & sizeof` *type*()
	`delete delete[]`
		präfix, unär
		C++-Zeiger, Typen
		C++-Heapverwaltung

3	`. ->`	Elementselektion
4 5	`* / %` `+ -`	binäre arithmetische Operatoren
6	`<< >>` `>>>`	Shift-Operatoren Java-Shift-Operator
7 8	`< <= > >=` `== !=`	Vergleichs-operatoren

9
10
11
	`&`
	`~`
	`|`
		Bitoperationen

12 13	`&&` `||`	Logische Verknüpfungen
14	`? :`	bedingte Zuweisung
15	`= += -= *= /= %= &= ~= |= <<= >>=` `>>>=`	Zuweisungsoperatoren Java-Zuweisungsoperator
16	`throw`	C++-Ausnahme erzeugen
17	`,`	Komma-Operator

1.8 Arbeiten mit Zahlen

1.8.1 Ausdrücke und Zuweisungen mit gemischten Zahlentypen

In Java und C++ können grundsätzlich in einem arithmetischen Ausdruck verschiedene Zahlentypen vermischt benutzt werden, z.B.

```
int ix = 27;
long lx = 123;
float fx = 53.6f;
double dx = 76.2, dy;
dy = 1e5 * ( (ix + dx - fx) / 2 );
```

```
int ix = 27;
long lx = 123;
float fx = 53.6f;
double dx = 76.2, dy;
dy = 1e5 * ( (ix + dx - fx) / 2 );
```

Bei der Verknüpfung zweier Zahlen unterschiedlichen Typs wird immer implizit in den umfassenderen Typ konvertiert, d.h. die Anweisungen

```
double dx = 5.0 / 2.0;
dx = 5.0 / static_cast<double>(2);
dx = 5.0 / 2;
dx = 5 / 2.0;
```

```
double dx = 5.0 / 2.0;
dx = 5.0 / (double) 2;
dx = 5.0 / 2;
dx = 5 / 2.0;
```

führen zum gleichen Ergebnis von `2.5`.

Im Gegensatz dazu erhält dx in beiden folgenden Fällen

```
dx = 5 / 2;
dy = (double) (5 / 2);
```

```
dx = 5 / 2;
dy = (double) (5 / 2);
```

den Wert `2.0`!

Einen weiteren Hinweis für Probleme bei der Konvertierung liefert das folgende zunächst harmlos erscheinende Beispiel. In der Definition `float f = 0.6;` findet nämlich eine implizite Konvertierung statt, bei der man auch eine Compiler-Warnung erhält, denn `0.6` ist eine Konstante des Typs `double` und – sofern `sizeof(double)` und `sizeof(float)` verschieden sind – nicht ohne Genauigkeitsverlust konvertierbar. Warum?

Eine Gleitkommazahl wird intern durch Mantisse und Exponent dargestellt. Für `0.6` ergibt sich dann die Mantissendarstellung `1001 1001 1001 ... 1001 ...`, denn $0.6 = 1 * \frac{1}{2} + 0 * \frac{1}{4} + 0 * \frac{1}{8} + 1 * \frac{1}{16} + 1 * \frac{1}{32} + 0 * \frac{1}{64} + 0 * \frac{1}{128} + 1 * \frac{1}{256} + ...$, was nicht exakt 0.6, sondern 0,5999... ergibt. Die Länge der Mantisse des Datentyps `float` ist in der Regel 24 Bit und die von `double` meist mehr als 40 Bit, sodass sich bei der Konvertierung von 0.6 als *double-Konstante* zu 0.6 als *float-Konstante* ein Genauigkeitsverlust ergibt. Um im obigen Fall Compiler-Warnungen zu vermeiden, ist daher die Definition `float f = 0.6f;` erforderlich.

Bei der Umwandlung von einem Typ ohne Vorzeichen in einen Typ mit Vorzeichen kann in C++ unerwartet ein Vorzeichenwechsel auftreten:

```
unsigned int ux = UINT_MAX;
int ix = ux; /*Ergebnis z.B.: -1 !*/
for (unsigned int j = 2; j >= 0; --j) {
  cout << j << " ";// Endlosschleife
}
```

```
/*
Entsprechende Probleme in
Java unbekannt, da es vor-
zeichenlose Typen nicht gibt
*/
```

Im Beispiel gibt es eine Endlosschleife, weil der Wert `-1` von `j` immer wieder in eine vorzeichenlose Zahl, z.B. $2^{32} - 1$, gewandelt wird.

Das Regelwerk für die Konvertierung von Zahlentypen ist im Detail recht umfangreich. Es ist sicher auch nicht sinnvoll, alle Möglichkeiten auszuschöpfen. Die wesentlichen Grundregeln sind:

- C++ erlaubt beliebige explizite und implizite Konvertierungen zwischen allen Character-, Integer- und Gleitkommatypen.
- Bei der Umwandlung eines Gleitkomma- in einen Ganzzahltyp wird der Nachkommateil abgeschnitten (truncate).
- Bei der Verknüpfung zweier Operanden unterschiedlichen Typs wird der einfachere in den umfassenderen Typ umgewandelt.
- Das Ergebnis der Umwandlung eines Wertes eines bestimmten Typs in einen Wert eines weniger mächtigen oder weniger genauen Typs kann undefiniert sein (z.B. $long \Rightarrow int$, aber auch $long \Rightarrow float$).
- Die Umwandlung zwischen *signed*- und *unsigned*-Typen kann wegen der unterschiedlichen Definitionsbereiche in C++ leicht zu unerwarteten Ergebnissen führen.

1.8.2 Einschränkungen gegenüber der Mathematik

In der Mathematik sind Zahlen (reelle und auch ganzzahlige) Elemente unendlicher Mengen. Die Darstellung von Zahlen im Rechner ermöglicht natürlich nur endliche Mengen, und das hat Konsequenzen, die man vielleicht auf den ersten Blick nicht erwartet. Es betrifft neben dem Aspekt *Rechengenauigkeit* z.B. auch die beschränkte Gültigkeit mathematischer Axiome beim Rechnen mit *Computerzahlen*:

- Die ganzzahligen Typen (`int, long, ...`) repräsentieren nur Annäherungen an die ganzen bzw. an die natürlichen Zahlen der Mathematik, da ihr **Bereich eingeschränkt** ist!
- Die Gleitkommatypen (`float, double` und `long double`) repräsentieren nur Annäherungen an die reellen Zahlen aus der Mathematik, da ihr **Bereich eingeschränkt und** zusätzlich ihre **Genauigkeit begrenzt** ist.

Das demonstrieren folgende Beispiele:

```
short a, b, c;
a = 20000; b = 20000;
c = a + b; // c != 40000
cout << c << endl;
```

```
short a, b, c;
a = 20000; b = 20000;
c = (short)(a + b); // c != 40000
System.out.println(c);
```

`c` hat in keinem Fall den Wert `40000`, weil das Ergebnis größer ist als `32767`, der größte, darstellbare `short`-Wert. Es ergibt sich ein Überlauf. Die Ausgabe ist jeweils `-25536`. Für C++ gelten diese Aussagen allerdings nur, wenn `short` zwei Byte belegt.

```
float x, y;
x = 1e20f;
y = x + 1; // Die 1 geht verloren wegen
        // mangelnder Genauigkeit
if (y-x != 1) { // ist true
  cout << "ungleich\n";
}
```

```
float x, y;
x = 1e20f;
y = x + 1; // Die 1 geht verloren wegen
        // mangelnder Genauigkeit
if (y-x != 1) { // ist true
  System.out.println("ungleich");
  }
```

Die Ausgabe ist jeweils das Wort *ungleich*.

Weiterhin folgt aus diesen Begrenzungen, dass bei der Formulierung von Ausdrücken die Axiome der Mathematik – z.B. die *Kommutativgesetze*, die *Assoziativgesetze* und die *Distributivgesetze* – nur mit Einschränkungen gelten. Die im Folgenden verwendeten Symbole $\oplus$ und $\otimes$ sollen hervorheben, dass nicht die mathematischen Operationen *plus* bzw. *mal*, sondern die im Rechner implementierten gemeint sind:

$$\begin{aligned} x \oplus y \oplus z &\neq x \oplus z \oplus y \\ x \otimes y \otimes z &\neq x \otimes z \otimes y \\ (x \oplus y) \oplus z &\neq x \oplus (y \oplus z) \\ (x \otimes y) \oplus (x \otimes z) &\neq x \otimes (y \oplus z) \end{aligned}$$

1.8.3 Aufzählungstypen

Häufig werden ganze Zahlen verwendet, um Begriffe zu codieren, z.B. Codierung für Bewertungen (*1: sehr gut, 2: gut, ..., 6: ungenügend*), Codierung für Monatsnamen (*1: Januar, ..., 12: Dezember*) oder Codierung für Farben (*0: rot, 1: gelb, 2: blau*). Die Aufzählungstypen bieten umgekehrt eine einfache Möglichkeit, im Programm anstelle der Zahlenkonstanten aussagekräftige Namen zu verwenden.

```
enum Farbe {rot, gelb, blau};
```

```
enum Farbe {rot, gelb, blau};
```

Diese Zeilen haben die gleiche Wirkung wie die Vereinbarung von den drei Konstanten mit `const` bzw. `final`:

```
const int rot = 0;
const int gelb = 1;
const int blau = 2;
```

```
final int rot = 0;
final int gelb = 1;
final int blau = 2;
```

Zusätzlich werden die drei neuen Konstanten mit der `enum`-Anweisung allerdings unter einem neu definierten Typen `Farbe` zusammengefasst. In C++ hätte Farbe allerdings auch weggelassen werden können (`enum rot, gelb, blau;`), wobei dann aber nur die drei C++-Konstanten ohne den Typ `Farbe` vereinbart sind.

Allgemeine Formen:

enum { *enumerator-list* }	nur C++
enum *identifier* { *enumerator-list* }	C++ und Java
enum *identifier*	nur C++

Mit *Aufzählungstypen* ist es möglich, eine Vielzahl gleichartiger ganzzahliger Konstanten in einer Menge, d.h. im Aufzählungstyp, zusammenzufassen. In C++ bekommt jedes Element dieser Menge eine Ordinalzahl zugeordnet. Man kann wie im folgenden Beispiel einige oder alle Elemente des Aufzählungstyps auch explizit initialisieren. Zur Initialisierung können beliebige (in diesem Aufzählungstyp oder anderswo) zuvor definierte Konstanten verwendet werden.

```
enum Skat {Ass=11, Bube=2, Dame,Koenig};
```

Wird hinter einem Element kein Wert wie bei `Dame` oder `Koenig` angegeben, erhält dieses den Wert des Vorgängers erhöht um 1. Es ist auch möglich, mehreren Bezeichnern den gleichen Wert zuzuordnen. Wie oben bereits erwähnt, muss auch in C++ hinter dem Schlüsselwort `enum` ein Typname angegeben werden, damit Variablen des Aufzählungstyps vereinbart werden können.

```
Farbe f1, f2, f3, f4;
f1 = rot; f2 = f1;
f3 = static_cast<Farbe>(2); // blau
switch (f4) {
  case rot : /*... */; break;
  case gelb : /*... */; break;
  case blau : /*... */;
}
```

```
Farbe f1, f2, f3, f4;
f1 = Farbe.rot; f2 = f1;
// f3 = (Farbe)(2);
switch (f1) {
  case rot : /*... */; break;
  case gelb : /*... */; break;
  case blau : /*... */;
}
```

In Java führt der Konvertierungsversuch der `int`-Konstanten `2` in eine Konstante des Typs `Farbe` zu einem Syntaxfehler. Aufzählungstypen sind eigenständige Typen und werden nicht intern als Ordinalzahl codiert. Die Syntax für Aufzählungen in Java ist umfangreich, man kann jedem Aufzählungselement beliebige Eigenschaften zuordnen, die abgefragt und nach denen unterschieden werden kann.

1.9 Eingabe und Ausgabe von Daten

Wir können nun mit einfachen Datentypen umgehen. In diesem Abschnitt geht es darum, wie man Daten auf dem Bildschirm ausgeben kann bzw. über die Tastatur einliest. Hier werden wir nur einige Grundlagen, die zum Verstehen der folgenden Abschnitte erforderlich sind, behandeln.

1.9.1 Ein- und Ausgabe im Standardmodus mit Standardgeräten

In Java und in C++ wird die Ein- und Ausgabe mit Hilfe von **Strömen** (`streams`) durchgeführt. Ein Strom stellt die Schnittstelle des Programms nach außen dar und ist eine geordnete Folge von Daten, die eine Quelle oder eine Senke haben.

Für die Ein- und Ausgabe auf Standardgeräten sind in C++ die Ströme (Objekte) `cin` und `cout` vordefiniert, die in der Regel die Tastatur bzw. den Bildschirm repräsentieren. Für die Ausgabe von Fehlern gibt es den Strom (das Objekt) `cerr`, der ebenfalls standardmäßig dem Bildschirm zugeordnet ist. Entsprechend gibt es Voreinstellungen für die Formatierung, sodass in vielen Fällen auf eine explizite Formatierung verzichtet werden kann. Als Operatoren sind die Symbole >> und << zur Ein- bzw. Ausgabe überladen worden, die ansonsten für Shift-Operationen verwendet werden.

Java stellt für die normale Ausgabe den Strom `System.out` und für die Fehlerausgabe `System.err` zur Verfügung. Die Tastatureingabe erfolgt standardmäßig über den Strom `System.in`.

Ein einfaches Beispiel mit der Ein- und Ausgabe von `double`-Werten kann dann z.B. wie im Listing 1.4 aussehen.

C++/Java 1.4: Ein- und Ausgabe über Tastatur und Bildschirm

```cpp
// Einbinden der Ströme cin,
// cout, und cerr
#include<iostream>

using namespace std;


int main() {



  double x, y;
  // Ausgabe von "x: " auf Bildschirm
  cout << "x: ";
  // Eingabe von x ueber Tastatur
  cin >> x;

  cout << "y: ";
  cin >> y;
  // Ausgabe von x + y
  cout << "x + y = " << x+y;
  // Zeilenumbruch auf Bildschirm
  cout << endl;
}
```

(Basics/EA/EAScan.cpp)

```java
// Verbinden von System.in mit
// vereinfachter Eingabe
import java.util.Scanner;
// Einbinden der EA-Objekte
import java.io.*;

public class EAScan {
public static void main(String[] args){
  // Verbinden der Tastatur mit
  // vereinfachter Eingabe
  Scanner keyb = new Scanner(System.in);
  double x, y;
  // Ausgabe von "x: " auf Bildschirm
  System.out.print("x: ");
  // Eingabe von x ueber Tastatur
  x = keyb.nextDouble();

  System.out.print("y: ");
  y = keyb.nextDouble();
  // Ausgabe von x + y
  System.out.print("x + y = " + (x+y));
  // Zeilenumbruch auf Bildschirm
  System.out.println();
}}
```

(Basics/EA/EAScan.java)

In C++ können (fast) alle Typen mit den Operatoren >> und << ein- und ausgegeben werden. In Java stehen für die Standardtypen geeignete Funktionen (in Java *Methoden* genannt) `nextInt`, `nextDouble`, `nextBoolean` etc. zur Verfügung.

1.9.2 Arbeiten mit Dateien

Das Arbeiten mit sequenziellen Textdateien wird genauso abgewickelt wie das Arbeiten mit Tastatur und Bildschirm, es müssen nur zusätzlich die betroffenen Dateien geöffnet und wieder geschlossen werden. Die folgende kurze Funktion demonstriert das Kopieren einer Datei in C++:

```
#include <fstream> // Einbinden der Dateifunktionen
using namespace std;
void FileCopy(char* name1, char* name2) {
  ifstream ein(name1);
  ofstream aus(name2);
  char ch;
  if (!ein || ! aus) {
    cerr << "Fehler beim Oeffnen der Dateien"; exit(1);
  }
  while (ein.get(ch)) aus.put(ch);
}
```

Das Öffnen (und implizit auch das Schließen) der Datei erfolgt durch das Anlegen der Strom-Variablen `ein` und `aus`.

In Java werden anstatt `ifstream` und `ofstream` die Typen `FileInputStream` und `FileOutputStream` verwendet. Wenn diese – wie im Listing 1.5 gezeigt – mit `Scanner` verknüpft werden, stehen auch für die Dateiein- und ausgabe die Methoden `nextInt`, `nextDouble`, `nextBoolean` etc. sowie `print` `println` etc. zur Verfügung.

1.10 Steueranweisungen

1.10.1 Übersicht

Anweisungen in Programmen dienen dazu, Algorithmen zu formulieren. Algorithmen wiederum sind Verarbeitungsvorschriften für die im Programm vereinbarten Daten. Neben den im vorigen Abschnitt genannten *einfachen Anweisungen*, die man durch Abschließen eines Ausdrucks mit einem Semikolon erhält, gibt es die *Steueranweisungen*. Das Konzept der Steueranweisungen ist dem vieler anderer höherer Programmiersprachen, wie z.B. *Pascal, Modula, Oberon und Ada*, sehr ähnlich, und zwar gibt es in C++ und Java folgende Arten von Steueranweisungen:

- Verbundanweisungen (*compound-statements*)
- Verzweigungs-Anweisungen (*selection statements*)
  - **if**-Anweisung
  - **if**/**else**-Anweisung
  - **switch**-Anweisung
- Wiederholungs-Anweisungen (*iteration statements*)
  - **while**-Anweisung
  - **do**/**while**-Anweisung
  - **for**-Anweisung

C++/Java 1.5: Ein- und Ausgabe über Datei

```cpp
 // Einbinden der Stroeme cin,
 // cout, und cerr
#include <fstream>

using namespace std;
void fileIO(){
  ifstream in("input.txt");
  ofstream out("output.txt");



  double x; int i;
  in >> x >> i;
  out << "x= " << x;
  out << "i= " << i << endl;
  in.close();
  out.close();
}
```

(Basics/EA/FileIO.cpp)

```java
import java.util.Scanner;
import java.io.*;

public class FileIO {
public static void main(String[] args){
  try {
  FileInputStream fileIn =
    new FileInputStream("input.txt");
  Scanner in = new Scanner(fileIn);
  PrintStream out =
       new PrintStream("output.txt");

  double x; int i;
  x = in.nextDouble();
  i = in.nextInt();

  out.print("x= " + x);
  out.println("i= " + i);
  in.close();
  out.close();
  }
  catch (IOException e) {
    System.out.println("Fehler: "+e);
  }
}
}
```

(Basics/EA/FileIO.java)

- Sprung-Anweisungen (*jump statements*) und markierte Anweisungen (*labeled statements*)
  - **goto-** (nur C++), **continue-**, **break-**, **return**-Anweisung
  - Anweisung mit Marke (nur C++), **case-**, **default**-Anweisung

Die Sprunganweisungen dienen im Wesentlichen dazu, Wiederholungen vorzeitig abzubrechen, die markierten Anweisungen `case` und `default` werden in der `switch`-Anweisung (*Vielfachverzweigung*) benötigt. Die `goto`-Anweisung und die Anweisung mit Marke gehören zusammen und können als Relikt der klassischen *Spaghetti-Programmierung* angesehen werden. Das Wort ist in Java zwar reserviert, es gibt aber keine `goto`-Anweisung in Java.

Das Syntaxdiagramm in Bild 1.1 gibt eine teilweise leicht vereinfachte Übersicht über die wichtigsten Steueranweisungen. Die abgerundeten Kästchen repräsentieren die Endsymbole, die explizit in der Anweisung auftreten, die eckigen Kästchen enthalten als Kurzbezeichnungen die Buchstaben *d* für *declaration*, *s* für *statement* und *e* für *expression*. Korrekte Formen für Anweisungen ergeben sich aus den Syntaxdiagrammen, indem sie in Pfeilrichtung durchlaufen werden. Bögen in Vorwärtsrichtung von links nach rechts definieren die Möglichkeiten, Teile zu umgehen, Bögen in Rückwärtsrichtung definieren entsprechend Wiederholungen. Im Fall der Verbundanweisung können z.B. *declaration* und *statement* ganz fehlen oder auch jeweils ein- oder mehrfach vorkommen.

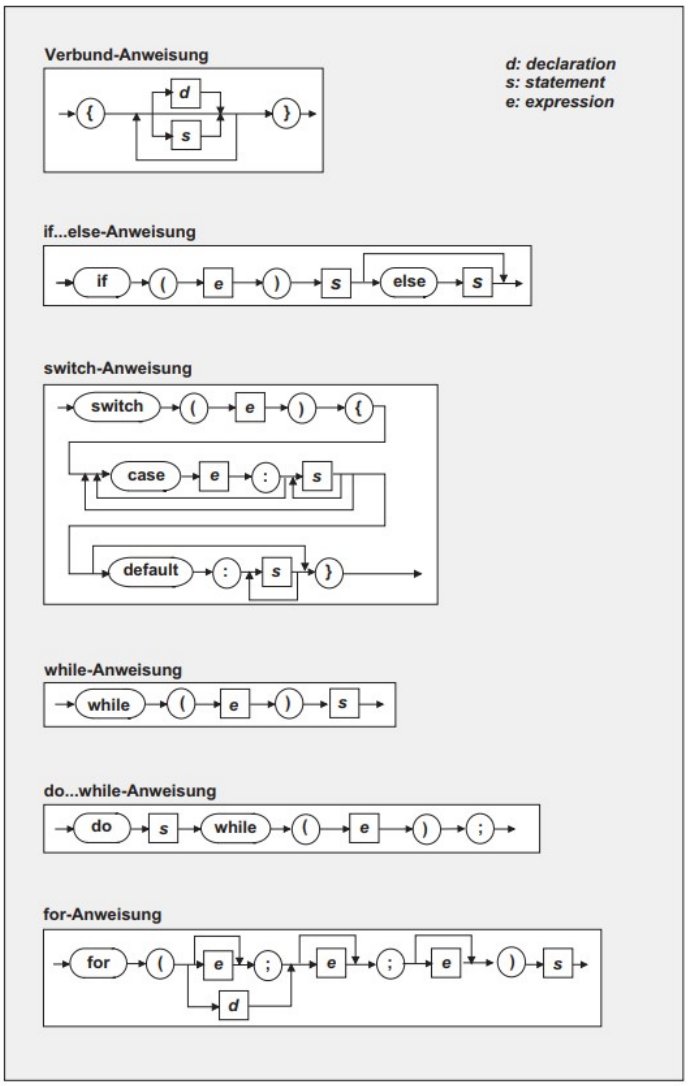

Bild 1.1: C++- und Java-Syntaxdiagramme der Steueranweisungen

1.10.2 Verbundanweisung (compound-statement)

Beispiel:

```
{
  double buffer;
  buffer = x; x = y; y = buffer;
  double tmp = x*x; y = tmp*tmp;
}
```

```
{
  double buffer;
  buffer = x; x = y; y = buffer;
  double tmp = x*x; y = tmp*tmp;
}
```

Allgemeine Form: { *statement-seq*$_{opt}$ }

Bedeutung: Eine Verbundanweisung ist eine Folge von mehreren Anweisungen in geschweiften Klammern. Sie dient dazu, mehrere Anweisungen (*statement-seq*) zu einer einzigen zusammenzufassen. Definitionen und *normale* Anweisungen dürfen sich beliebig (auch mehrfach) abwechseln, d.h. formal ist jede Definition ebenfalls eine Anweisung. Variablen, die innerhalb einer Verbundanweisung definiert wurden, existieren nur innerhalb der Verbundanweisung.

1.10.3 Verzweigungen

if-Anweisung

Beispiel:

```
if (x < y) {
  double buffer;
  buffer = x;
  x = y;
  y = buffer;
}
```

```
if (x < y) {
  double buffer;
  buffer = x;
  x = y;
  y = buffer;
}
```

Allgemeine Form: **if** (*expression*) *statement*

Bedeutung: Der Ausdruck (*expression*) wird ausgewertet. Wird das Ergebnis als logisch `true` interpretiert, dann wird die Anweisung (*statement*) ausgeführt, sonst nicht – d.h. wenn die Auswertung den Wert `false` liefert.

In Java muss der Ausdruck (*expression*) vom Typ `boolean` sein, in C++ nicht unbedingt. Das eröffnet eine beliebte Fehlerquelle:

```
int a=2, b=1;
if (a=b) { ... } // Zuweisung
```

```
int a=2, b=1;
if (a==b) { ... } // Vergleich
```

Beide Beispiele werden vom C++-Compiler akzeptiert, obwohl es sich im linken Fall um eine *Zuweisung* und keinen *Vergleich* handelt. Das Ergebnis der Zuweisung ist in diesem Fall der Wert von `a`, der in C++ dann als Boolescher Wert interpretiert wird. Der Java Compiler akzeptiert nur die rechte Variante mit einem Vergleich in der Bedingung.

if/else-Anweisung

Beispiel:

```
if (x < y) {
  min = x;
}
else {
  min = y;
}
```

```
if (x < y) {
  min = x;
}
else {
  min = y;
}
```

Allgemeine Form: **if** (*expression*) *statement* **else** *statement*

Bedeutung: Wenn die Auswertung des Ausdrucks (*expression*) `true` ergibt, dann wird die auf *expression* folgende Anweisung (*then-Teil*) ausgeführt, sonst wird die auf `else` folgende Anweisung (*else-Teil*) durchgeführt.

Tipp Im obigen Beispiel hätten die geschweiften Klammern sowohl im *then-Teil* als auch im *else-Teil* weggelassen werden können, weil beide Teile jeweils nur eine Anweisung enthalten. Grundsätzlich empfiehlt es sich jedoch, immer geschweifte Klammern zu verwenden, d.h. sowohl im *then-* wie auch im *else-Teil* wird eine Verbundanweisung (siehe Abschn. 1.10.2) verwendet.

Die Anweisung im *else*-Teil einer *if/else-Anweisung* kann natürlich auch wieder eine *if/else-Anweisung* sein, was dann zu Mehrfachverzweigungen wie im nächsten Beispiel führt. Es zeigt auch, wie man durch logische Verknüpfungen im Ausdruck (*expression*) die Anzahl der Verzweigungen reduziert.

Beispiel:

```
int art;
string klasse;

if ( ch < ' ' || ch == 127 /*!*/ ) {
  klasse = "Steuerzeichen"; art=0;
}
else if ( ch >= '0' && ch <= '9' ) {
  klasse = "Ziffer";   art=2;
}
else if ( ch >= 'A' && ch <= 'Z' ||
       ch >= 'a' && ch <= 'z' ) {
  klasse = "Buchstabe"; art=3;
}
else {
  klasse = "Sonderzeichen"; art=1;
}
```

```
int art;
String klasse;

if ( ch < ' ' || ch == 127 /*!*/ ) {
  klasse = "Steuerzeichen"; art=0;
}
else if ( ch >= '0' && ch <= '9' ) {
  klasse = "Ziffer";   art=2;
}
else if ( ch >= 'A' && ch <= 'Z' ||
       ch >= 'a' && ch <= 'z' ) {
  klasse = "Buchstabe"; art=3;
}
else {
  klasse = "Sonderzeichen"; art=1;
}
```

switch-Anweisung

Beispiel:

```
switch (art) {
  case 0 : /*... */ break;
  case 1 : /*... */;
  case 2 : /*... */; break;
  case 3 : break;
  case 4 : case 9: /*... */ break;
  default : cout << "Fehler\n";
}
```

```
switch (art) {
  case 0 : /*... */ break;
  case 1 : /*... */;
  case 2 : /*... */; break;
  case 3 : break;
  case 4 : case 9: /*... */ break;
  default:System.out.println("Fehler");
}
```

Allgemeine Form:

switch (*expression*) *statement*

Übliche Verwendung:

switch (*expression*) **{**
  case *constant expression*: *statements*
  . . .
  case *constant expression*: *statements*
  default: *statements*
}

Bedeutung: Die *switch-Anweisung* ist speziell zur Formulierung von Mehrfachverzweigungen vorgesehen. Die allgemeine Syntax lässt viele Konstrukte zu, die übliche Verwendung enthält aber im Rumpf – dem Beispiel und der angegebenen eingeschränkten Form entsprechend – nur Anweisungen oder Anweisungsfolgen, die jeweils durch eine *case-Marke* eingeleitet werden, sowie maximal eine Anweisung oder Anweisungsfolge, die durch eine *default-Marke* eingeleitet wird. Es sollte – aus Gründen der guten Lesbarkeit – nur diese eingeschränkte Form verwendet werden!

Der Ausdruck wird ausgewertet, und es wird geprüft, ob eine dem errechneten Wert entsprechende Sprungmarke vorhanden ist. Ist das der Fall, wird dorthin verzweigt, wenn nicht, wird – falls vorhanden – zum *default-Teil* verzweigt. Wenn weder eine entsprechende Sprungmarke explizit angegeben noch ein *default-Teil* vorhanden ist, wird keine Anweisung durchgeführt!

Die *case-Teile* werden üblicherweise mit einer *break-Anweisung* abgeschlossen, die die *switch-Anweisung* an der Stelle beenden. Wenn das `break` entfällt, wird sequenziell mit dem folgenden *case-Teil* oder mit dem *default-Teil* weitergemacht. Statt mit einer *break-Anweisung* kann in Funktionen die *switch-Anweisung* auch mit einer *return-Anweisung* beendet werden. Der Ausdruck (*expression*) muss in allen Anweisungen einen ganzzahligen Typ liefern (char, int, ... oder enum). Die auf `case` folgenden Marken müssen ganzzahlige Konstanten sein. Mit der C++-*switch-Anweisung* kann die *if/else-Anweisung* simuliert werden, z.B.:

```
switch ( static_cast<int>(i < j) ) {
  case 1: cout << "kleiner\n"; break;
  case 0: cout << "nicht kleiner\n";
}
```

```
// In Java keine Konvertierung von
// boolean nach int definiert.
```

1.10.4 Wiederholungen

while-Anweisung

Beispiel:

```
int sum = 0, count = 1;
while (count <= 100) {
  sum += count; count++;
}
```

```
int sum = 0, count = 1;
while (count <= 100) {
  sum += count; count++;
}
```

Allgemeine Form: **while** (*expression*) *statement*

Bedeutung: Die Anweisung wird so lange wiederholt, wie die Auswertung des Ausdrucks `true` ergibt. Der Ausdruck wird bei jedem Schleifendurchlauf neu berechnet. Hat er bereits zu Beginn den Wert `false`, so wird die Anweisung gar nicht ausgeführt. Die Anzahl der Schleifendurchläufe kann somit auch *Null* sein!

Dementsprechend ist `while (true)...` eine Endlosschleife (repeat forever)!

do/while-Anweisung

Beispiel:

```
int sum = 0, count = 1;
do {
  sum += count; count++;
}
while (count <= 101);
```

```
int sum = 0, count = 1;
do {
  sum += count; count++;
}
while (count <= 101);
```

Allgemeine Form: **do** *statement* **while** (*expression*)**;**

Bedeutung: Die Anweisung wird so lange wiederholt, bis die Auswertung des Ausdrucks den Wert `false` ergibt. Im Unterschied zur `while`-Anweisung erfolgt die Auswertung nicht zu Beginn, sondern zum Schluss, und somit wird die Anweisung mindestens einmal ausgeführt.

Dementsprechend ist `do...while (true);` ebenfalls eine Endlosschleife.

for-Anweisung

Beispiel 1:

```
int sum = 0;
for (int count=1; count<101; ++count)
  sum += count;
```

```
int sum = 0;
for (int count=1; count<101; ++count)
  sum += count;
```

Beispiel 2:

```
int sum = 0;
for (int i=1, j=100; j>50; ++i,--j){
  sum = sum + i + j;
}
```

```
int sum = 0;
for (int i=1, j=100; j>50; ++i,--j){
  sum = sum + i + j;
}
```

Allgemeine Form: **for** (*for-init-statement*$_{opt}$ *expression*$_{opt}$*;* *expression*$_{opt}$) *statement*

Bedeutung: Im *for-init-statement* wird die Initialisierung spezifiziert. Im ersten Ausdruck *expression* ist entsprechend die Bedingung angegeben, die erfüllt sein muss, damit eine (weitere) Iteration der Anweisung(en) durchgeführt wird. Im zweiten Ausdruck wird eine bei jeder Iteration durchgeführte Operation spezifiziert.

Tipp

Im Gegensatz zur Schreibweise im Beispiel 1 sollten die Anweisungen der `for-Schleife` wie im Beispiel 2 immer in geschweiften Klammern gesetzt werden, selbst dann, wenn nur eine einzige Anweisung folgt. Ein oft nicht leicht zu entdeckender Fehler bei Verwendung der `for-Schleife` ist ein Semikolon vor der Anweisung:

```
for (i = 0; i < 10; ++i) ;
  summe += i;
```

Dies entspricht allerdings:

```
for (i = 0; i < 10; ++i) {}
summe += i;
```

und somit wahrscheinlich nicht dem Gewünschtem, was ein weiterer Grund dafür ist, den *statement*-Teil der `for`-Schleife auch dann zu klammern, wenn dieser lediglich aus einer leeren Anweisung besteht.

Wie das Beispiel 2 demonstriert, können die Ausdrücke in der *for-Anweisung* sich aus jeweils mehreren durch Kommata getrennten Teilausdrücken zusammensetzen. `for (;1;) ...` und `for (;;) ...` sind Formulierungen für Endlosschleifen.

Die Laufvariable einer for-Anweisung kann auch innerhalb dieser Anweisung vereinbart werden, z.B.:

```
for (int i = 0; i < 100; ++i) {...}
```

Die Gültigkeit dieser Definition von `i` erstreckt sich auf die zur for-Schleife gehörende Verbundanweisung. Nach der `for`-Schleife ist `i` nicht mehr definiert. Es ist somit möglich, für alle unabhängigen Schleifen die gleiche Laufvariable (z.B. i) zu verwenden.

Bezüglich der Entscheidung, welche der drei Anweisungen `while`, `do`/`while` oder `for` jeweils zur Formulierung einer bestimmten Iteration zu verwenden ist, gibt es unterschiedliche Ansichten. Puristen verwenden stets die *sichere while-Anweisung*, typische C-Programmierer alter Schule haben eine ausgeprägte Vorliebe für die *mächtige for-Anweisung*. Sinnvoll ist allerdings eine differenzierte Auswahl:

Tipp

Verwendung der *while-Anweisung*, wenn die Anzahl der Iterationen n unbekannt ist, mit $n \geq 0$ (auch *null* Durchläufe möglich!), Verwendung der *do/while-Anweisung*,

wenn im Gegensatz dazu $n \geq 1$ (mindestens ein Durchlauf!), und Bevorzugung der *for-Schleife* insbesondere dann, wenn die Anzahl der Iterationen schon bekannt ist.

Tipp In der Bedingung einer for-Schleife sollte, wenn möglich, immer der Kleiner- bzw. Größer-Operator (anstatt <= bzw. >=) verwendet werden, d.h. es wird ein halboffenes Intervall verwendet. Die Anzahl der Schleifendurchläufe kann dann sehr einfach immer auf die gleiche Weise durch Differenzbildung von Schleifenendwert und -anfangswert ermittelt werden.

Besondere Aufmerksamkeit ist bei Verwendung der do-Schleife geboten. Empirische Untersuchungen der Universität Münster [27] haben gezeigt, dass bei Verwendung der do-Schleife etwa 50% mehr Fehler auftreten als bei Verwendung der while-Schleife. Hieraus sollte aber keineswegs der Schluss gezogen werden, die do-Schleife zu meiden, sondern bei ihrer Benutzung große Sorgfalt auf die Festlegung von Schleifenanfangs- und Schleifenendwert zu verwenden.

1.10.5 Sprung-Anweisungen und markierte Anweisungen

Die folgenden sieben Sprung- und markierten Anweisungen – mit der jeweils angegebenen Syntax – bieten weitere Möglichkeiten in C++ und in Java, den Kontrollfluss zu beeinflussen:

(1) **goto** *identifier*; (nur C++)

(2) **break;**

(3) **continue;**

(4) **return** $expression_{opt}$;

(5) *identifier*: *statement*

(6) **case** *const-expression*: *statement*

(7) **default:** *statement*

Mit der **goto**-Anweisung (1) können Sprünge auf Anweisungen mit Marken (5) durchgeführt werden – ein Relikt aus der Zeit der *Spaghetti*-Programmierung früherer Zeiten.

Mit den Anweisungen (2), (3) und (4) können Wiederholungen abgebrochen (**break**, **return**) bzw. verkürzt (**continue**) werden.

Die Anweisungen (6) und (7) werden zur Markierung in **switch**-Anweisungen verwendet; siehe Abschn. 1.10.3.

Detailliertere Hinweise zur Verwendung dieser Anweisungen finden sich z.B. in [29, 30, 59].

1.11 Arrays (Vektoren, Felder)

In Anwendungen tritt häufig das Problem auf, dass man viele Variablen vom gleichen Typ benötigt, z.B. 2000 Buchungen oder 50 Strings zur Abspeicherung von Namen. Es ist nicht sinnvoll, so viele Variablen explizit einzeln zu vereinbaren. Hier helfen **Arrays**, auch Vektoren oder Felder genannt. Sie gestatten es, mehrere Variablen vom gleichen Typ unter einem einzigen Namen anzusprechen. Die Zeilen

```
int*    feld;
double* buchung;
string* namen;
```

```
int[]   feld;
double[] buchung;
String[] namen;
```

deklarieren drei Referenzen für Arrays von Integer-Werten, Gleitkommawerten und Strings. Zur Definition eines Arrays wird in C++ ein Stern (*) hinter dem Typnamen angegeben; in Java wird nach dem Typnamen oder nach dem Variablennamen ein eckiges Klammernpaar `[]` angegeben.[1] Hiermit sind aber jeweils nur die Referenzen selbst definiert, ohne dass bisher festgelegt wurde, wie lang die Arrays sein und welche Inhalte die einzelnen Variablen (Elemente) der Arrays haben sollen. Es ist auch noch nicht festgelegt, wo (an welcher Adresse) die zugehörigen Werte im Speicher liegen werden. Dies erfolgt durch die folgenden Zeilen:

```
feld    = new int [10];
buchung = new double [2000];
namen = new string[50];
```

```
feld    = new int [10];
buchung = new double [2000];
namen = new String[50];
```

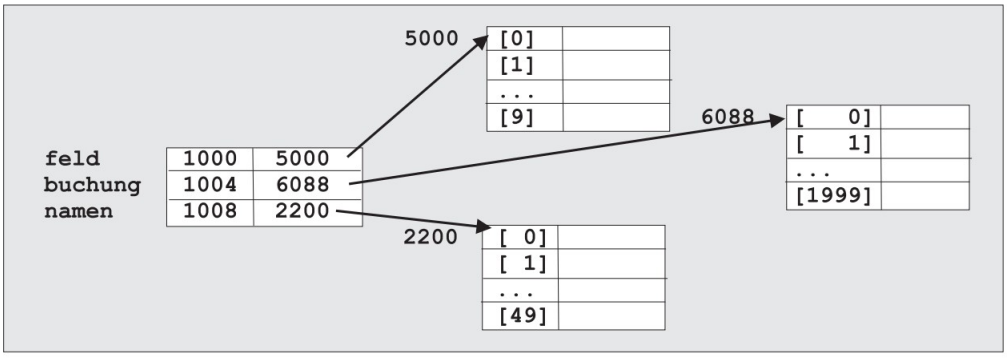

Bild 1.2: Arrays als Referenzen

Bild 1.2 zeigt eine mögliche Speicherbelegung des Programmausschnitts mit den Arraynamen als Referenzen auf die Variablen selbst. `buchung` ist nun ein Verweis auf 2000 Speicherplätze zur Abspeicherung von `double`-Gleitkommavariablen. Die einzelnen Speicherplätze werden durch einen Integer-Ausdruck größer gleich Null als

[1] Die Definition von Arrays in C++ über ein eckiges Klammernpaar ist ebenfalls gestattet. Deren Semantik entspricht aber nicht ganz der von Java; siehe dazu auch Abschn. 2.2.2.

Index angesprochen, der das ausgewählte Element des Arrays angibt, das angesprochen werden soll.

```
int i = 4, j = 22;
buchung[11*i] = 22.4;
namen[0]    = "Fritz";
feld[i*j - 22*j] = 44;
```

```
int i = 4, j = 22;
buchung[11*i] = 22.4;
namen[0]    = "Fritz";
feld[i*j - 22*j] = 44;
```

Der Benutzer ist dafür verantwortlich, dass der verwendete Ausdruck zur Berechnung der Indizes zur Laufzeit einen erlaubten Wert ergibt, d.h. kein negativer Wert und kein Wert größer gleich der Arraygröße, die in der `new`-Anweisung verwendet wurde. Die Indexwerte für `buchung` müssen somit zwischen `0` und `1999` liegen. Ist diese Bedingung wie im Falle des Index für `array` verletzt, wird in Java eine `ArrayIndexOfBoundsException` ausgelöst; in C++ läuft das Programm meist noch eine Weile weiter, allerdings ist sein Verhalten undefiniert.

Die Arrays können auch direkt bei ihrer Definition initialisiert werden. Die Länge des Arrays ergibt sich hierbei aus der Anzahl der übergebenen Werte.

```
int za = 9;
int feld[] = {1, za, 9, 8, -1, 5};
double buchung[] = {1.2, 3.45, 6.0};
string namen[] = {"Fritz", "Hans",
     "Karl", "Henriette", "Heiko"};
```

```
int za = 9;
int feld[] = {1, za, 9, 8, -1, 5};
double buchung[] = {1.2, 3.45, 6.0};
String namen[] = {"Fritz", "Hans",
     "Karl", "Henriette", "Heiko"};
```

In Java wird das `new` quasi automatisch eingefügt. Die Syntax von C++ und Java sind hier zwar identisch, die Semantik ist jedoch eine andere. In C++ wird ein `new` nicht eingefügt, die C++-Arrays werden direkt auf dem Programm-Stack angelegt. Hierauf gehen wir in Zusammenhang mit Zeiger- und Wertesemantik im Kap. 4 im Detail ein.

1.12 Übungen

Übung 1.1: Was gibt das folgende Programm aus?

```
#include <iostream>
using namespace std;
int main(void) {
  int i = 700 / 3;
  int j = 'z' - 'a';
  int k = i > j;
  int b = 8;
  int m = b>10 ? 7 : b > 5 ? 2 : 4;
  bool b1 = 0 < i < 100;
  if (b1) cout << "passt\n";
  else cout << "passt nicht\n";

  cout << i << " " << j << " "
    << k << " " << m << endl;
  return 0;
  }
```

(./Grundlagen/anw1/aufg/main.cpp)

Übung 1.2: Erweitern Sie das folgende Programm, das die Potenzen von 2 ausgibt, sodass nun beliebige Basen (Wahl durch Konstante im Programmcode) verwendet werden können. Außerdem soll nicht bei n^{10} abgebrochen werden, sondern die `10` soll ein Parameter sein.

```
#include <iostream>
using namespace std;
int main(void) {
  int basis = 2; long ergebnis = 1;
  int i;
  cout << "Ausgabe der Potenzen von "
    << basis << "\n";
  for (i=0; i < 10; ++i) {
    cout << basis << " hoch " << i
      << " = " << ergebnis << "\n";
    ergebnis = ergebnis * basis;
  }
  return 0;
}
```

(./Grundlagen/Potenz2ErwN/aufg/PotenzVon2.cpp)

Übung 1.3: Was gibt die Funktion `typen` in den Strom `datei` aus? Beschreiben Sie jeweils an den mit `???` gekennzeichneten Zeilen, was dort geschieht.

```
void typen() {
  double f=0.2;
  f += 0.4;
  if (f == 0.6) { /* ??? */
    datei << "f==0.6: Gleich\n";
    }
  else {
    datei << "f==0.6: Ungleich\n";
    }
  double wert = 7/2; /* ??? */
  datei << "7/2:" << wert << "\n\n";
  }
```

(./Grundlagen/steuer/main.cxx)

Übung 1.4: Was gibt die Funktion `postPreIncrement` in den Strom `datei` aus? Beschreiben Sie jeweils an den mit `???` gekennzeichneten Zeilen, was dort geschieht.

```
void postPreIncrement() {
  int k=44;
  int m = k--;          /* ??? */
  datei << "k=" << k << " m=" << m << endl;
  int n = --k;          /* ??? */
  datei << "k=" << k << " n=" << n << "\n\n";
  }
```

Übung 1.5: Was gibt die Funktion `forLoops` in den Strom `datei` aus? **C++**

```
void forLoops() {
  for (int i=5; i<10; ++i) {
    datei << i << " ";
  }
  datei << "\n";

  for (int j=5; j<10; j++) {
    datei << j << " ";
  }
  datei << "\n";

  int k=44;
  for (; k < 48; k++) {
    datei << k << " ";
  }
  datei << "\n";

  int m=44, n = 11;
  for (;(m<48)&&(n<15); k++, n=k/4) {
    datei << k << " " << n << endl;
    }
  datei << "\n\n";
}
```

(./Grundlagen/steuer/main.cxx)

Übung 1.6: Was gibt das folgende Programm auf den Bildschirm aus?

```
#include <iostream>
using namespace std;
int main(void)
  {
  int i = 0;
  for (i=0; i<5; ++i) { cout << i << " "; } cout << "\n";
  for (i=0; i<5; i++) { cout << i << " "; } cout << "\n";
  for (i=0; i<5; i++) { cout << i++ << " "; } cout << "\n";
  for (i=0; i<5; i++) { cout << ++i << " "; } cout << "\n";
  i = 0;
  for ( ; i++ < 5; ) { cout << i++ << " "; } cout << "\n";
  }
```

Übung 1.7: Implementieren Sie ein Programm, das zwei ganze Zahlen a und b über die Tastatur einliest, anschließend a^b berechnet und das Ergebnis geeignet ausgibt. Verwenden Sie einmal zur Implementierung eine `for`-Schleife und einmal eine `while`-Schleife.

Java **Übung 1.8:** Der `main`-Funktion wird in Java das Argument `String[] args` für Kommandozeilenparameter übergeben. Über `args.length` können Sie die Anzahl der Argumente beim Programmaufruf abfragen, der Ausdruck `Integer.parseInt(args[i])` liefert einen Integer-Wert des i.-ten Arguments. Schreiben Sie ein Programm, welches das Minimum aller Argumente beim Aufruf auf den Bildschirm schreibt (d.h. gar nichts, falls keine Argumente übergeben wurden).

Übung 1.9: Schreiben Sie eine Funktion, die die Fußball-Bundesliga-Tabelle (oder eine andere Tabelle) geeignet in eine Datei ausgibt.

Kapitel 2

Funktionen und Datenstrukturen

Sich ähnlich wiederholende Programmteile werden manchmal erst kopiert und dann modifiziert. Das ist einfach und bequem bei der Erstellung, wird aber sehr schnell unübersichtlich. Außerdem besteht die Gefahr, vorhandenen Fehler gleich mehrfach zu kopieren. Soll der Fehler behoben werden, müssen wir uns an alle Kopien erinnern und korrigieren. In diesem Kapitel werden wir Anweisungsfolgen und Variablen zu Funktionen und Datenstrukturen zusammenfassen und dem Programm damit zu einer besseren Struktur verhelfen.

2.1 Funktionale Abstraktion (Funktionen)

Es gehört zu den praktischsten Eigenschaften eines Rechners, dass man Dinge, die man ihm einmal „beigebracht“ hat, niemals wieder selbst durchführen muss: Der Rechner kann es für uns beliebig oft wiederholen. Wie schon angedeutet, ist eine einmalige Definition von Abläufen auch organisatorisch vorteilhaft, weil Fehler nur zentral an einer Stelle behoben werden müssen, damit ab dann alle weiteren Wiederholungen fehlerfrei abgewickelt werden.

Einen Ablauf „beibringen“ und anschließend „wiederholt ablaufen“ lassen, entspricht der **Definition** und dem **Aufruf** von Funktionen (auch Unterprogramm oder Prozedur genannt). Unterprogramme sind so alt wie die Informatik, und es gibt sie in allen Programmiersprachen. Ohne sie wäre eine effiziente Software-Entwicklung nicht möglich. Auch in unseren Beispielen zu Beginn haben wir schon Funktionen benutzt, bspw. zur Ausgabe von Informationen auf dem Bildschirm (`printf` in C, `System.out.println` in Java; in C++ ist der Funktionsaufruf im Operator `<<` versteckt). Die Strukturierung von Software mit Hilfe von Funktionen und Prozeduren bezeichnet man auch als **prozedurale Programmierung**.

Meistens handelt es sich bei einem Funktionsaufruf um keine exakte Wiederholung früherer Aufrufe, sondern kleine Details wurden verändert, z.B. geben wir unterschiedliche Dinge auf dem Bildschirm aus. Was sich von Aufruf zu Aufruf verändern kann, wird als **Argument** oder **Parameter** der Funktion übergeben. Die Definition einer Funktion legt eine Folge von Anweisungen fest, die nun diese Argumente benutzt, deren konkrete Werte erst beim Aufruf der Funktion festgelegt werden. Manche Funktionen liefern auch ein eindeutiges Ergebnis (z.B. Sinus-Funktion), das als **Funktionswert** oder **Rückgabewert** der Funktion weiterverarbeitet wird.

Die Definition einer Funktion sieht schematisch wie folgt aus:

```
void f( arglist ) { cmdseq }              (Prozedur)
type f( arglist ) { cmdseq; return r; }   (Funktion)
```

Dabei ist `type` der Typ des Funktionswertes (`void` im Fall einer Prozedur ohne Rückgabewert), `f` der Funktionsname, `arglist` eine Folge von Argumenten und `cmdseq` eine Sequenz von Anweisungen. Am Ende eines Durchlaufs durch eine Funktion muss eine Anweisung der Art `return r` stehen, die `r` als den Wert auszeichnet, der von der Funktion als Ergebnis zurückgegeben werden soll. Auch eine Prozedur kann eine oder mehrere `return`-Anweisungen enthalten, denen allerdings kein Wert nachstehen darf, weil Prozeduren keinen Wert zurückliefern. Ob Prozedur oder Funktion, das Erreichen der Anweisung `return` bewirkt, dass die Bearbeitung des Unterprogramms beendet wird.

C++/Java 2.1: Definition einer Funktion `summe`

```cpp
void summe() {
  int s=0;
  for (int i=1; i<101; ++i) {
    s += i;
  }
  cout<<"Summe ist"<< s << "\n";
}
```

(Funk/Summe/Mathe.cpp)

```java
public class Mathe {
  public static void summe() {
    int s = 0;
    for (int i=1; i<101; ++i) {
      s += i;
    }
    System.out.println("Summe ist "+s);
  };
};
```

(Funk/Summe/Mathe.java)

Listing 2.1 zeigt ein einfaches Beispiel für eine Prozedur, die die Summe der Zahlen zwischen 1 und 100 auf dem Bildschirm ausgibt. Am Beispiel der Variablen `s` im Listing 2.1 erkennen wir, dass in Funktionen **lokale Variablen** vereinbart werden können, die außerhalb der Funktion nicht sichtbar sind. Nach Beendigung der Funktion ist ihr Wert nicht mehr definiert, d.h. lokale Variablen verbrauchen Speicherplatz nur während der Ausführung der Funktion, in der sie definiert sind. Lokale Variablen verdecken die Variablen gleichen Namens, die außerhalb der Funktion definiert sind. Das ermöglicht erst die Entwicklung von großen Programmsystemen im Team: Der einzelne Entwickler braucht sich nicht darum zu kümmern, welche Variablennamen außerhalb der von ihm entwickelten Funktionen verwendet werden.

Wichtiger Hinweis für den weiteren Umgang mit Java: Bevor wir Funktionen und Prozeduren im Detail betrachten, müssen wir an dieser Stelle bzgl. der Verwendung von Java etwas klarstellen. Wie schon zuvor erwähnt, ist es ein erklärtes Ziel des

Buches, aus der schrittweisen Verbesserung der prozeduralen Programmierung die objektorientierte Programmierung „abzuleiten“. Die Sprache C++ eignet sich dafür hervorragend, weil sie sowohl rein prozedural (Untermenge C) als auch objektorientiert verwendet werden kann. Java ist hingegen eine rein objektorientierte Sprache, was nicht bedeutet, dass wir in Java nicht auch prozedural programmieren könnten, aber es ist nicht unbedingt so gedacht. Bis wir Java also genau so nutzen, wie es vorgesehen ist, werden wir uns noch einige Kapitel gedulden müssen. Dem Leser sei versichert, dass nichts, was er bis dahin lernt, hinterher überflüssig oder falsch ist. Um alle Beispiele parallel in C++ und Java zeigen zu können, treffen wir folgende Vereinbarung: Vor jede Funktion setzen wir in Java die Schlüsselworte `public static`. Außerdem notieren wir alle Funktionen innerhalb der geschweiften Klammern von `public class IrgendeinName { ... }`, wie in Listing 2.1 gesehen (manchmal nicht abgedruckt, aber in den angegebenen Quellcode-Dateien immer vorhanden). Damit ordnen wir die Funktionen einer Klasse `IrgendeinName` zu. (Solche Funktionen werden Methoden genannt.) Wir verstehen unter einer Klasse fürs Erste einfach eine Zusammenfassung mehrerer Funktionen. Der Name der Datei, in der die Funktionen abgelegt werden, muss mit dem Klassennamen übereinstimmen. Damit haben wir zunächst die Semantik der C++-Funktionen erhalten, die genaue Bedeutung dieser Schlüsselworte werden wir im weiteren Verlauf klären.

2.1.1 Funktionen definieren und aufrufen

Funktionsdefinition

Die Funktion `summe` im Listing 2.1 ist ein Beispiel für eine Prozedur (oder auch Anweisungsfunktion), die keine Argumente besitzt. Listing 2.2 zeigt nun eine *echte* Funktion `summeP`, die die Summe der Zahlen zwischen ihren beiden Eingangsparametern `a` und `b` zurückliefert.

C++/Java 2.2: Definition einer echten Funktion mit Parametern

```cpp
int summeP(int a, int b) {
  int s=0;
  for (int i=a; i<b+1; ++i) {
    s += i;
  }
  return s;
}
```

(Funk/Summe/SummeP.cpp)

```java
public static int summeP(int a, int b) {
    int s = 0;
    for (int i=a; i<b+1; ++i) {
      s += i;
    }
    return s;
  };
```

(Funk/Summe/SummeP.java)

Es gibt zwei verschiedene Mechanismen, um Parameter an eine Funktion zu übergeben:

- Beim *call by value*, auch Übergabe per **Werteparameter** genannt, verhält sich der formale Parameter wie eine lokale Variable, die durch den Wert des aktuellen Parameters initialisiert wird. Man kann auch sagen: Der Wert des aktuellen Parameters wird in den formalen Parameter kopiert und die Funktion arbeitet dann mit einer Kopie weiter. Eine Änderung dieses Parameters während der Funktionsausführung hat also keinerlei Auswirkungen außerhalb des Funktionsaufrufes.

- Beim *call by reference*, auch Übergabe per **Referenzparameter** oder per **Variablenparameter** genannt, ist der formale Parameter nur ein anderer Name für den aktuellen Parameter, d.h. eine Referenz darauf. Ändert die Funktion den formalen Parameter, so ändert sich genauso der entsprechende aktuelle Parameter in der aufrufenden Funktion. Referenzparameter stellen somit eine weitere Möglichkeit dar, um Ergebnisse aus der Funktion nach außen zu liefern. Referenzparameter werden detailliert in Abschn. 2.2.2 und Kap. 4 behandelt.

In C++ hat man für (fast) jeden Parameter durch Hinzufügen des Ampersand-Zeichens (&) die Möglichkeit, festzulegen, dass es sich bei diesem Parameter um einen Referenzparameter und damit um einen Ausgabeparameter handelt. In Java dagegen werden die einfachen Typen wie `double`, `int`, `boolean` immer als Werteparameter übergeben und alle anderen Typen (z.B. strukturierte Datentypen; siehe Abschn. 2.2) immer als Referenzparameter. Arrays (siehe Abschnitt 1.11) werden in beiden Sprachen immer wie Referenzparameter behandelt, d.h., eine Änderung eines Arrayelements in der Funktion führt immer auch zu einer Änderung des entsprechenden aktuellen Parameters in der aufrufenden Funktion.

Der folgende Programmausschnitt zeigt die Implementierung der Funktion `summeR` als Anweisungsfunktion mit einem Referenzparameter `s` für das Ergebnis.

C++/Java 2.3: Funktionen mit Referenzparameter

```cpp
void summeR(int a, int b, int& s) {

  s = 0;
  for (int i=a; i<b+1; ++i) {
    s += i;
  }
}
```

(Funk/Summe/SummeP.cpp)

```java
public static void summeR(
          int a, int b, int s[]) {
    s[0] = 0;
    for (int i=a; i<b+1; ++i) {
      s[0] += i;
    }
  };
```

(Funk/Summe/SummeP.java)

Wir sehen, dass wir in Java zu einem Trick greifen müssen, indem wir den Eingangsparameter als Array definieren. Eine andere Möglichkeit wäre, einen neuen, strukturierten Typ zu definieren, der lediglich aus einem Element zur Speicherung eines `int`-Wertes besteht.[1]

Eine recht häufig benötigte Funktionalität ist das Vertauschen (engl. swap) der Werte zweier Variablen. Das Beispiel in Listing 2.4 zeigt nochmals ganz deutlich, dass die Funktion `swapWert` durch die Verwendung von Werteparametern den gewünschten Effekt nach außen nicht erzielt. Die Funktion ändert nur den Inhalt der formalen Parameter, ohne einen Einfluss auf die übergebenen aktuellen Parameter der aufrufenden Funktion zu haben. `swapRef` ändert dagegen auch die Werte der aktuellen Parameter der aufrufenden Funktion.

[1] Dies darf allerdings nicht mit den *Hüllklassen* (*wrapper classes*), die Java selbst zur Verfügung stellt, verwechselt werden (Abschn. 4.1.6), da letztere für diesen Zweck nicht geeignet sind. Siehe auch Abschn. 4.1.5.

C++ 2.4: Unterschiedliche Wirkung von Werte- und Referenzparametern

```
void swapWert(int a, int b)
{
  int tmp=a; a=b; b=tmp;
}

void swapRef(int& a, int& b)
{
  int tmp=a; a=b; b=tmp;
}
```

```
void testRef() {
  int i=10, j=9;
  swapWert(i, j);
  // Ausgabe: 10 9
  cout << i << " " << j;
  swapRef(i, j);
  // Ausgabe: 9 10
  cout << i << " " << j;
}
```

(Funk/Summe/FunkRef.cpp)

Funktionsaufruf

Der Aufruf einer Funktion erfolgt wie in den folgenden Codebeispielen angegeben:

```
summe();
int st = 5;
int e = summeP(3, 14*15-1+st);
e = summeP(st,8) + summeP(st,9);
```

```
Mathe.summe();
int st = 5;
int e = Mathe.summeP(3,14*15-1+st);
e=Mathe.summeP(st,8)+Mathe.summeP(st,9);
```

Befinden sich die Funktionsaufrufe in derselben Klasse wie die Funktionsdefinition (vgl. Hinweise zu Java auf Seite 38), so ist die Syntax die gleiche wie in C++. Erfolgt der Aufruf allerdings von außerhalb der Klasse `Mathe`, zu der die Funktionen `summe` und `summeP` hier gehören, muss dem Aufruf der Klassenname `Mathe` wie gezeigt vorangestellt werden.

Beim Aufruf einer echten Funktion steht der Funktionsname normalerweise in einem Ausdruck rechts von einer Zuweisung, d.h. der Rückgabewert der Funktion wird weiterverwendet. Eine Anweisungsfunktion (z.B. `summe` im Listing 2.1) liefert dagegen nichts zurück, sodass der Aufruf der Prozedur selbst schon die ganze Anweisung ist. Beim Aufruf einer Funktion übergibt die aufrufende Funktion die Kontrolle (vorübergehend) an die aufgerufene Funktion. Nach Beendigung der Funktion erfolgt der Rücksprung an die Aufrufstelle in der aufrufenden Funktion. Beim Aufruf einer echten Funktion wird der Rückgabewert für den Funktionsaufruf eingesetzt und damit weitergearbeitet.

Für Werteparameter dürfen Konstanten, Werte, Ausdrücke oder Variablen als aktuelle Parameter eingesetzt werden. Dagegen müssen für Referenzparameter immer Variablen übergeben werden, denn die Referenzparameter werden lediglich als anderer Name für die angegebenen Variablen verwendet. Der folgende Codeausschnitt zeigt die Übergabe von aktuellen Parametern an Referenzparameter. Der Wert der Variablen `e` wird durch Verwendung der Anweisungsfunktion berechnet, anstatt durch die echte Funktion `summeP` wie im vorigen Listing.

```
int f;
summeR(st, 8, f);
e = f;
summeR(st, 8, f);
e += f;
```

```
int f[] = {8};
Mathe.summeR(st, 8, f);
e = f
Mathe.summeR(st, 9, f);
e += f;
```

C++

Prototypen

Java und C++-Programme werden von einem Compiler erst auf korrekte Syntax geprüft, bevor sie übersetzt und damit ausführbar werden. In Java ist jede Funktion, wurde sie erst einmal *definiert* (d.h. implementiert), grundsätzlich überall bekannt. In C++ muss **entweder** die Definition *vor* dem Aufruf erfolgen **oder** der Funktionskopf, bestehend aus Rückgabetyp, Funktionsname und formalen Parametern, wird, wie in Listing 2.5 gezeigt, in Form eines sog. *Prototypen* vorher *deklariert*.

C++ 2.5: Beispiel mit Prototypen

```
void swapWert(int a, int b);
void swapRef(int& a, int& b);

void testRef() {
  int i=10, j=9;
  swapWert(i, j); cout << i << " " << j;
  swapRef(i, j); cout << i << " " << j;
}
```

(Funk/Summe/FunkRefProto.cpp)

```
void swapWert(int a, int b) {
  int tmp=a; a=b; b=tmp;
}

void swapRef(int& a, int& b) {
  int tmp=a; a=b; b=tmp;
}
```

2.1.2 Überladen von Funktionen

Eine Funktion wird nicht allein über ihren Namen identifiziert, sondern über ihre **Signatur**, d.h. über ihren Namen sowie zusätzlich über Anzahl und Typ der Argumente. Die im Listing 2.6 gezeigten Beispiele gelten ganz analog für Java.

C++ 2.6: Überladen von Funktionen

```
// Prototypen
void f (int, int);
void f (long, long);
void f (double, double);

// Implementierungen:
void f (int, int) {/* ... */}
void f (long, long) {/* ... */}
void f (double, double) {/* ...*/}
```

(Funk/Summe/Ueberladen.cpp)

```
// Demo-Programm
int  i = 1, j = 2;
long k = 4, m = 3;
double x = 1.1, y = 2.2;
/*
// Eindeutige Aufrufe
swap(i, j); swap(k, m); swap(x, y);
// Mehrdeutige Aufrufe --> Syntaxfehler
swap(i, k); swap(j, x); swap(m, y);
```

(Funk/Summe/Ueberladen.cpp)

Der Compiler erkennt bei den ersten drei Aufrufen anhand der aktuellen Parameter, welche der drei gleichnamigen Funktionen jeweils zu verwenden ist. Bei den folgenden Aufrufen passt keine Funktion genau. Über die definierten Typkonvertierungen wird dann versucht, eine eindeutig auszuwählen. Im Beispiel scheitert auch das. Eine bessere Lösung, als die Funktion mehrfach zu implementieren, lernen wir in Abschn. 2.3 kennen.

Neben den Parametertypen können sich die überladenen Funktionen in C++ und Java auch einfach in der Anzahl der Parameter unterscheiden, wie in Listing 2.7 am Beispiel der `min`-Funktion für Java gezeigt ist:

Java 2.7: Überladen von Funktionen mit unterschiedlicher Parameteranzahl

```
public static int
          min (int a, int b){/*...*/}
public static int
     min (int a, int b, int c){/*...*/}
public static int
  min (int a,int b,int c,int d){/*...*/}
```

```
// Demo-Funktion
public static void testMin() {
  int erg = min(4, 6);
  erg = min(3, 7, 4);
  erg = min(4, 2, 3, 9);
}
```

(Funk/Summe/UeberladenMin.java)

Der Rückgabetyp einer Funktion dient allerdings nicht zur Identifizierung einer Funktion. Daher führen die folgenden Deklarationen zu einem Syntaxfehler:

```
// Prototypen
void swap (int&, int&);
int  swap (int&, int&);
```

Der Compiler kann nicht entscheiden, welche Funktion durch den Aufruf `int i=1, j=9; swap(i, j);` ausgeführt werden soll.

Default-Parameter

C++ erlaubt die Angabe von *Default-Werten für Parameter*. Argumente mit Default-Wert (z.B. `int i=0`) können beim Aufruf wahlweise weggelassen werden. Fehlt das Argument bei einem Funktionsaufruf, erhält es automatisch den angegebenen Default-Wert. Mit Hilfe der überladenen Funktionen können Defaultparameter in Java aber simuliert werden, wie das folgende Beispiel demonstriert:

```
double sum(double a, double b,
      double c=0.0, double d=0.0) {
  return a + b + c + d;
}
void test1(){
/* i.O.: y1 == 11.0 */
double y1 = sum(1.1, 2.2, 3.3, 4.4);
/* i.O.: y2 == 6.6 */
double y2 = sum(1.1, 2.2, 3.3);
/* i.O.: y3 == 3.3 */
double y3 = sum(1.1, 2.2);
}
```

```
double sum(double a, double b,
      double c, double d) {
  return a + b + c + d;
}
double sum(double a,double b,double c){
  return sum(a, b, c, 0.0);
}
double sum(double a, double b) {
  return return sum(a, b, 0.0, 0.0);
  }
void test1(){
double y1 = sum(1.1, 2.2, 3.3, 4.4);
double y2 = sum(1.1, 2.2, 3.3);
double y3 = sum(1.1, 2.2);
}
```

Beim Weglassen von aktuellen Parametern geht in C++ die Reihenfolge stets von rechts nach links, dadurch wird die Zeile `y2 = summe(1.1, 2.2, 3.3);` eindeutig: der dritte Parameter hat den Wert `3.3` und der vierte den Wert `0.0`.

Refactoring durch funktionale Abstraktion

Von Refactoring spricht man allgemein, wenn man Änderungen am Code vornimmt, ohne seine Funktionalität zu verändern. Refactoring dient somit *lediglich* dazu, über-

sichtlichen, wartungsfreundlichen und damit weniger fehleranfälligen Code zu erhalten. Genau das haben wir in diesem Kapitel durchgeführt.

Eine Art des Refactoring ist die einfache Aufspaltung einer großen Funktion in **mehrere** kleinere Funktionen – im Original wird jede dieser Funktionen **einmal** aufgerufen. Ein Hauptprogramm, das nacheinander die Funktionen `test1` bis `testn` aufruft, ist viel schneller zu überblicken und zu verstehen (vgl. Listing **??**).

Eine zweite Art des Refactoring ist die Zusammenfassung gleichartiger Funktionalität zu **einer** Funktion, die dann an **mehreren** Stellen aufgerufen wird. Das haben wir bspw. bei der Funktion `plane` praktiziert, die nun in jedem Test aufgerufen wird.

Wenn man beide Möglichkeiten zur Auswahl hat, welche soll man dann zuerst anwenden? Die Antwort ist schwierig und einfach zugleich. Man sollte es gar nicht erst so weit kommen lassen, sondern viel früher eine Aufspaltung in Funktionen durchführen, und nicht erst, wenn es gar nicht mehr anders geht und die Funktion Hunderte von Codezeilen enthält. Wenn man dann aber doch beide Möglichkeiten hat, ist es meist gleichgültig, womit man beginnt. Man sollte die Schritte allerdings nacheinander durchführen, da sonst die Gefahr des versehentlichen Einbaus von Fehlern zu groß ist.

Dieses Refactoring wird natürlich wesentlich erleichtert, wenn man bereits über automatisch ablaufende Tests verfügt . Die Tests müssen dann nämlich vor dem Refactoring und nach dem Refactoring erfolgreich durchlaufen.

Die Notwendigkeit für ein Refactoring ergibt sich oft nach einer Erweiterung der Funktionalität. Wir werden im Folgenden noch einige Refactoring-Techniken kennenlernen und verweisen den Leser im Übrigen auf [22].

Wenn man sich allerdings manche Funktionen näher ansieht, fällt auf, dass manchmal die Schnittstelle einer Funktion durch die große Anzahl der Parameter mehr Codezeilen belegt als die Funktion selbst. Diesem Problem werden wir uns im nächsten Abschnitt zuwenden, in dem wir lernen, wie man Variablen zu so genannten *strukturierten Datentypen* zusammenfassen kann.

2.1.3 Übungen

Übung 2.1: Was gibt der folgende Programmausschnitt in die Datei `datei` aus?

```
void f1(int& i) {
  i = 44;
  }
void f2(int i) {
  i = 44;
  }
```

(./Funk/WasGibtAus/main.cpp)

```
int main() {
  int i = 55;
  int j = 22;
  f1(i);
  f2(j);
  datei << "i=" << i
    << ", j=" << j <<endl;
  }
```

Übung 2.2: Implementieren Sie eine Funktion `istGerade`, die testet, ob eine übergebene ganze Zahl (... -3, -2, -1, 0, 1, 2, 3 ...) eine gerade Zahl (0, 2, 4, ...) ist.

Wird der Funktion eine negative Zahl übergeben, wird die Funktion `false` als Funktionswert liefern, andernfalls `true`. Falls im letzteren Fall der Funktion eine gerade Zahl übergeben wird, soll die Funktion in ihrem zweiten Parameter den Wert `true` zurückliefern, andernfalls (ungerade Zahl) `false`. Die Funktion soll z.B. wie folgt aufgerufen werden können:

```
bool gerade;
bool fwert;
fwert = istGerade(3, gerade);
```

```
boolean gerade[]=new boolean[1];
boolean fwert;
fwert = istGerade(3, gerade);
```

Gehen Sie bei der Erstellung der Lösung zumindest in den folgenden Schritten vor: Definition und Implementierung von Tests, Beschreibung der Funktion, Implementierung der Funktion, Ausführung der Tests, evtl. Fehlersuche/Verbesserung der Implementierung.

Übung 2.3: Implementieren Sie die folgende Anweisungsfunktion auch als echte Funktion und geben Sie mindestens zwei Implementierungen für Tests der echten Funktion an.

```
void summe(int& s, int n) {
  s = 0;
  for (int i=0; i<n; ++i) {
    s += i*i;
    }
  }
```

```
void summe(int s[], int n) {
  s[0] = 0;
  for (int i=0; i<n; ++i) {
    s[0] += i*i;
    }
  }
```

Übung 2.4: Implementieren Sie eine Funktion `mathe`, die sowohl `a + b`, als auch `a - b` als auch `a * b` liefert, d.h. mit einem einzigen Aufruf.

Übung 2.5: Was gibt der folgende Programmausschnitt in die Datei `datei` aus?

```
void f1(int k);
void f1(double m);
void f1(double n, double p);
void f1(bool b, double k=4, double m=22);

int main(){
  f1 (8);
  f1 (8.0);
  f1(true);
  f1 (2, 4, 7);
  f1 (false, 4.0);
  f1(true, 21);
  }
```

```
void f1(int k) {
  datei << k << "\n";
  }
void f1(double m) {
  datei << m << "\n";
  }
void f1(double n, double p) {
  datei << n << " " << p << "\n";
  }
void f1(bool b, double k, double m){
  datei << b << " " << k
    << " " << m << "\n";
  }
```

(./Funk/WasGibtAus/Defaultpar.cpp)

Übung 2.6: Implementieren Sie eine Funktion `fibo`, die die N-te Fibonacci-Zahl fib_N berechnet, wobei gilt: $fib_0 = 1$, $fib_1 = 1$ und $fib_n = fib_{n-1} + fib_{n-2}$, d.h. die folgende Reihe soll geliefert werden: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

Implementieren Sie zunächst eine rekursive und dann eine iterative Lösung für eine echte Funktion. Ändern Sie anschließend die Funktion `fibo`, sodass sie als Anweisungsfunktion (Rückgabetyp `void`) statt eines Funktionswerts das Ergebnis in einem

Referenzparameter zurückliefert, d.h. nach dem Aufruf `fibo(6,wert);` hat `wert` anschließend den Wert `13`.

2.2 Datenabstraktion: Strukturierte Datentypen

Die Variablen der elementaren Datentypen wie `int` oder `double` speichern direkt ihre Werte. Eine Variable ist an einer bestimmten Speicherstelle im Arbeitsspeicher des Rechners abgelegt.

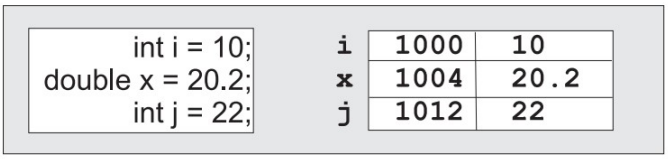

Bild 2.1: Speicherbelegung von elementaren Datentypen

Im Bild 2.1 ist angenommen, dass die Variable `i` ab der Adresse `1000` im Speicher abgelegt ist und `4` Byte belegt. Variable `x` vom Typ `double` beginnt dann entsprechend bei `1004` und belegt `8` Byte usw. Wird nun `x` auf den Wert `48` gesetzt, werden entsprechend die Speicherstellen ab `1004` verändert.

In Zusammenhang mit strukturierten Datentypen sind insbesondere **Referenzen** von zentraler Bedeutung. In einem Referenztyp wird im Gegensatz zu den elementaren Variablen nur eine Referenz auf den eigentlichen Wert der Variablen gespeichert. Wären die Variablen `i`, `x` und `j` aus dem vorigen Beispiel Referenzen, könnte die Speicherbelegung wie folgt aussehen:

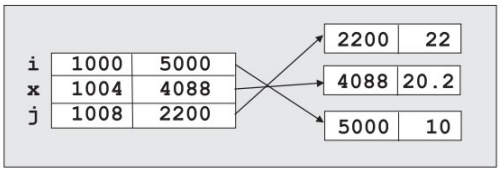

Bild 2.2: Speicherbelegung von Referenzen

Die Speicherstelle `1000` enthält somit nicht mehr direkt den Wert der Variablen `i`, sondern einen Verweis auf die Speicherstelle, die den Wert enthält: In diesem Fall steht der Wert von `i` somit in der Speicherstelle `5000`. `Referenzen` werden wir im Detail in Kap. 4 behandeln.

Bisher haben wir elementare Datentypen, Steueranweisungen und Funktionen kennengelernt. Hiermit ist es zwar zumindest im Prinzip möglich, jedes Programm zu

schreiben, aber für größere Programme wäre dies zumindest sehr, sehr mühselig, weil man z.B. für jede Variable einen eigenen Namen vergeben müsste. Wir werden deshalb in diesem Abschnitt zwei Verbesserungen kennenlernen:

- Ein **Array** *(Vektor)* ist eine **homogene Datenstruktur**, die verschiedene Datenelemente gleichen Typs zusammenfasst. Ein einzelnes Datenelement wird dann über einen Index als Selektor ausgewählt; siehe auch Abschnitt 1.11.
- Eine **Struktur** *(Record)* ist hingegen eine im Allgemeinen **inhomogene Datenstruktur**, die verschiedene Datenelemente unterschiedlichen Typs zusammenfassen kann; ein einzelnes Datenelement (häufig *Komponente, Attribut* oder engl. *Mitglied, member* genannt) wird dann über einen symbolischen Komponentennamen als Selektor ausgewählt.

2.2.1 Strukturen und Klassen

Im Abschnitt 1.11 sind Arrays beschrieben, um homogene Datenelemente, d.h. solche vom gleichen Typ, zusammenzufassen. Jetzt lernen wir die **Struktur** (*Record*) kennen, um verschiedene Datenelemente unterschiedlichen Typs zusammenzufassen.

Deklaration einer Klasse bzw. Struktur

Das folgende Code-Fragment zeigt z.B. links in C++ und rechts in Java die Beschreibung eines Vorgangs, der durch seine Dauer, seinen frühesten Anfang und sein spätestes Ende beschrieben werden soll. Diese einzelnen Elemente werden auch als Attribute, Komponenten oder Datenelemente bezeichnet. Wir werden sie im Folgenden als **Attribute** bezeichnen.

```
struct Vorgang {
  double dauer;
  double fruehanf;
  double spaetend;
};
```

```
public class Vorgang {
  double dauer;
  double fruehanf;
  double spaetend;
}
```

Wir sehen, dass sich hier die Syntax beider Sprachen kaum unterscheidet. In C++ wird für eine Struktur das Schlüsselwort **struct** benutzt. In Java benutzen wir wieder das Schlüsselwort **class**, das es aber auch in C++ gibt. Auf Seite 38 hatten wir schon Funktionen zu einer Klasse zusammengefasst, nun sehen wir, dass auch Daten zu einer Klasse zusammengefasst werden können. Damit kommen wir der tatsächlichen Bedeutung einer Klasse immer näher, werden die volle Bedeutung des Begriffs Klasse aber erst in Kap. 6 diskutieren.

Instanziierung einer Klasse bzw. Struktur

Wir sehen nun, wie Variablen (Objekte) einer Klasse bzw. Struktur erzeugt werden – man spricht hier von **Instanziierung**.

```
Vorgang* v1 = new Vorgang();
Vorgang* v2;
v2       = new Vorgang();
```

```
Vorgang v1 = new Vorgang();
Vorgang v2;
v2       = new Vorgang();
```

Entsprechend der Erzeugung von Arrays mit dem new-Operator wird hierdurch lediglich eine Referenz, d.h. ein Verweis, auf eine Instanz (Variable) der Klasse erzeugt; siehe Bild 2.3. Details zum Operator new sind im Abschn. 4.1.3 zu finden.

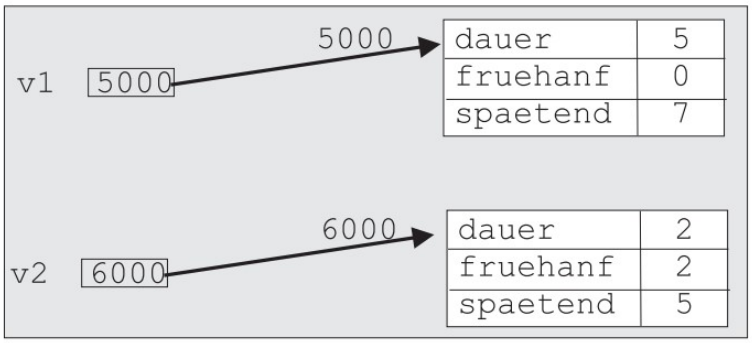

Bild 2.3: Klassen/Strukturen als Referenzen

Zugriff auf einzelne Attribute

Das folgende Code-Fragment zeigt, wie lesend und schreibend auf die einzelnen Attribute zugegriffen werden kann.

```
v1->dauer = 7;
v2->fruehanf = 0;
(*v1).spaetend = 7;
(*v2).fruehanf = 5;

cout << v1->dauer << " " << v2->dauer;
```

```
v1.dauer = 7;
v2.fruehanf = 0;
v1.spaetend = 7;
v2.fruehanf = 5;

System.out.print(v1.dauer+" "+v2.dauer);
```

In C++ erfolgt der Zugriff wahlweise über den Pfeil- (->) oder den Punkt-Operator. Bei Verwendung des Punkt-Operators muss die Referenz zunächst mit dem Stern-Operator (*) in den Inhalt umgewandelt werden; siehe auch Abschn. 4.1.4 auf Seite 80 für weitere Details. In Java steht nur der Punkt-Operator zur Verfügung. Die Umwandlung erfolgt damit quasi automatisch.

2.2.2 Referenzen auf Arrays und Klassen

Mit dem new-Operator reservieren wir Speicherplatz für die Variable und liefern eine Referenz hierauf zurück. Dies hat entscheidende Konsequenzen bei der Verwendung von Referenztypen und wird in diesem Abschnitt nochmals herausgearbeitet.

Zuweisung von Referenzdatentypen

Im folgenden Programmausschnitt wird die Referenz `v1` der Referenz `v2` zugewiesen.

```
Vorgang* v1 = new Vorgang();
v1->dauer = 55;

Vorgang* v2;
v2 = v1;// v2, v1 zeigen auf gl. Objekt

v2->dauer = 44;

// Ausgabe 44 44
cout << v1->dauer << " " << v2->dauer;
```

```
Vorgang v1 = new Vorgang();
v1.dauer = 55;

Vorgang v2;
v2 = v1;// v2, v1 zeigen auf gl. Objekt

v2.dauer = 44;

// Ausgabe 44 44
System.out.print(v1.dauer+" "+v2.dauer);
```

Beide Referenzen verweisen nun auf die gleiche Instanz (auf das gleiche Objekt/die gleiche Variable), wie das Bild 2.4 veranschaulicht.

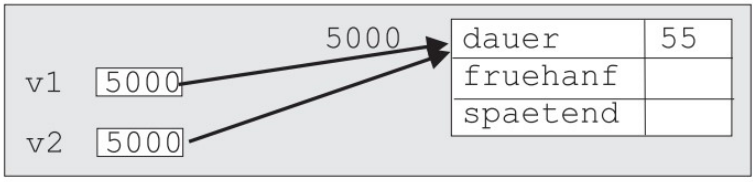

Bild 2.4: Zwei Referenzen verweisen auf die gleiche Variable

Wird nun z.B. das Attribut `dauer` der Variablen über die Referenz `v2` verändert, ändert sich damit natürlich auch `v1->dauer`, weil `v1` und `v2` auf dieselbe Variable verweisen. Entsprechendes gilt für die Verwendung von Arrays, wie der Programmausschnitt 2.8 und das zugehörige Bild 2.5 zeigen.

C++/Java 2.8: Zuweisung von Arrayreferenzen

```
int* array1 = new int[6];
array1[0] = 55;

int* array2;
array2 = array1;// Verweis auf gl. Array

array2[0] = 44;

// Ausgabe 44 44
cout << array1[0] << " " << array2[0];
```

(Funk/Klassen/RefVerwendung.cpp)

```
int array1[] = new int[6];
array1[0] = 55;

int array2[];
array2 = array1; // Verweis auf gl. Array

array2[0] = 44;

// Ausgabe 44 44
System.out.println(array1[0]+" "+array2[0]);
```

(Funk/Klassen/RefVerwendung.java)

Die Verwendung von zwei verschiedenen Referenzen, die auf die gleiche Variable verweisen, mag an dieser Stelle zunächst wenig sinnvoll erscheinen, aber in Zusammenhang mit den formalen Funktionsparametern werden wir im Folgenden die Nützlichkeit sofort erkennen.

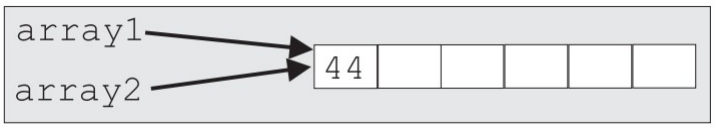

Bild 2.5: Zwei verschiedene Referenzen auf ein Array

Referenzen als formale Parameter

Wir sehen im folgenden Listing am Beispiel von `Vorgang`, wie Referenzen als Ausgangsparameter von Funktionen verwendet werden können.

```
void initVorg(Vorgang* v) {
  v->dauer = 44;
  v->fruehanf = 21;
  v->spaetend = 56;
}

void printVorg(Vorgang* v) {
  cout << "Dauer:" << v->dauer;
  cout << "\nFB:" << v->fruehanf;
  cout << "\nSE:" << v->spaetend;
}

void testInit() {
  Vorgang* v1 = new Vorgang;
  initVorg(v1);
  printVorg(v1);
}
```

```
void initVorg(Vorgang v) {
  v.dauer = 44;
  v.fruehanf = 21;
  v.spaetend = 56;
}

void printVorg(Vorgang v) {
  System.out.print("Dauer:" + v.dauer);
  System.out.print("\nFB:" + v.fruehanf);
  System.out.print("\nSE:" + v.spaetend);
}

void testInit() {
  Vorgang v1 = new Vorgang();
  initVorg(v1);
  printVorg(v1);
}
```

Der aktuelle Parameter `v1` in der Funktion `testInit` verweist auf das gleiche Objekt bzw. die gleiche Variable wie der formale Parameter `v` in den Funktionen `initVorg` und `printVorg`. Damit wirken sich alle Änderungen in den Funktionen `initVorg` und `printVorg` auch auf die Hauptfunktion aus, d.h. auf das Objekt, auf das `v1` verweist. In `initVorg` ist die Referenz `v` ein Ausgabeparameter, denn die Attribute des referenzierten Objekts vom Typ `Vorgang` werden verändert. In `printVorg` wird hingegen das Objekt nicht verändert, es ist somit nur ein Eingangsparameter. Syntaktisch ist dieser Unterschied aber bisher nicht zu erkennen.

Konstante Referenzparameter

Wie wir im Abschnitt 1.3.4 beschrieben haben, stellen beide Sprachen das Schlüsselwort `const` bzw. `final` zur Beschreibung von Konstanten zur Verfügung. Allerdings bietet nur C++ die Möglichkeit, auch formale Parameter sinnvoll mit dem Schlüsselwort `const` als unveränderliche Eingangsparameter zu kennzeichnen. Das folgende Listing zeigt, dass die Verwendung von `final` prinzipiell auch in Java möglich ist, allerdings mit anderer Semantik. Hier bedeutet es nur, dass die Referenz in der Funktion nicht auf ein anderes Objekt *umgebogen* wird. Diese Information ist aber für

den Benutzer einer Funktionsschnittstelle von eher untergeordneter Bedeutung, denn nach Aufruf der Funktion zeigt die Referenz in der aufrufenden Funktion `testInit` in jedem Fall immer noch auf das gleiche Objekt (auf die gleiche Variable)!

```
void printVorg(const Vorgang* v) {
  cout << "Dauer:" << v->dauer;
  cout << "\nFB:" << v->fruehanf;
  cout << "\nSE:" << v->spaetend;
  // alles syntaktisch falsch wäre:
  v->dauer = 44;
  v->fruehanf = 11;
  v = new Vorgang; // richtig
}
```

```
void printVorg(final Vorgang v) {
  System.out.print("Dauer:" + v.dauer);
  System.out.print("\nFB:" + v.fruehanf);
  System.out.print("\nSE:" + v.spaetend);

  v.dauer = 44; // leider richtig
  v.fruehanf = 11; // leider richtig
  v = new Vorgang(); // falsch
}
```

Im obigen Beispiel würde der schreibende Zugriff auf einen als konstant gekennzeichneten Parameter zu einem Syntaxfehler führen. Deshalb ist es in C++ nicht nur guter Programmierstil, Referenzen, die lediglich als Eingangsparameter dienen, mit dem Schlüsselwort `const` zu kennzeichnen, sondern geradezu fahrlässig, diese Unterstützung vom C++-Compiler nicht zu nutzen. **Tipp**

2.2.3 Übungen

Übung 2.7: Implementieren Sie eine Funktion `minMaxDurch`, der beliebig viele ganzzahlige Argumente in einem Array übergeben werden und die das maximale und das minimale Arrayelement sowie den Durchschnitt aller Arrayelemente zurückliefert.

Übung 2.8: Implementieren Sie nochmals die Funktion `mathe` aus Übung 2.4. Verwenden Sie nun einen strukturierten Typ, um die drei Ausgabewerte einmal als Funktionswert und einmal über einen Referenzparameter zurückzugeben. Begründen Sie Ihre Entscheidung, ob zur Rückgabe des Funktionswertes ein Wert oder eine Referenz die bessere Wahl ist. **C++**

Übung 2.9: Implementieren Sie den Quicksort-Algorithmus (siehe z.B. [30]) zum Sortieren eines Arrays einmal zum Sortieren von `int`-Werten und einmal zum Sortieren eines Array mit `Vorgang`-Elementen. Die Vorgänge sollen aufsteigend nach ihrer Dauer sortiert werden. Vergessen Sie nicht, automatisch ablaufende Tests zu spezifizieren und zu implementieren.

2.3 Generische Programmierung, 1. Teil

Oft ist dieselbe Aufgabe für verschiedene Datentypen zu erledigen – z.B. das Auffinden eines bestimmten Wertes in einem sortierten Array (siehe Übung 2.9). Entsprechendes gilt für zwei strukturierte Datentypen, deren Implementierung sich manchmal nur im zugrunde liegenden Datentypus unterscheidet, z.B. eine *Warteschlange* zur Verwaltung von Variablen vom Typ `int` oder vom Typ `Vorgang`.

Eine Möglichkeit der Implementierung von Funktionen oder Datentypen fast identischer Funktionalität ist, sie einmal für einen bestimmten Datentyp zu implemen-

tieren, zu testen und anschließend per *Copy & Paste* geeignet anzupassen. Dieses Vorgehen ist natürlich zum einen fehleranfällig und zum anderen wenig komfortabel, da z.B. Fehlerkorrekturen in allen Versionen durchgeführt werden müssten.

Schablonenfunktionen und -klassen sind hier die bessere Wahl (in C++ *Templates* und in Java *Generics* genannt), das Prinzip ist elementar und einfach zu verstehen: Anstelle eines konkreten Datentyps, z.B. `int`, nimmt man einen Platzhalter, z.B. `T`, und formuliert die Funktion oder Datenstruktur unter Verwendung dieses Platzhalter-Typs, der als Typparameter verwendet wird.

Wir werden in diesem Kapitel zunächst generische Funktionen kennenlernen. Im Abschn. 6.3.1 werden wir dann generische Klassen (Schablonenklassen) vorstellen.

Generische Funktionen

Es gibt viele Funktionen, d.h. Algorithmen, die ihre Aufgabe unabhängig vom verwendeten Datentyp erledigen. Der folgende *Bubblesort*-Algorithmus – der hier beispielhaft für den Datentyp `int` angegeben ist – ist prinzipiell unabhängig davon, ob nun ganze Zahlen, Gleitkommawerte, Zeichenketten oder Vorgänge sortiert werden sollen, solange ein Vergleich zwischen diesen Daten definiert ist (vgl. Abschn. 6.2.2). Entsprechendes gilt (zumindest in C++) auch für die im *Bubblesort* verwendete Funktion `myswap` zum Austausch von zwei Werten. Das Listing zeigt links die normale Version für Integer-Werte und rechts die generische Version, die bspw. auch mit Gleitkommazahlen benutzt werden kann.

```
void myswap(int& x, int& y) {
  int t = x; x = y; y = t;
}

void bubbleSort(int beg, int stop, int f[]) {
 for (int i=beg+1; i<stop; i++) {
   for (int j=stop-1; j<i-1; j--) {
     if (f[j-1]>f[j]) myswap(f[j-1],f[j]);
} } }
```

```
template <typename T>
void myswap(T& x,T& y) {
  T t = x; x = y; y = t;
}
template <typename T>
void bubbleSort(int beg, int stop, T f[]) {
 for (int i=beg+1; i<stop; i++) {
   for (int j=stop-1; j<i-1; j--) {
     if (f[j-1]>f[j]) myswap(f[j-1],f[j]);
} } }
```

Wie man sieht, gleicht sich der Code bis auf die Verwendung des Typ-Parameters. Auch die Formulierung etwas komplexerer generischer Algorithmen bereitet keine größeren Schwierigkeiten.

Obwohl es seit Java 1.5 ein auf den ersten Blick sehr ähnliches Konzept gibt, sind die Ansätze nicht direkt vergleichbar. Für jedes Typ-Argument erzeugt der C++-Compiler eine eigene Funktion, geradeso, als hätte der Entwickler die Funktion mehrfach mit verschiedenen Parametern selbst programmiert. Aus einer Schablone entstehen also mehrere Binärcodes. Im Gegensatz dazu erzeugt der Java-Compiler aus einer Schablone auch nur einen Binärcode, büßt dafür aber einiges an Flexibilität ein (z.B., dass keine elementaren Datentypen übergeben werden können).

Funktions-Templates können wie auch *normale* Funktionen überladen werden, was bei den entsprechenden Templates der C++-Standardbibliothek fast die Regel ist. Der folgende Code skizziert ein Beispiel in C++:

```
template<typename T> void sort(T* vec, int n) { . . . }

template<typename T> void sort(vector<T>& vec) { . . . }

template<typename RndIt> void sort(RndIt first, RndIt last) { . . . }
. . .
int array[10]; vector<int> buffer(10);
. . .
Sort(array, 10); sort(buffer); sort(buffer.begin(), buffer.end());
```

Es sind drei Templates mit dem Namen `sort` definiert, die sich in der Signatur unterscheiden, und entsprechend wird in der Anwendung jeweils das passende Template ausgewählt. Darüber hinaus können auch weitere *Nicht-Template-Funktionen* gleichen Namens definiert sein, die sich natürlich auch wieder in der Signatur unterscheiden müssen. Die Auswahlregeln sind im Detail komplex, stimmen aber mit den Erwartungen überein, wenn nicht zu komplexe Situationen konstruiert werden.

Im Kap. 6 und 7 werden wir auf die generische Programmierung zurückkommen. Hier ging es schwerpunktmäßig darum, die Möglichkeiten aufzuzeigen, mit denen es gelingt, Algorithmen unabhängig vom zugrunde liegenden Datentyp zu formulieren. Auf die Unterschiede zwischen Templates und Generics werden wir in Kap. 7 noch zurückkommen.

Wir haben somit in diesem Kapitel neben der bereits bekannten funktionalen Abstraktion (Abschn. 2.1) und der Datenabstraktion (Abschn. 2.2) eine weitere Abstraktion kennengelernt. Durch die Parametrisierung von Algorithmen und Datentypen (siehe Kap. 7) können wir nochmals von Repräsentationsdetails abstrahieren.

Übungen

Übung 2.10: Implementieren Sie eine generische Funktion `compare`, die zwei Werte `a` und `b` vom gleichen Typ vergleicht. Die Funktion soll eine negative Zahl liefern, wenn a kleiner b gilt, eine positive Zahl, wenn a größer b gilt und Null sonst. Welche Voraussetzungen müssen die Datentypen erfüllen, damit `compare` dafür aufgerufen werden kann? Vergessen Sie nicht, automatisch ablaufende Tests zu spezifizieren und zu implementieren. **C++**

Übung 2.11: Implementieren Sie den Quicksort-Algorithmus aus Übung 2.9 nun als generische Funktion. Vergessen Sie wiederum nicht, automatisch ablaufende Tests zu spezifizieren und zu implementieren. **C++**

2.4 Zusammenfassung

Nur durch Abstraktion kann man komplexe Zusammenhänge verstehen, was die reale Welt im Allgemeinen betrifft und für die Software im Besonderen gilt. In diesem

Kapitel ging es im Wesentlichen um die Themen *Funktionale Abstraktion* und *Datenabstraktion*:

- *Funktionale Abstraktion* bedeutet, dass man sich Funktionen definiert, die dann auf höherer Ebene als abstrakte Anweisungen (Funktionsaufrufe) verwendet werden. Die Funktionen können von außen – aus Sicht des Benutzers der Funktionen – als *black-boxes* angesehen werden. Funktionen verwenden häufig Parameter, über die ihre Funktionalität dann im Detail spezifiziert werden kann.
- *Datenabstraktion* bedeutet, dass man sich Datentypen (Datenstrukturen) definiert, die auf höherer Ebene als Namen für komplexere Objekte verwendet werden. Diese selbst definierten Datentypen können von außen – aus Sicht des Benutzers der Datentypen – als *black-boxes* angesehen werden.

In Kap. 6 über *Abstrakte Datentypen* werden wir zeigen, wie man mit Hilfe des Klassenkonzepts beides miteinander kombinieren kann.

Bei der Entwicklung größerer Software-Systeme im Rahmen von Teams rückt das Thema der *Programmierung im Detail* häufig etwas in den Hintergrund. Aber gerade bei der Entwicklung solcher Systeme kommt es besonders darauf an, Programmteile mit Hilfe der Daten- und funktionalen Abstraktion so zu strukturieren, dass sie auch für andere Teammitglieder lesbar und beherrschbar sind.

Kapitel 3

Organisation des Quellcodes

Je größer ein Programm, desto praktischer ist es, die Quellen auf mehrere Dateien zu verteilen. Wenn der ganze Quelltext in nur einer Datei vorhanden wäre, müssten alle Teammitglieder *dieselbe* Datei gleichzeitig bearbeiten. Da Compiler dateiweise arbeiten, würde außerdem jede Änderung zu einer kompletten Neucompilierung führen. Im Abschn. 3.1 stellen wir daher zunächst vor, wie man den Code sinnvoll auf unterschiedliche Dateien aufteilt. Abschn. 3.2 zeigt anschließend, wie man inhaltlich zusammengehörende Dateien in Namensräumen zusammenfasst. Zur Auslieferung größerer Funktionsblöcke dienen Bibliotheken (*libraries*), die wir in Abschn. 3.3 ansprechen werden. Schließlich wollen wir in Abschn. 3.4 dann sehen, wie man wiederkehrende Arbeiten des Neucompilierens, Bibliothekenerstellens, der Installation etc. effizient automatisieren kann.

3.1 Modularisierung auf Dateiebene

Die ersten Programme sind Ein-Dateien-Programme (vgl. Kap. **??**). Je größer die Software wird, desto unpraktischer ist es, alles in einer Datei zu halten: Eine Quellcode-Datei ist linear, wir können uns nur vorwärts und rückwärts in der Datei bewegen, ein gezieltes Springen zu bestimmten Stellen ist schwer möglich (allenfalls durch freundliche Unterstützung der Entwicklungsumgebung). Viel praktischer ist es, Ordner und Dateien als Ordnungsmittel zu nutzen, genauso wie die heimischen Unterlagen auch in Ordnern mit Registern fachlich sortiert werden.

Aber auch aus technischer Sicht ist die Aufteilung in kleinere Portionen sinnvoll: Oft werden bei der Erweiterung eines Programms nur sehr lokal Änderungen vorgenommen. Warum die unveränderten 99% jedes Mal neu compilieren? Das kostet unnötig Zeit.

Daher werden inhaltlich verwandte Teile in einer Quellcode-Datei zu **Übersetzungseinheiten** (compilation units) zusammengefasst (Java: `.java`, C++: `.cpp` oder auch `.cxx`), damit sie einzeln compiliert werden können (wir erhalten sog.

Objektcode-Dateien, Java: `.class`, C++: `.o`). Über den Zeitstempel der compilierten Objektcode-Datei kann dann erkannt werden, ob seit der letzten Compilierung der Quelltext verändert wurde, d.h. eine Neucompilierung erforderlich ist. Sind eine Quellcode-Datei und alle weiteren Dateien, von denen sie abhängt, älter als die Objektcode-Datei, kann auf die Neucompilierung verzichtet werden.

Während wir bisher meistens Funktionen aufgerufen haben, die vorher in *derselben* Datei definiert wurden, wollen wir nun auch Funktionen aufrufen, die in *anderen* Dateien definiert wurden. Diese Dateien sollen möglichst nicht erst vorher komplett eingelesen werden müssen – das wäre dann kein Fortschritt gegenüber der Alles-in-einer-Datei-Lösung. In C++ besteht die Lösung dieses Problems in der Wiederholung der Funktionsdeklarationen in der so genannten **Header-Datei** (auch H-Datei, Definitions-Datei, Schnittstellen-Datei genannt). Diese Header-Dateien sind viel kürzer als die Quellcode-Dateien, denn sie enthalten nicht den Quelltext der Funktionen, sondern nur die Funktionsprototypen. Durch das Nachladen (engl. *include*) dieser Header-Dateien wird der gewünschte Effizienzgewinn erzielt. Problematisch an dieser Lösung ist, dass der Entwickler für die Konsistenz von Header- und Quelltext-Datei selbst verantwortlich ist (Prototyp in Header-Datei muss mit Funktionsdefinition in Quelltext-Datei übereinstimmen), es wird dem Entwickler Mehraufwand aufgebürdet. In Java hat man daher einen anderen Weg gewählt, aus den Objekt-Dateien selbst werden die darin definierten Funktionalitäten ausgelesen, der Entwickler wird nicht weiter belastet.

Durch die Aufteilung des Objektcodes auf mehrere Dateien entsteht außerdem der Bedarf für einen weiteren Schritt in der Programmerstellung, nämlich das Zusammenfügen der verschiedenen Objektcode-Dateien zu einem Programm. Diese Aufgabe übernimmt ein so genannter **Linker**. (Man spricht vom *Binden* der Objektdateien.)

Das folgende Listing zeigt ein Beispiel: In der ersten C++-Version wird in der Header-Datei `sum.h` die `summiere`-Funktion deklariert, aber nicht implementiert. In der Datei `sum.cpp` wird die Funktion dann implementiert. Auf der Java-Seite haben wir hier nur eine Datei mit der Implementierung der Funktion. Die jeweils letzte Datei *benutzt* dann die `summiere`-Funktion: Dazu muss in C++ die Schnittstelle von `summiere` durch das Einlesen der Header-Datei erst angegeben werden (`#include ¨sum.h¨`), während Java die Datei automatisch findet.[1]

```
// Header-Datei sum.h
int summiere(const int arr[], int n);
```

```
#include "sum.h"
// Source-Datei sum.cpp
int summiere(const int arr[], int n) {
  int sum = 0;
  for (int i=0;i<n;++i) sum+=arr[i];
  return sum;
}
```

```
class Summe {
  public static int summiere(int[] arr) {
    int sum = 0;
    for (int i=0;i<arr.length;++i)
      sum += arr[i];
    return sum;
  }
}
```

[1] Das liegt an den strengen Java-Namenskonventionen: Die Klasse `Summe` **muss** in einer Datei gleichen Namens (`Summe.java`) implementiert werden. Wenn die Funktion `Summe.summiere` aufgerufen wird, weiß der Java-Compiler, nach welcher Datei er suchen muss.

```
// Weitere C++−Datei, die sum(..) benutzt
#include "sum.h"
void f(...) {
  ...
  summiere(array,4);
  ...
}
```

```
// class Summe im gleichen Verzeichnis
class Anwendung {
  public static void f(..) {
    ...
    Summe.summiere(array);
    ...
} }
```

Das Beispiel stellt die Problematik dar, vereinfacht aber ein wenig, zumindest auf C++-Seite. Das Vorgehen ist fehleranfällig, weil durch verschiedene Schreibweisen (Tippfehler) in Header- und Quelltext-Datei Funktionen nicht zugeordnet werden können. Durch routinemäßiges Einbinden der zugeordneten Header-Datei in die Quelltext-Datei kann der Compiler aber die Kontrolle der Funktionsprototypen übernehmen. Der C++-Entwickler muss nicht nur zwei Dateien pflegen, er muss auch noch manuell dafür sorgen, dass dieselbe Header-Datei nicht mehrfach eingelesen wird. Dieser Aspekt wird im folgenden Abschnitt behandelt. Der nur an Java interessierte Leser kann diesen Abschnitt überspringen.

3.1.1 Allgemeiner Aufbau einer Header-Datei

C++

Die Anweisung zum Einfügen einer Header-Datei, wie z.B. `#include <iostream>`, steht häufig in verschiedenen Dateien eines Programms, die dann wieder über entsprechende *include*-Anweisungen zu einem Gesamtprogramm zusammengefügt werden. Include-Anweisungen befinden sich aber auch direkt in Header-Dateien, die wiederum selbst andere oder die gleichen Header-Dateien einbinden. Dadurch kann eine betroffene Header-Datei dann fälschlicherweise und unbeabsichtigt mehrfach in eine Quellcode-Datei eingefügt werden. Schlimmer noch: Wenn A unter anderem B einbindet, B wiederum C und C selbst dann A, so liegt ein Zyklus vor. Wenn im Laufe der Compilierung einmal A, B oder C eingebunden wurde, so werden in einer Endlosschleife immer wieder A, B und C wechselseitig eingebunden, wenn niemand den Zyklus erkennt und unterbricht. Das mehrfache Einbinden der gleichen Header-Datei und damit auch die Endlosschleife vermeidet man zweckmäßigerweise durch folgenden Aufbau einer Header-Datei:

```
#ifndef _name_h
#define _name_h
  Inhalt der Header-Datei
#endif
```

Als Beispiel dazu dient die folgende Header-Datei `Netz.h`, die Funktionen der Netzplanung bereitstellt.

```
#ifndef NETZ_HEADER
#define NETZ_HEADER
#include "Vorgang.h"
struct Netz {
/* Maximale Vorgangsanzahl im Netzplan */
```

```
  enum {MAX = 100};
  double startzeit, endzeit;
  int anzahl;
  Vorgang vorg[MAX]; // Knoten
  int nachf[MAX][MAX]; // Adjazenzmatrix
};
bool plane(Netz *netz);
void ausgabeNetzplan(const Netz* netz);
void initNetzplanAllg(Netz* netz);
#endif /* NETZ_HEADER */
```

In der Datei `Netz.h` wird der Typ `Vorgang` verwendet. Deshalb muss hier `Vorgang.h` per `#include` eingebunden werden. Wir haben die Konstante `MAX` durch eine `enum`-Anweisung innerhalb der Struktur `Netz` vereinbart, weil die Konstante nur innerhalb der Struktur `Netz`, d.h. in den Funktionen in der Datei `Netz.cpp`, benötigt wird. Wenn man allerdings die Datei `Netz.h` in andere Dateien einbindet, hat man auch außerhalb von `Netz.cpp` und `Netz.h` Zugriff auf den Wert dieser Konstanten[2] (über `Netz::MAX` vgl. Abschn. 3.2).

Der Name `NETZ_HEADER` bezeichnet eine **Präprozessor-Variable**, deren Name sich zweckmäßigerweise an den Dateinamen `Netz.h` anlehnt. Der Zusatz `_HEADER` expandiert den Namen, um eventuelle Namenskonflikte zu vermeiden. Beim ersten Durchlauf ist diese Variable zunächst nicht gesetzt, d.h. der folgende Teil bis `#endif` wird durchlaufen. Dabei wird in der zweiten Zeile durch das `#define NETZ_HEADER` die Variable gesetzt und damit verhindert, dass die Datei ein zweites Mal beim Übersetzen der gleichen Quellcode-Datei durchlaufen wird.

Mit Hilfe der soeben besprochenen Technik, die als **Include-Wächter** bezeichnet wird, werden direkte und indirekte Mehrfacheinfügungen von Header-Dateien automatisch verhindert. Alle Bibliotheksmodule sind deshalb entsprechend ausgerüstet.

Header-Dateien sollten nur reine Deklarationen, d.h. keine Definitionen, die bereits Speicherplatz belegen, enthalten. Der Grund hierfür ist, dass es ansonsten durch Einbinden der Header-Datei in verschiedenen Übersetzungseinheiten möglich wäre, dass eine Definition von z.B. einer Funktion oder einer Variablen unzulässigerweise in einer Anwendung mehrfach vorhanden ist. In Header-Dateien gehören natürlich nur solche Deklarationen hinein, die auch in mehreren Quellcode-Dateien zugleich benötigt werden. Außerdem sollte eine Implementierungs-Datei immer ihre eigene Header-Datei einbinden.

3.1.2 Aufteilung der Netzplanung auf verschiedene Dateien

Die folgende Aufteilung des Codes in mehrere Dateien erscheint sinnvoll:

1. Die Anwendung mit der `main`-Funktion (hier bestehend aus dem Aufruf der einzelnen Tests);
2. die Testfunktionen (werden üblicherweise nicht an den Kunden ausgeliefert);

[2] Durch die Konstantendefinition `const int MAX=100; struct Netz{...};` hätte man natürlich auch die Konstante vereinbaren können. `MAX` wäre dann allerdings in jeder Quellcode-Datei, die `Netz.h` einbindet, definiert gewesen und hätte so unnötig Speicherplatz belegt.

3. die Funktionen zur Netzplanung selbst (können in anderen Projekten wiederverwendet werden).

Aufteilung der Netzplanung: Wir benutzen zwei Datenstrukturen für die Netzplanung: `Netz` und `Vorgang`. Generell gilt: Solange wir mehrere Datenstrukturen in einer Datei belassen, lässt sich die eine nicht ohne die andere wiederverwenden. In Java wurden wir gezwungen, für jede eine eigene Datei anzulegen (für jede Klasse eine eigene Datei); diesen Schritt ziehen wir nun auch für C++ nach (Aufspaltung von `Netzplanung.h` in `Netz.*` und `Vorgang.*`). Sind die Datenstrukturen auf mehrere Dateien aufgeteilt, so ist es sinnvoll, auch die Gliederung der Funktionalität danach auszurichten, d.h. Funktionen, die sich auf Vorgänge beziehen, in die Vorgangs-Datei zu verschieben. Das Ergebnis dieser Aufteilung zeigt in Ausschnitten Listing 3.1 für das `Netz` (für C++: Header-Datei `Netz.h` siehe Seite 57).

C++/Java 3.1: Dem `Netz` zugeordnete Funktionalität.

```
#include "Netz.h"

#include <iostream>//wg. Bildschirmausgabe
using namespace std;

void berechneVorwaerts(int v, Netz *netz) {
  /*...*/
}
void berechneRueckwaerts(int v, Netz *netz){
  /*...*/
}
bool istDurchfuehrbar(Netz *netz) {
  /*...*/
}
bool plane(Netz *netz) {
  /*...*/
}
void initNetzplanAllg(Netz* netz){
  /*...*/
}
```

(netzplanung/v3-makefiles/Netz.sht)

```
public class Netz {
// Daten

public static void berechneVorwaerts(int v,
                                      Netz netz)
{ /* ... */ }
public static void berechneRueckwaerts(int v,
                                      Netz netz)
{ /* ... */ }
public static boolean istDurchfuehrbar(
                                      Netz netz)
{ /* ... */ }
public static boolean plane(Netz netz) {
  /* ... */
}
public static void initNetzplanAllg(
                                      Netz netz)
{ /* ... */ }

}
```

(netzplanung/v3-build/Netz.sht)

Trennung von Funktionalität und Test Entsprechend verfahren wir mit den Tests. Dabei muss die Netzplanungs-Funktionalität natürlich bekannt und verfügbar sein, was wir durch das Einbinden der entsprechenden Header-Dateien (C++) oder die Platzierung der Quellcode-Dateien im gleichen Verzeichnis (Java) erreichen können (vgl. Abschn. 3.2 bei mehreren Verzeichnissen).

Wir haben die Funktion `initNetzplanTest5` benutzt, um ein Beispielnetz für unsere drei Tests aufzubauen (fünf Vorgänge aus Bild **??**). Weil diese Funktion ausschließlich für die Tests des Netzplans verwendbar ist, gehört sie eindeutig zur Test-Funktionalität. Die Funktion `initNetzplanAllg` (Zurücksetzen eines Netzes auf einen definierten Anfangszustand) kann jedoch zur Initialisierung beliebiger Netzpläne verwendet werden und gehört deshalb in die Dateien zur Netzplanung. Ihre Deklaration muss öffentlich sein (wie z.B. auch `plane`), da sie aus den Tests (aus der Funktion `initNetzplanTest5`) aufgerufen wird.

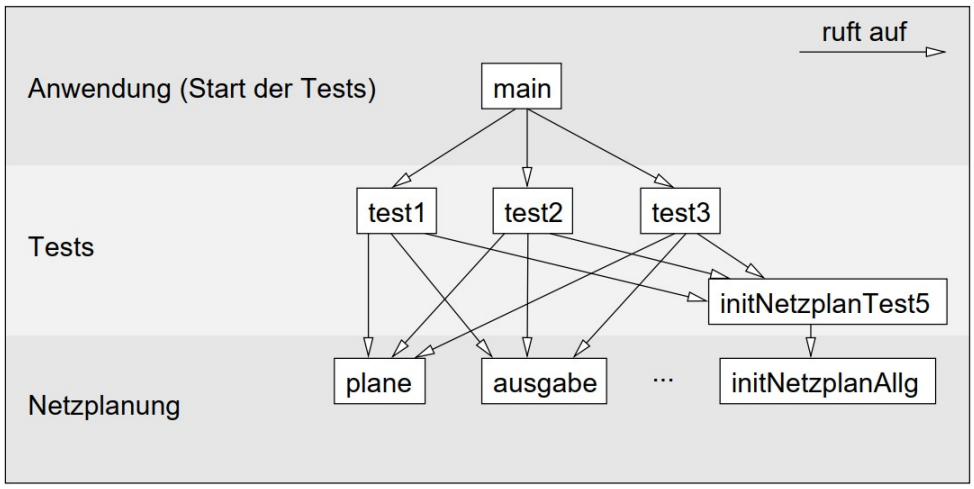

Bild 3.1: Aufruf-Abhängigkeiten der Netzplanungsfunktionen

Bild 3.1 zeigt die Situation schematisch: Die Anwendung (Test-Applikation) verwendet die Funktionen `test1-test3`, diese müssen daher öffentlich bekannt sein. Die Funktion `initNetzplanTest5` hingegen wird von außen, d.h. von der Test-Applikation, nicht benutzt. Diese Funktion muss nicht öffentlich sein und braucht daher in `Test.h` nicht vorhanden zu sein (C++). In Java erzielen wir diesen Effekt, indem wir das Schlüsselwort **public** weglassen (Details folgen in Kap. 6). Da wir nicht alle Funktionen in die Header-Datei aufnehmen, reduzieren wir die von einem Benutzer wahrgenommene Komplexität: Es sind weniger Funktionen zu verstehen. Gleichzeitig verhindern wir, dass die Funktion von anderen *wiederverwendet* wird. Listing 3.2 zeigt die Funktionalität, die wir den Tests zugeordnet haben (am Beispiel C++).

C++ 3.2: Header-Datei und Ausschnitt aus Quellcode-Datei

```
#ifndef TEST_HEADER
#define TEST_HEADER

bool testFall1();
bool testFall2();
bool testFall3();

#endif /*#ifndef TEST_HEADER */
```

(netzplanung/v3-makefiles/Test.h)

```
#include "Test.h"
#include "Netz.h"
void initNetzplanTest5(Netz* netz){
  netz->anzahl = 5;
  initNetzplanAllg(netz);
  /* ... */                    }
bool testFall1() { /* ... */ }
bool testFall2() { /* ... */ }
bool testFall3() { /* ... */ }
```

(netzplanung/v3-makefiles/Test.sht)

3.2 Strukturierung jenseits von Dateigrenzen

Wird die Wiederverwendung von Code intensiv praktiziert, so ist es nur eine Frage der Zeit, bis ein Entwickler Quelltext eines anderen benutzt, der eine Klasse oder

Funktion exakt genauso benannt hat wie er selbst. Man spricht dann von einem **Namenskonflikt** (engl. *name clash*). Das Problem besteht für den Linker nun darin herauszufinden, welche der verschiedenen Implementierungen bei einem Funktionsaufruf referenziert wurde.

3.2.1 Namensräume und Pakete

Natürlich bieten wieder beide Sprachen hierfür eine Lösung: Es muss ein weiteres Strukturierungsmittel jenseits der Dateiebene her, sodass bspw. jeder Entwickler einen eigenen Bereich (Java: Paket (engl. *package*), C++: Namensraum (engl. *namespace*)) für seine Bezeichner besitzt. Wenn Entwickler A nun Code von Entwickler B nutzen will, muss er zusätzlich den Bereich angeben, aus dem der Code stammt. Die Ansprache von Variablen, Funktionen und Klassen wird damit um die Spezifikation des Bereiches erweitert. In beiden Sprachen werden diese Bereichsnamen als Präfix vor den normalen Bezeichner gehängt, in C++ getrennt durch zwei Doppelpunkte und in Java durch einen einfachen Punkt (sog. **Bereichsauflösungsoperatoren**).

Die Wege, die in C++ und Java für die Organisation der Bereiche beschritten werden, sind allerdings unterschiedlich. Java nutzt aus, dass der Entwickler sinnverwandten Code ohnehin in einem Verzeichnis organisiert, daher wird der Verzeichnispfad genutzt, um die Namensbereiche (Pakete) zu definieren. Weil Ordner hierarchisch organisiert sind, gilt dasselbe dann auch für Java-Pakete. Ein positiver Nebeneffekt ist, dass jeder Entwickler (und auch der Compiler) weiß, dass es sich um eine Klasse `Kunde` im Verzeichnis `com/software/data` handelt,[3] wenn er einen Bezeichner `com.software.data.Kunde` vorfindet. Die Klassen in Java benutzen wir bisher nur als eine Art Klammer für eine Menge von Funktionen, sie bilden somit auch eine Art Namensraum.

In C++ sind die Grenzen von Namensräumen nicht an Verzeichnisse geknüpft. Stattdessen können (auch in ein und derselben Datei) beliebig Namensräume gekennzeichnet werden: Alle Definitionen innerhalb dieses Bereiches gehören dann dem entsprechenden Namensraum an.

Das folgende Beispiel und Tabelle 3.1 illustrieren die wesentlichen Elemente: In C++ wird der Code eines bestimmten Namensbereichs durch `namespace x { ... }` eingeklammert. In Java gehören immer alle Definitionen aus einer Datei einem Bereich an. Dieser wird durch die Direktive `package x;` am Anfang der Datei bekannt gemacht. Damit das Beispiel funktioniert, müssen die Java-Dateien in Unterverzeichnissen entsprechend dem Paketnamen `x` bzw. `y` untergebracht werden.

[3] Java-Namenskonvention: Paketnamen beginnen immer mit einem Kleinbuchstaben, Klassennamen mit einem Großbuchstaben.

```cpp
// Source-Datei f1.cpp
namespace x {
  void f() { ... }
}
```

```cpp
// Source-Datei f2.cpp
namespace y {
  void f() { ... }
}
```

```java
package x;
class FKlasse {
  public static void f() { ... }
}
```

```java
package y;
class FKlasse {
  public static void f() { ... }
}
```

Tabelle 3.1: Überblick über Namensräume/Pakete

Aktion	C++	Java
Zuordnung zu einem Namensraum/Erweiterung eines Namensraumes	Umschließen des Codes in `namespace name { ... }`	Nennung der Paketzugehörigkeit am Anfang der Datei `package name;`
Einführung eines Zweitnamens	Durch `namespace a = b;` werden beide Namensräume gleichgesetzt.	nicht möglich
Verwendung aller Bezeichner eines Namensraumes	`using namespace a::Klasse;` oder `using namespace a;`	`import a.Klasse` oder `import a.*` für alle Klassen

Neben der (eher umständlichen) direkten Ansprache über den voll qualifizierten Namen (Bereichsname, Klasse und Bezeichner: `x::f()` bzw. `x.FKlasse.f()`) gibt es die Möglichkeit, einmalig die Verwendung bestimmter Bereiche anzukündigen, sodass danach die Verwendung des vollständigen Namens (inklusive Namensraum) entfallen kann. In C++ geschieht dies durch die Direktive `using namespace Y` und in Java durch `import y.FKlasse` für einen einzelnen Bezeichner oder `import y.*` für alle Bezeichner im Paket `y`. Für Java ist zu beachten, dass alle Bezeichner, auf die wir von einem anderen Paket aus zugreifen wollen, öffentlich sein müssen (`public`). Wir werden die Bedeutung von `public` detaillierter in Kap. 6 erläutern.

```cpp
using namespace y;
void g() {
  x::f(); // f() aus Namensraum x
  f(); // f() aus Namensraum y
}
```

```java
import y.*;
class Anwendung {
  public static void g() {
    x.f(); // f() aus Namensraum x
    f(); // f() aus Namensraum y
  }
}
```

Ein wichtiger Namensraum in C++ ist `std`, in dem die Standard Template Library zu finden ist. Die Laufzeitumgebung von Java ist in viele Pakete unterteilt, `java.lang` für die elementaren Sprachelemente (muss nicht importiert werden), `java.io` für Ein/Ausgabe, `java.net` für Netzkommunikation etc.

3.3 Bibliotheken

Unser Netzplanungsbeispiel ist noch sehr klein. Es besteht derzeit nur aus den Quellcode-Dateien für `Netz`, `Vorgang` und `Test`. Trotzdem handelt es sich um einen gut wiederverwendbaren Baustein, den wir möglichst leicht in andere Projekte integrieren wollen. Dazu müssen nur die entsprechenden Objektcode-Dateien vom Linker eingebunden werden. Wenn die Funktionalität aber durch zehn oder gar durch Hunderte von Dateien erbracht wird, ist es sehr mühselig, jedesmal alle erforderlichen Dateien explizit anzugeben. Deshalb werden inhaltlich zusammengehörende Dateien in Bibliotheken zusammengefasst. Lediglich der Bibliotheksname muss dann an den Linker übergeben werden.

Eine **Bibliothek** (engl. *library*) oder ein Archiv (engl. *archive*) bezeichnet eine Sammlung von Funktionalitäten für zusammengehörende Aufgaben. Bibliotheken sind im Unterschied zu Programmen keine eigenständigen Einheiten, sondern Hilfsmodule, die Programmen zur Verfügung stehen.

Zu unterscheiden ist zwischen *statischen* und *dynamischen Bibliotheken*. **Statische Bibliotheken** werden nach dem Compilieren durch den Linker sofort mit dem ausführbaren Programm verbunden, indem der Linker aus den Dateien in der Bibliothek die benötigten Funktionen, globale Variablen etc. heraussucht.

Dynamische Bibliotheken werden dagegen erst bei Bedarf in den Arbeitsspeicher geladen und mit dem ausführbaren Programm verbunden. Dadurch muss eine Bibliothek, die von mehreren Programmen genutzt wird, nur einmal im Speicher gehalten werden. Dies ist vorteilhaft, wenn die Bibliotheken insgesamt sehr groß sind und von vielen Prozessen gleichzeitig verwendet werden. Trifft ein Programm auf den Verweis zu einer Funktion, die noch nicht eingebunden wurde, dann wird ein Laufzeitlinker aktiviert. Dieser sucht die Funktion in den dynamisch in das Programm eingebundenen Bibliotheken, fügt die Adresse am Aufrufpunkt ein und führt die Funktion erstmalig aus. Bei jedem weiteren Aufruf der Funktion ist dann deren Adresse vorhanden, sodass das Unterprogramm direkt aufgerufen wird. Die Ausführungszeit, insbesondere die Startzeit eines Programms, ist hier geringfügig erhöht.

Beim Windows-Betriebssystem wird eine dynamische Bibliothek als *DLL* (*Dynamic Link Library*, Dateiendung `.dll`) bezeichnet. Auf Unix-artigen Betriebssystemen (Unix, Linux usw.) spricht man von *shared libraries* (Dateiendung `.so`). In Java erfolgt das Laden des Codes einer Klasse immer dynamisch, d.h. beim erstmaligen Verwenden; statische Bibliotheken gibt es nicht.

Erzeugen von Bibliotheken

Das folgende Listing zeigt, mit welchen Kommandos die compilierten Objekt-Dateien zu einer Bibliothek zusammengeführt werden können (die C++-Beispiele beziehen sich auf Linux/Unix). Alle Werkzeuge erwarten nach einigen Kommandozeilen-Optionen den Namen der Bibliothek und dann die Objekt-Dateien, die der Bibliothek angehören sollen.

```
% statische Bibliothek netzplanung.a
ar -cur netzplanung.a Netz.o Vorgang.o
% dynamische Bibliothek netzplanung.so
ld -shared netzplanung.so Netz.o Vorgang.o
```

```
% Java-Archiv netzplanung.jar
jar cvf netzplanung.jar \
      Netz.class Vorgang.class
```

Mit dem folgenden Kommando kann man sich zu jeder Bibliothek anzeigen lassen, welche Funktionalität sie bereitstellt:

```
# Dateien von libnetzplaning.a anzeigen
ar -t libnetzplanung.a
# externe Symbole der Bibliothek listen
nm -g libnetzplanung.a
```

```
% Java-Archiv netzplanung.jar
jar tvf netzplanung.jar
```

Dabei sind Java-Archive nichts anderes als gezippte Archive mit class-Dateien, sie könnten auch mit einem normalen Entpacker ausgepackt werden. Das Introspektions-Konzept von Java erlaubt es dann, die ausgepackten Klassen nach der bereitgestellten Funktionalität zu befragen. Bei Java ist es wichtig, auch die Verzeichnisstruktur und nicht nur die `.class`-Dateien einzupacken: Wie bereits erwähnt, sucht der Klassenlader immer in einem Verzeichnis, das mit dem Paketnamen korrespondiert. Daher muss diese Verzeichnisstruktur bewahrt bleiben.

Benutzung von Bibliotheken

Bei der Erstellung von Programmen, die die Bibliothek benutzen, muss bekanntgegeben werden, welche Funktionen (mit welcher Syntax) in der Bibliothek vorhanden sind. Das geschieht durch Einbinden einer Header-Datei oder Import-Zeilen für die Java-Klasse. Beim Linken muss die Bibliothek selbst als Quelle genannt werden (C++: eine Compiler/Linkeroption `-l library` je Bibliothek, Java: Archive dem Klassenpfad, durch Doppelpunkte getrennt, hinzufügen `-classpath .:library1.jar:library2.jar`).

```
% Compilieren + Linken mit libnetzplanung.a/so
g++ newprogram.cpp -lnetzplanung
% Zwang zur statischen Bindung
g++ -static Test.cpp -lnetzplanung
```

```
% Compilieren mit netzplanung.jar
 javac -classpath .:netzplanung.jar Test.java
```

Java stellt darüber hinaus Mechanismen zur Versiegelung von Archiven bereit, wodurch sichergestellt wird, dass alle Klassen eines Paketes aus demselben `.jar`-Archiv stammen müssen. Außerdem können Archive elektronisch unterschrieben (signiert) werden, sodass der Anwender sich vergewissern kann, dass ein Archiv seit der Auslieferung nicht verändert wurde [58].

Gestaltungsrichtlinien für Bibliotheken

Wenn man eine Bibliothek benutzt, so kann diese selbst wieder Funktionalität aus anderen Bibliotheken referenzieren, die ihrerseits wiederum andere Bibliotheken benötigen. Üblicherweise spielt die Reihenfolge, in der Bibliotheken dem Linker zur Verfügung gestellt werden, eine Rolle: Eine neue Bibliothek darf nur Funktionalität benutzen, die in zuvor angegebenen Bibliotheken bereitgestellt wurde. Andernfalls

beendet der Linker seinen Dienst mit der Aussage, dass er eine bestimmte Funktionalität nicht finden konnte.

Aber unabhängig davon, ob der verwendete Linker diese Reihenfolge erzwingt oder nicht: Bibliotheken sind (größere) Software-Bausteine, die eine in sich abgeschlossene Funktionalität bereitstellen. Selbst wenn die Abläufe innerhalb der Bibliothek unstrukturiert sind (potenziell kann jede Funktion jede andere Funktion aufrufen), so sollte die Benutzungs-Beziehung zwischen Bibliotheken hierarchisch aufgebaut sein. Zum einen ist ein System aus drei Komponenten A, B und C schwerer zu verstehen, wenn zum Verständnis von A auch B bekannt sein muss, für B wiederum C und für C bereits A. Zum anderen ist eine so enge Kopplung der drei Komponenten A, B und C ein Zeichen dafür, dass sie sehr eng zusammenarbeiten und lieber gemeinsam in einer Bibliothek untergebracht werden sollten.

Für C++ soll noch einmal erwähnt werden, dass zusammen mit der Bibliothek auch die Header-Dateien ausgeliefert werden müssen, um die Bibliothek nutzen zu können. Es bietet sich manchmal an, für die Bibliothek eine einzige, neue Header-Datei zusammenzustellen, statt eine größere Menge kleinerer Header-Dateien zu behalten.

3.4 Build-Management

Bereits bei wenigen Dateien wird die manuelle Compilierung, Pflege der Abhängigkeiten, Erstellung von Bibliotheken etc. sehr umständlich. Der Lebenszyklus der Installation könnte durch folgende Phasen zusammengefasst werden:

- **Konfiguration** (configure): Ermittlung, ob alle benötigten Werkzeuge (Compiler, Linker, Bibliotheken etc.) in den benötigten Versionen vorliegen. Für diesen Schritt gibt es unter den verschiedenen Plattformen unterschiedliche Unterstützungswerkzeuge, die wir hier in diesem Buch aber nicht weiter betrachten wollen (z.B. unter Unix das **autoconf**-Tool).
- **Compilierung** (compile): Compilierung aller Quellen und Erstellen der Bibliotheken.
- **Test** (test): Ausführen aller erstellten Tests, um die Korrektheit der Version zu überprüfen. So kann rasch festgestellt werden, ob sich gegenüber der vorigen Version eine ungewollte Änderung der Semantik ergeben hat.
- **Installation** (install, deploy): Installation der Software, zum Beispiel Kopieren von Bibliotheken in spezielle Verzeichnisse, in denen das Betriebssystem nach Bibliotheken sucht.
- **Säuberung** (clean): Entfernen der Zwischenergebnisse (so können Objektcode-Dateien gelöscht werden, wenn sie in eine Bibliothek eingefügt wurden).
- **Deinstallation** (uninstall, undeploy): Entfernen der zuvor installierten Software.

Bei der Entwicklung werden diese Schritte vielfach ausgeführt, nicht nur für jede ausgelieferte Version. Daher ist es sinnvoll, diese Schritte zu automatisieren. Einige dieser Schritte werden von Entwicklungsumgebungen unterstützt, die ihrerseits auf Werkzeuge wie **make** [38, 42] oder **ant**; siehe `ant.apache.org/manual` bzw. [20, 28]

zurückgreifen. Für diese Werkzeuge wird in einem Makefile oder einem Buildfile (allgemein: Projektdatei) festgehalten, was für die oben genannten (und möglicherweise weitere) Aufgaben zu tun ist.

Der Aufbau solcher Projektdateien folgt meistens folgendem Schema (wobei die exakte Syntax je nach verwendetem Werkzeug variiert):

```
Ziel: Abhängigkeiten
    Anweisungsliste
```

Jeder Arbeitsschritt (z.B. Compilierung einer Datei, Bibliothek erstellen etc.) wird durch einen Namen charakterisiert (`Ziel`) und über eine Folge von Anweisungen erreicht (`Anweisungsliste`). Dabei müssen bestimmte andere Arbeitsschritte bereits vorher erledigt worden sein, damit die Anweisungen den gewünschten Erfolg bringen können; zum Beispiel müssen vor der Erstellung der Bibliothek alle Objekt-Dateien erzeugt worden sein. Vorher abzuarbeitende Arbeitsschritte oder zu erreichende Ziele werden als `Abhängigkeiten` aufgeführt. Das Werkzeug wird dann mit den Zielen aufgerufen, die der Benutzer erreichen möchte, z.B. `make install` oder `ant deploy`. Daraufhin werden alle notwendigen Schritte automatisch ausgeführt.

Im folgenden Listing wird links ein Makefile-Ausschnitt für das Erzeugen einer Bibliothek gezeigt. Das Ziel ist der Name der Bibliothek, deren Aktualität von den Objekt-Dateien `Netz.o` und `Vorgang.o` abhängig ist (Abhängigkeiten). Die Anweisungsliste besteht hier aus nur einer Zeile, die wir schon im Abschn. 3.3 kennengelernt haben. Rechts ist ein Ausschnitt aus dem in der Java-Welt stärker verbreitetem Buildfile im XML-Format angegeben. Das Ziel (engl. *target*) heißt hier `jar`, es hängt von der erfolgreichen Ausführung eines anderen Ziels ab: `compile`. Das Kommando zum Erzeugen der Bibliothek ist `ant` bekannt, es wird über die XML-Tags nur spezifiziert, wie der Dateiname lautet, wo die Objekt-Dateien zu finden sind etc. Der exakte Aufruf zum Erstellen der Bibliothek wird von `ant` dann selbst erstellt und muss nicht, wie im Makefile, explizit angegeben werden.

```
# Makefile-Ausschnitt für C++
netzplanung.a : Netz.o Vorgang.o
    ar -cur netzplanung.a Netz.o \
                          Vorgang.o
```

```
# Makefile-Ausschnitt für Java
netzplanung.jar : Netz.class Vorgang.class
    jar cvf netzplanung.jar Netz.class \
                        Vorgang.class
```

```
<! Ausschnitt von build.xml für Java !/>
<target name="jar" depends="compile">
    <jar destfile="netzplanung.jar"
        basedir="./"/>
</target>
```

Typische Zielnamen (neben den Namen von Bibliotheken oder Objekt-Dateien) sind in der obigen Aufzählung in Klammern bereits aufgeführt.

3.4.1 Abhängigkeiten

Große Programme bestehen im Allgemeinen aus vielen Bausteinen. Eine Übersetzungseinheit in C++ besteht aus Header- und Quellcode-Datei, eine Bibliothek enthält viele Übersetzungseinheiten, ein Programm benutzt viele Bibliotheken. Bei einer

Änderung einer einzelnen Datei wird die Neucompilierung aller Übersetzungseinheiten erforderlich, die diese Datei benutzen, die Neuerstellung aller Bibliotheken, die eine dieser neu compilierten Einheiten einbinden, und schließlich die Neucompilierung des Programms selbst.

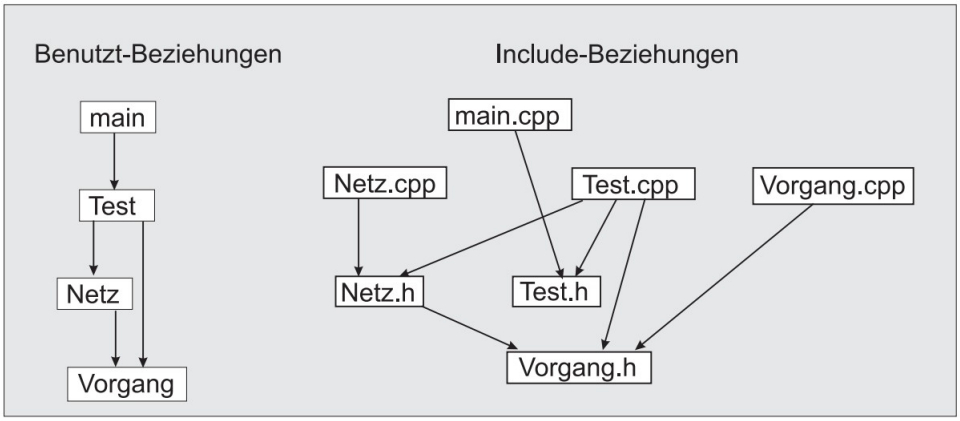

Bild 3.2: Abhängigkeiten der Netzplanungsdateien

Bereits in unserem Netzplanungs-Beispiel haben wir einige Abhängigkeiten zwischen den Dateien wie in Bild 3.2 für die C++-Version gezeigt: vier Übersetzungseinheiten in sieben Dateien, die sich in einer hierarchischen Abhängigkeit befinden:

- Modul `main` *benutzt* das Modul `Test`,
- Modul `Test` *benutzt* die Module `Netz` und `Vorgang`,
- Modul `Netz` *benutzt* das Modul `Vorgang`,
- Datei `main.cpp` *bindet* die Datei `Test.h` ein,
- Datei `Test.cpp` *bindet* die Dateien `Test.h`, `Netz.h` und `Vorgang.h` ein,
- Datei `Netz.cpp` *bindet* die Datei `Netz.h` ein,
- Datei `Vorgang.cpp` *bindet* die Datei `Vorgang.h` ein,
- Datei `Netz.h` *bindet* die Datei `Vorgang.h` ein.

Wir sind dabei davon ausgegangen, dass wir auch Tests für `Vorgang` implementiert haben, sodass das `Test`-Modul das Modul `Vorgang` ebenfalls benötigt und die Datei `Test.cpp` die Schnittstelle von `Vorgang` einbindet.

Die in einer Projektdatei anzugebenden Abhängigkeiten können direkt aus diesem Graphen übernommen werden. Die C++-Objekt-Datei `Netz.o` hängt von der Quellcode-Datei `Netz.cpp` und der zugehörigen Header-Datei `Netz.h` ab, aber auch von der Datei `Vorgang.h`, die innerhalb von `Netz.h` eingebunden wird. Die in einem Makefile einzutragenden Abhängigkeiten sehen wie folgt aus:

```
main.o:  main.cpp Test.h
Test.o:  Test.cpp Test.h Netz.h Vorgang.h
Netz.o:  Netz.cpp Netz.h Vorgang.h
Vorgang.o: Vorgang.cpp Vorgang.h
```

Man kann sich leicht vorstellen, dass die Pflege dieser Abhängigkeiten sehr aufwändig ist. Als Entwickler möchte man sich mit solchen Aufgaben möglichst nicht beschäftigen müssen. Java ist in dieser Beziehung *von Haus aus* sehr entwicklerfreundlich ausgestattet, so ist es nicht einmal erforderlich, für die Compilierung einer Anwendung explizit alle anderen benötigten Java-Dateien zu compilieren. Der Java-Compiler übersetzt automatisch auch alle anderen Java-Quellen, die die Anwendung benötigt. Entsprechend spielt die Modellierung der Abhängigkeiten wie im Beispiel oben für Java-Entwickler keine Rolle.

Das folgende Beispiel zeigt, wie man den GNU C++ Compiler (`gcc.gnu.org`) die Abhängigkeiten selbst erzeugen und an ein bestehendes Makefile anhängen lässt. Das Ziel `dep` hat in diesem Beispiel selbst keine Abhängigkeiten, sondern wird jedes Mal erzeugt (die `#include`-Abhängigkeiten könnten sich ja in einer beliebigen Datei verändert haben). Zunächst wird das Makefile bis zu einer Zeile `# Abhängigkeiten` nach `Makefile.new` kopiert. Dann hängt der C++-Compileraufruf (mit Parameter `-MM`) die selbstständig gefundenen Abhängigkeiten ans Ende von `Makefile.new` und ersetzt im letzten Schritt das bisherige Makefile durch das neue.

```
dep:
  sed '/^# Abhaengigkeiten /q' Makefile > Makefile.new
  g++ -MM $(SRCS) >> Makefile.new
  cp -f Makefile.new Makefile
# Abhaengigkeiten Diese Zeile nicht entfernen.
```

3.4.2 Die Projektdatei

Während Aufgabe und Arbeitsweise sehr ähnlich sind, unterscheidet sich der Aufbau der Projektdateien von `make` (Projektdatei heißt `Makefile`) und `ant` (Projektdatei heißt`build.xml`). Grundsätzlich ist es möglich, auch für Java-Programme ein `Makefile` zu erzeugen. Warum gibt es dann überhaupt ein weiteres Werkzeug? Java Code läuft (*plattformunabhängig*) auf jedem Rechner, der über eine Java Virtual Machine (JVM) verfügt. Das Konzept von `make`, das aus der Unix-Welt stammt, baut auf der Verwendbarkeit zahlreicher kleiner Werkzeuge und Kommandos auf, die aber nicht unbedingt auf jedem anderen Ziel-Betriebssystem existieren müssen. Dann wäre der Java-Code (dank JVM) zwar portabel, aber die Projektdatei nicht. Darum finden sich in der Projektdatei von `ant` auch keine echten Kommandos, wie wir sie auf der Kommandozeile eingeben, sondern über XML-Tags werden die Kommandos spezifiziert.

Betrachten wir die Bestandteile einer Projektdatei im Einzelnen:

- Aufbau Default-Ziel:
  Das erste Argument von `make` oder `ant` gibt das Ziel an, das erzeugt werden soll. Was aber geschieht, wenn kein Argument angegeben wird? `make` nimmt das erste

Ziel (im Beispiel: `all`), und bei `ant` wird das Standardziel in Projekt-Tag angegeben (`default="compile"`). Unsere Projektdatei beginnen wir daher mit

```
all : compile

hier alle weiteren Ziele und
Definitionen eintragen

# Abhaengigkeiten
...
```

```
<?xml version="1.0" encoding="ISO-8859-1">
<project name="Netzplan" basedir="."
                    default="compile">

hier alle weiteren Ziele und
Definition eintragen

</project>
```

- Umgebungsvariablen:
  In einer Projektdatei kommen immer wieder dieselben Dateinamen, Kommandozeilenoptionen oder Bezeichner vor. Um im Falle einer Umbenennung nicht an vielen Stellen eine Ersetzung durchführen zu müssen, bietet es sich an, Variablen einmalig zu belegen und dann zu benutzen. Im Folgenden werden zwei Variablen `SRCDIR` und `DESTDIR` angelegt. Sie können durch `$(SRCDIR)` (im `Makefile`) oder `${srcdir}` (in `build.xml`) referenziert werden. Außerdem definiert das `Makefile` über Umgebungsvariablen, wie und mit welchen Parametern Linker (`LINK`) und C++-Compiler (`CC`) aufgerufen werden.

```
SRCDIR = netzplan/
DESTDIR = usr/lib/
C++FLAGS=-Wall -g -c
LDFLAGS = -g -Wall
CC=c++ $(C++FLAGS)
LINK=c++
```

```
  <property name="srcdir" value="."/>
  <property name="destdir" value="bin"/>
```

- Ziel `compile`:
  Da es bei unserer Netzplanung nur die Tests als ablauffähigen Quelltext gibt, gilt es beim Ziel `compile`, diese Quelle zu compilieren. Während der Java-Compiler selbst herausfindet, welche Quelltexte dazu noch zusätzlich übersetzt werden müssen, haben wir bei C++ etwas mehr Arbeit: Die Umgebungsvariable `SRCS` enthält alle Quellcode-Dateien, die übersetzt werden müssen, `OBJS` erzeugt daraus eine Liste von Objekt-Dateien (dieselben Dateien mit Endung `.o` statt `.cpp`). Das `compile`-Ziel hängt nur von `NetzTest` ab (`PROG=NetzTest`), wobei `NetzTest` von allen Objekt-Dateien abhängt. `make` hat eingebaute Regeln für die Überführung von `.cpp` in `.o`-Dateien, kann anhand dieser Regeln die Objekt-Dateien erstellen. Es bleibt nur die Anweisung für das Linken der Gesamtanwendung anzugeben.

```
PROG = NetzTest
SRCS = main.cpp Netz.cpp Vorgang.cpp \
     Test.cpp
OBJS=$(SRCS:.cpp=.o)
compile: $(PROG)

$(PROG): $(OBJS)
 $(LINK) $(LDFLAGS) -o $(PROG) $(OBJS)
```

```
  <target name="compile">
    <javac srcdir="${srcdir}"
          destdir="${destdir}"
          classpath="${destdir}"/>
  </target>
```

- Ziel `test`:
  Hier wird beschrieben, welche Kommandos erforderlich sind, um die Tests auszuführen, in unserem Fall wird die Datei (bzw. die Java-Klasse) `NetzTest` gestartet.

Dieses Ziel hat eine Abhängigkeit: Wird das Ziel `test` angefordert, muss vorher `NetzTest` compiliert worden sein.

```
test: compile
  NetzTest
```

```
<target name="test" depends="compile">
  <java classpath="${destdir}$"
       classname="NetzTest"/>
</target>
```

- Ziel `install`/`deploy`:
  Hier wird beschrieben, wie die erstellte Software, z.B. eine Bibliothek, in das Gesamtsystem installiert wird, z.B. werden Header-Dateien in das Standardverzeichnis `usr/include` kopiert oder ein Java-Archiv in ein `lib`-Verzeichnis im `CLASSPATH`. Quellen, die sich auf Tests beziehen, werden nicht mit installiert.

```
INC_INSTALL = Vorgang.h Netz.h
install: compile
  cp -f $(INC_INSTALL) /usr/include/
  cp -f $(PROG) /usr/bin/
```

```
<target name="deploy" depends="jar">
  <copy file="netzplanung.jar"
       todir="${destdir}"/>
</target>
```

- Ziel `clean` und Ziel `uninstall`:
  Nach der Installation können Objekt-Dateien und lokale Kopien gelöscht werden. Zum Entfernen der Software sind die bei `install` durchgeführten Schritte zurückzunehmen.

```
clean:
  rm -f NetzTest *.o *~ Makefile.new

uninstall:
  @ for f in $(INC_INSTALL); do \
      rm -f /usr/include/$$d; done
  rm -f /usr/bin/$(PROG);
```

```
<target name="clean">
 <delete dir="${bindir}"/>
 <delete file="netzplan.jar"/>
</target>
<target name="undeploy">
 <delete file="../netzplan.jar"/>
</target>
```

Während es sich bei `build.xml` um normales XML handelt, gibt es beim `Makefile` eine Besonderheit zu beachten: Die Zeilen, die die auszuführenden Anweisungen zu einem Ziel enthalten, **müssen** mit einem Tabulator beginnen!

Oft entfällt bei Einsatz einer entsprechenden Entwicklungsumgebung die manuelle Erstellung eines Projektfiles völlig, *Eclipse* z.B. erzeugt es automatisch. Abschließend zeigt Listing 3.3 noch einmal ein Beispiel für eine komplette Projektdatei.

3.4.3 Übungen

Übung 3.1: Erstellen Sie wenigstens zwei Quellcode-Dateien, auf die Sie einfache Funktionen für die Grundrechenarten Summe, Differenz, Produkt etc. aufteilen. Entwickeln Sie parallel dazu einfache Tests. Entwerfen Sie eine Projektdatei, die die mathematischen Funktionen zu einer Bibliothek zusammenfasst. Sorgen Sie dafür, dass zum Testprogramm die Bibliothek hinzugelinkt wird.

make/ant 3.3: Projektdatei für Netzplanung.

```
LIB = libNetz
PROG = NetzTest
C++FLAGS=-Wall -g -c
LFl = -g -Wall -L./
CC=g++ $(C++FLAGS)
LINK=g++ -static
Lsh = ld -shared
Lst = ar -cur

# Alle Quellcodedateien der Anwendung
SRCS = main.cpp Netz.cpp \
        Vorgang.cpp Test.cpp

# Regel
OBJS=$(SRCS:.cpp=.o)
LIB_OBJ_FILES = Netz.o Vorgang.o Test.o

# Erstellung des Anwendungsprogramms
$(PROG): $(OBJS) $(LIB).a $(LIB).so
      $(LINK) $(LFl) -o $(PROG) \
           main.o -lNetz

$(LIB).a: $(LIB_OBJ_FILES)
      $(Lst) $(LIB).a $(LIB_OBJ_FILES)

$(LIB).so: $(LIB_OBJ_FILES)
      $(Lsh) $(LIB_OBJ_FILES) \
           -o $(LIB).so

test: $(PROG)
      $(PROG)

clean:
      rm -f $(PROG) *.o *~ \
        Makefile.new $(LIB).a $(LIB).so

DFLAGS = -MM
dep:
      sed '/^# Abhaengigkeiten /q' \
        Makefile > Makefile.new
      $(CC) $(DFLAGS) $(SRCS) \
        >> Makefile.new
```

```
<?xml version="1.0"
     encoding="ISO-8859-1"?>

<project name="Netzplan" basedir="."
       default="compile">

<property name="srcdir" value="src"/>
<property name="bindir" value="bin"/>

<target name="compile">
  <mkdir dir="${bindir}"/>
  <javac srcdir="${srcdir}"
        bindir="${bindir}"
        classpath="${bindir}"/>
</target>

<target name="jar" depends="compile">
  <jar destfile="netzplan.jar"
      basedir="${bindir}"/>
</target>

<target name="test" depends="jar">
  <java classpath="${bindir}"
    classname="netztest.TestApplication"/>
</target>

<target name="deploy" depends="jar">
  <copy file="netzplan.jar"
       todir=".."/>
</target>

<target name="clean">
  <delete dir="${bindir}"/>
  <delete file="netzplan.jar"/>
</target>

<target name="undeploy">
  <delete file="../netzplan.jar"/>
</target>

</project>
```

3.5 Zusammenfassung

Je größer das Programm, desto mehr Codezeilen, mehr Dateien und mehr Abhängigkeiten unter den Dateien gibt es. Als Konsequenz der zunehmenden Komplexität des Quellcodes stellen Programmiersprachen Mittel zur Quellcode-Organisation wie Namensräume, Pakete oder Bibliotheken zur Verfügung. Bei der Verteilung von Arbeitspaketen auf die Projektbeteiligten sollten möglichst früh Absprachen zur Paket/Namensraum-Organisation getroffen werden. Eine Widerspiegelung der Paket/Namensraum-Organisation durch die Verzeichnisstruktur ist dringend anzuraten (und in Java verpflichtend).

Je größer das Programm, desto mehr wird sich ein einzelner Entwickler nur mit Ausschnitten des Programms beschäftigen, aber nicht mehr jedes Mal das Gesamtprojekt compilieren und testen. Das Wissen über das korrekte Erstellen einer lauffähigen Applikation verteilt sich so schnell über mehrere Entwickler. Das Compilieren der aktuellen Version sollte aber nicht nur im Zusammenspiel aller Beteiligten möglich sein. Daher sind alle (periodisch wiederkehrenden) Schritte zur Erstellung einer aktuellen Version in einer Projektdatei zu verankern. Die Standardisierung der Projektdateien ermöglicht es jedem Teammitglied, das Projekt komplett neu zu compilieren und die Korrektheit nach erfolgter Modifikation zu testen (zum Beispiel bevor Änderungen in das Versionsmanagement eingepflegt und für andere Entwickler damit freigegeben werden, vgl. Abschn. **??**).

Kapitel 4

Werte- und Referenzsemantik

Wir haben in Kapitel 2 im Zusammenhang mit Funktionsparametern bereits Zeiger und Referenzen auf Variablen kennengelernt, das Thema werden wir in diesem Kapitel vertiefen.

Programmieranfängern stellt sich immer wieder die Frage, weshalb in Programmiersprachen überhaupt überwiegend der indirekte Datenzugriff verwendet wird, obwohl man im Prinzip doch alle Werte auf dem Programm-Stack verwalten könnte. Wir wollen das Problem durch einen Vergleich illustrieren:

> Als Student möchte ich Zugriff auf bestimmte Literatur haben. Ich richte mir eine Bibliothek ein, indem ich einen entsprechenden Raum zur Verfügung stelle, ihn mit Bücherregalen ausstatte und die Regale mit den entsprechenden Büchern fülle. Als Alternative kann ich mir auch eine Kartei oder eine Datei anlegen, in der jedes Buch als Eintrag vertreten ist. Der Eintrag besteht im einfachsten Fall aus dem Buchtitel und aus einem Verweis (Referenz) auf eine Beschaffungsmöglichkeit (z.B. Bibliothek-Signatur oder ISBN-Nummer).

Die erste Lösung entspricht der Wertesemantik, die zweite der Referenzsemantik. Der Vorteil der zweiten Lösung besteht darin, dass sie sparsamer mit Ressourcen umgeht und flexibler ist. Wenn dieser Vergleich vielleicht auch etwas hinkt, so lässt sich das Gesagte aber doch gut auf die Verwendung von Werte- sowie Referenzsemantik in der Programmierung übertragen.

Daher werden wir uns in den Abschnitten 4.1 und 4.2 zunächst mit den Grundlagen der Speicherverwaltung beschäftigen und dabei z.B. erfahren, wie Variablen auf den realen Programmspeicher zur Laufzeit abgebildet werden. In Abschn. 4.3 werden das Thema *Werte oder Referenzen* zusammenfassend behandelt, die Hintergründe erläutert und Entscheidungshilfen gegeben. Im Abschn. 4.4 über *Zeiger auf Funktionen* gehen wir auf eine spezielle Programmiertechnik von C++ ein. In Abschn. 4.5 kommen wir wieder auf das Netzplan-Beispiel zurück und zeigen, wie man mit

dem in diesem Kapitel Gelernten die Netzplanung effizienter gestalten kann. Außerdem geben wir eine Einführung in das Thema *Dynamische Datenstrukturen* (Listen, Bäume usw.).

4.1 Speicherverwaltung im Detail

4.1.1 Die verschiedenen Speicherbereiche eines Programms

Wie in Bild 4.1 dargestellt, enthält der für ein Programm bereitgestellte Arbeitsspeicher je einen statischen Bereich für den lauffähigen Programmcode und für die globalen Daten sowie je einen dynamischen Bereich für die vom System und für die vom Benutzer verwalteten Daten. Als globale Daten sind hier die als extern und die als statisch vereinbarten Variablen zusammengefasst.

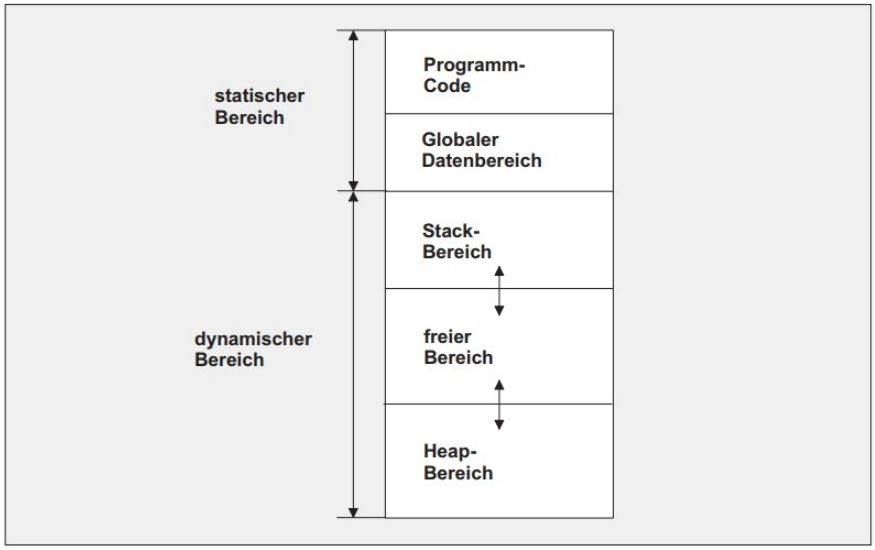

Bild 4.1: Speicherverwaltung

Der vom System verwaltete Teil des Arbeitsspeichers wird häufig *Programm-Stack* genannt. Sobald eine Funktion aufgerufen wird, wird ihr vom System im Programm-Stack ein Datenbereich für ihre lokalen Daten und für ihre Funktionsparameter zugewiesen. Entsprechendes gilt bis auf die Funktionsparameter für den Eintritt in eine Verbundanweisung (durch geschweifte Klammern begrenzter Block).

Als Erstes wird immer die Hauptfunktion `main` ausgeführt. Sobald `main` selbst eine Funktion aufruft (das kann auch eine Bibliotheksfunktion sein), wird ihr ein entsprechender zweiter Datenbereich auf dem *Programm-Stack* zugeordnet. Wenn diese Funktion eine weitere aufruft, erhält sie den dritten Datenbereich auf dem Stack

usw. Sobald eine Funktion oder ein Block beendet wird, gibt das System auch den zugehörigen – auf dem Programm-Stack liegenden – Speicherbereich wieder frei.

Bei rekursiven Funktionen belegt jeder neue rekursive Aufruf wieder seinen eigenen Speicherbereich auf dem Stack, in dem der für den Aufruf aktuelle Satz von lokalen Daten und aktuellen Parametern abgelegt wird.

Der vom Anwender verwaltete Teil des Arbeitsspeichers wird *Programm-Heap* genannt. Die Erzeugung von Objekten auf dem *Programm-Heap* – und damit die Reservierung entsprechender Speicherbereiche – erfolgt jeweils mit dem Operator `new`. Die Freigabe der *Heap-Objekte* erfolgt in C++ in der Regel explizit durch den Programmierer mit den Operatoren `delete` und `delete[]`. In Java werden die Objekte automatisch durch einen *Garbage Collector* freigegeben, wenn sie nicht mehr referenziert werden.

Generell besteht die Möglichkeit, direkt oder indirekt mit den Werten von Variablen (Objekten) zu arbeiten, man spricht dann von *Wertesemantik* bzw. von *Referenzsemantik*. Der Begriff *Referenz* steht hier allgemein für einen *Verweis* und bezeichnet damit die indirekte Verwendung der Objekte. In C++ gibt es dafür zwei Formen, nämlich *Zeiger* und *Referenzen*, d.h. der Begriff *Referenz* wird zum einen als Oberbegriff für zwei verschiedene Formen von Verweisen verwendet und zum anderen für eine spezielle Form von Verweisen in C++. Im Falle der Wertesemantik werden die Objekte auf dem *Programm-Stack* abgelegt. Die Werte der Objekte werden direkt manipuliert; bei der Referenzsemantik befinden sich die Objekte auf dem *Programm-Heap*. Sie werden dann indirekt über Referenzen manipuliert.

4.1.2 Speicherverwaltung auf dem *Programm-Stack*

Das kleine Demo-Programm im Listing 4.1 veranschaulicht zusammen mit Bild 4.2, wie elementare Variablen jeweils auf dem *Programm-Stack* verwaltet werden.

Zur Veranschaulichung wird hier vereinfachend angenommen, dass der *Programm-Stack* bei der Adresse `5000` beginnt und dann zu den größeren Adressen hin wächst:

- Zunächst belegt nur die Variable `i` Speicher auf dem *Programm-Stack*. Beim Aufruf der Funktion `f` wird die Rücksprungadresse `main-Rsp` ebenfalls auf dem *Programm-Stack* abgelegt.[1]
- Die Werteparameter `f1` und `f2` der Funktion `f` werden an die Funktion übergeben und belegen nun ebenfalls – wie lokale Variablen – Platz auf dem *Programm-Stack*.
- Dann wird die Funktion `g` aufgerufen. Dazu werden wieder die aktuelle Rücksprungadresse (`f-Rsp`) und die aktuellen Werte für ihre beiden Parameter `g1` und `g2` abgelegt. Die lokale Variable `tmp` belegt nur Speicher, wenn der `if`-Zweig auch wirklich durchlaufen wird, was bei diesem ersten Aufruf von `f` nicht der Fall ist.

[1] Das ist ebenfalls eine Vereinfachung. In realen C++- und Java-Implementierungen werden zumeist noch weitere Informationen bei einem Funktionsaufruf auf dem *Programm-Stack* abgelegt. Das wird hier aber ignoriert, weil es nichts mit dem Prinzip der unterschiedlichen Heap- und Stackspeicherverwaltung zu tun hat.

C++/Java 4.1: Speicherbelegung auf dem Stack

```cpp
void h(int h1){
  cout << h1*h1 << endl; /* 1 */
}
void g(int g1, int g2){
  int h1p=g1+g2;
  h(h1p);
}
void f(int f1, int f2){
  g(f1, f2*f2);
  if (f1<f2) {
    int tmp = f1 + f2;
    h(tmp);        /* 2 */
  }
}
int main(){
  int i=9;
  f(i, 3);
  f(3, i);
}
```

(WertRef/Funk/FunkMain.cpp)

```java
static public void h(int h1){
  System.out.println(h1*h1); /*1*/
}
static public void g(int g1, int g2){
  int h1p=g1+g2;
  h(h1p);
}
static public void f(int f1, int f2){
  g(f1, f2*f2);
  if (f1<f2) {
    int tmp = f1 + f2;
    h(tmp);         /*2*/
  }
}
static public void main(String[] args){
  int i=9;
  f(i, 3);
  f(3, i);
}
```

(WertRef/Funk/FunkMain.java)

- In der Funktion `g` wird dann die Funktion `h` aufgerufen, d.h. die aktuelle Rücksprungadresse (`g-Rsp`) und der Wert des Parameters `h1` werden abgelegt.

Der linke Teil im Bild 4.2 zeigt die Speicherbelegung auf dem *Programm-Stack* zum Zeitpunkt `/*1*/`, der rechte Teil zeigt die entsprechende Belegung beim zweiten Aufruf von `f` zum Zeitpunkt `/*2*/`. Wir erkennen hierbei auch, dass die Variablen, die zuletzt auf dem *Programm-Stack* angelegt wurden, als Erstes wieder vom *Programm-Stack* entfernt werden (daher der Name *Programm-**Stack***).

4.1.3 Speicherverwaltung auf dem *Programm-Heap*

Ganz anders sieht die Situation in Java bei der Verwendung strukturierter Datentypen wie Klassen und Arrays aus und ebenso auch in C++, wenn für die entsprechenden Objekte der *Programm-Heap* gewählt wird:

- Beim Arbeiten mit *Heap-Objekten* wird auf dem *Programm-Stack* nur eine Referenz (echte Referenz oder ein Zeiger) auf die Variable selbst (entsprechend der Karteikarte im Beispiel zu Beginn dieses Kapitels) abgelegt; siehe Listing 4.2 und Bild 4.3.

In Java werden strukturierte Objekte grundsätzlich – in C++ wahlweise – auf dem *Programm-Heap* abgelegt. Die Variablen `va`, `vb`, `fa` usw. sind nur die Referenzen bzw. Zeiger auf diese *Heap-Objekte*. Beim Funktionsaufruf werden wie üblich die Werte der aktuellen Parameter an die formalen Parameter übergeben, was dazu führt, dass `va` und `fa` auf den gleichen Vorgang verweisen.

Zum Zeitpunkt `/*2*/` ist die Variable `fc` nicht mehr gültig, d.h. sie befindet sich auch nicht mehr auf dem Stack. Der Speicherbereich (ab Adresse `8000`) existiert noch, allerdings gibt es keinen gültigen Verweis mehr auf ihn. In Java ist das kein

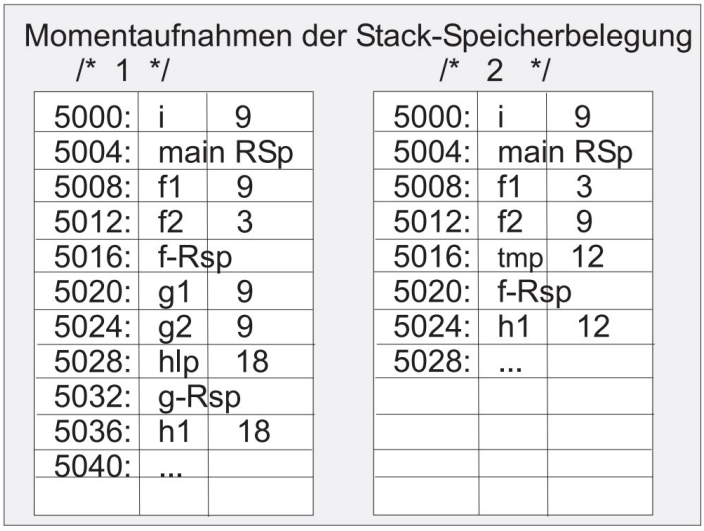

Bild 4.2: Speicherbelegung auf dem *Programm-Stack*

Problem, das Laufzeitsystem – genauer gesagt: der *Garbage Collector* – erkennt das und gibt diesen Speicherbereich (irgendwann) an das Laufzeitsystem zurück. In C++ dagegen wird in der Regel so ein *Freispeichersammler* nicht verwendet.[2] Damit entsteht dann ein Speicherleck, d.h. ein Speicherbereich, auf den es keinen gültigen Verweis mehr gibt. Deshalb muss dieser Speicherbereich in C++ explizit freigegeben werden, solange es noch einen Verweis hierauf gibt. Hierzu stellt C++ die Operatoren `delete` und `delete[]` zur Verfügung.

[2]Entgegen weit verbreiteten Vorstellungen ist es im C++-Standard **nicht** festgelegt, ob C++-Implementierungen einen *Garbage Collector* verwenden oder nicht!

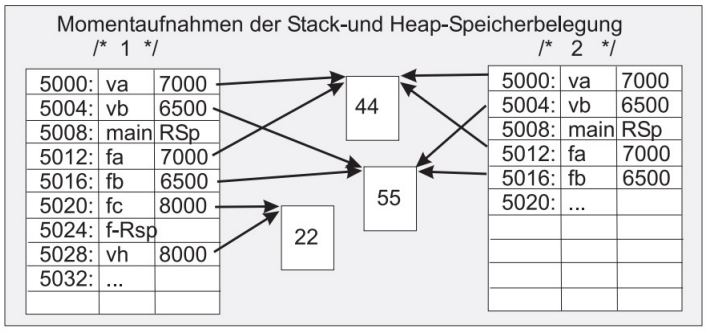

Bild 4.3: Speicherbelegung auf dem *Programm-Stack* und *Programm-Heap*

C++/Java 4.2: Speicherbelegung auf dem *Programm-Stack* und *Programm-Heap*

```
void h(Vorgang* vh){
  cout << vh->dauer << endl;/*1*/
}
void f(Vorgang* fa,
     Vorgang* fb){
  fb->dauer = 55;
  if (fa->dauer < fb->dauer) {
    Vorgang* fc =
      new Vorgang;
    fc->dauer=22;
    h(fc);
    delete fc; fc=0; /* siehe Text */
  }
  h(fa);              /* 2 */
}
int main(){
  Vorgang* va;
  Vorgang* vb;
  va = new Vorgang();
  va->dauer = 44;
  vb = new Vorgang();
  f(va, vb);
  f(vb, va);
  delete va; va=0; /* siehe Text */
  delete vb; vb=0; /* siehe Text */
}
```

(WertRef/FunkRefs/FunkMain.cpp)

```
public class FunkMain {
static public void h(Vorgang vh){
  System.out.println(vh.dauer);/*1*/
}
static public void f(Vorgang fa,
     Vorgang fb){
  fb.dauer = 55;
  if (fa.dauer < fb.dauer) {
    Vorgang fc =
      new Vorgang();
    fc.dauer=22;
    h(fc);
  }
  h(fa);              /* 2 */
}

static public void main(String[] args){
  Vorgang va;
  Vorgang vb;
  va = new Vorgang();
  va.dauer = 44;
  vb = new Vorgang();
  f(va, vb);
  f(vb, va);
}
}
```

(WertRef/FunkRefs/FunkMain.java)

Die C++-Operatoren `delete` und `delete[]`: Der Aufruf `delete ptr;` gibt den ab der Speicherstelle (Adresse) `ptr` zuvor mit `new` reservierten Speicherbereich wieder frei.

Achtung: `delete` darf man nur auf einen Zeiger anwenden, dem man vorher mit `new` Speicherplatz zugewiesen hat und der noch nicht wieder freigegeben wurde.

Tipp

Wird `delete` allerdings auf einen Null-Zeiger[3] angewendet, so ist der Aufruf des Operators wirkungslos. Es ist auch noch aus einem zweiten Grund zweckmäßig, nach einem `delete`-Aufruf dem Zeiger, der auf den gerade freigegebenen Speicher zeigt, wie im obigen Beispiel den Wert `0` zuzuweisen. Anschließend kann durch diesen Zeiger nicht mehr versehentlich auf eine Variable zugegriffen werden, die es gar nicht mehr gibt.

Das nächste Beispiel zeigt die Verwendung der Operatoren `new` und `delete` beim Arbeiten mit Arrays:

```
float* vector = new float[20];
. . .
delete[] vector;
//   ^^                  // wichtig !
```

[3] Es gibt einen besonderen Zeigerwert, der als Konstante mit dem Namen `NULL` in C++ bzw. `null` in Java definiert ist. Die interne Darstellung ist implementierungsabhängig. Der `NULL`-Zeiger kann jeder C++-Zeigervariablen zugewiesen werden bzw. jede Java-Referenz kann mit der `null`-Referenz belegt werden. Außerdem ist eine Abfrage/Test auf den Wert `NULL` bzw. `null` möglich.

Bei der Freigabe eines Arrays muss der **Subskriptoperator** (`[]`) verwendet werden! Das Weglassen des Subskriptoperators führt zu keinem Syntaxfehler, sondern zu undefiniertem Verhalten zur Laufzeit. Zu beachten ist ferner, dass die Operatoren `new` und `new[]` den gelieferten Speicher **nicht** initialisieren!

4.1.4 Strukturierte Objekte auf dem C++-*Programm-Stack*

C++

Bisher war es noch relativ einfach: Objekte elementarer Datentypen wurden als Werte auf dem *Programm-Stack* abgelegt, Objekte strukturierter Datentypen wie Klassen und Arrays wurden mit `new` auf dem *Programm-Heap* erzeugt.

In diesem Abschnitt werden wir nun aber sehen, dass in C++ auch strukturierte Objekte auf dem *Programm-Stack* abgelegt werden können.

In unserem Beispiel in Listing 4.3 und Bild 4.4 wird das Objekt `v` direkt auf dem *Programm-Stack* angelegt und das Objekt, auf das `pv` verweist, auf dem *Programm-Heap*:

- Der Funktion `f` wird beim Aufruf in `main` nun der erste Parameter als Zeiger, einer speziellen Form einer Referenz in C++, und der zweite als Wert übergeben: `f(pv, v);`
- Änderungen in der Funktion am Werteparameter `fv` wirken sich damit auch nur lokal in der Funktion `f` aus,
- während über den formalen Parameter `fp`, einem Zeigertyp, auch der aktuelle Parameter in der aufrufenden Funktion verändert wird, d.h. die Dauer der Variablen, auf die `pv` verweist, wird von `44` auf `45` erhöht.

C++ 4.3: Strukturierte Variablen auf dem *Programm-Stack* und *Programm-Heap*

```
#include <iostream>
using namespace std;
#include "Vorgang.h"

void printVo(const Vorgang& v) {
  cout << v.dauer;
  cout << " ";
}

void f(Vorgang* fp,
     Vorgang fv){
  ++(fp->dauer);
  ++(fv.dauer);
}
```

```
int main(){
  Vorgang* pv = new Vorgang;
  pv->dauer = 44; // auf dem Heap
  Vorgang v; // auf dem Stack
  v.dauer = 55;
  printVo(*pv); printVo(v);
  f(pv, v);
  printVo(*pv); printVo(v); /*1*/
  // Uebergabe der Adresse von v
  // und des Wertes von pv
  f(&v, *pv);
  printVo(*pv); printVo(v); /*2*/
  delete pv; pv=0;
}
```

(WertRef/StaKlasse/MainVorg.cpp)

Die Ausgabe des Programms ist:

```
44 55 45 55 45 56
```

Etwas überraschend mag auf den ersten Blick der zweite Aufruf der Funktion `f` in `main` – d.h. `f(&v, *pv);` – wirken:

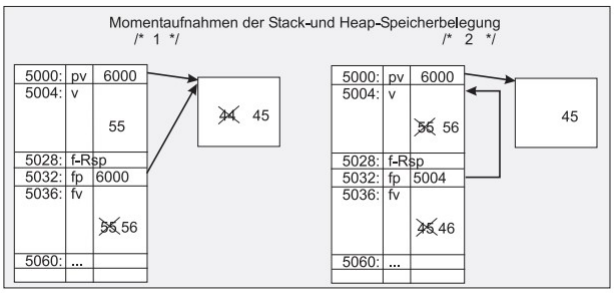

Bild 4.4: Strukturierte Variablen auf dem *Programm-Stack*

- Der erste formale Parameter `fp` erwartet einen Zeiger, d.h. die Adresse einer Variablen. C++ bietet nun mit dem Adressoperator (`&`) die Möglichkeit, die Speicheranfangsadressen von beliebigen Variablen (zur Laufzeit) zu ermitteln. Damit ist es möglich, die Adresse des Vorgangs v, der auf dem *Programm-Stack* liegt, an den formalen Parameter `fp` zu übergeben.
- Umgekehrt erwartet der zweite formale Parameter `fv` einen Wert. Mit dem Dereferenzierungs-Operator (`*`) ist es in C++ möglich, einen Zeiger in einen Wert umzuwandeln. Da der Funktion in `fv` eine Kopie eines Vorgangs (auf dem Programm-Stack) übergeben wird, führt die Anweisung `*pv` dazu, dass der Wert der Variablen, auf die `pv` verweist, vom Heap auf den Programm-Stack kopiert wird; siehe Bild 4.4 rechts.
- Insgesamt führt der zweite Aufruf von `f` somit dazu, dass der Vorgang auf dem *Programm-Heap* nicht verändert wird, da `f` auf der Kopie `fv` arbeitet.
- Andererseits wird in `f` der Vorgang `v` auf dem *Programm-Stack* verändert, da an `f` dessen Adresse übergeben wurde.

Als ob die Sache mit Werten und Adressen auf dem *Programm-Stack* oder auf dem *Programm-Heap* nicht schon kompliziert genug wäre, bietet C++ mit seinen Referenzen noch eine weitere Möglichkeit. Eine Referenz wurde im Beispiel in der Funktion `printVo` zur Parameterübergabe verwendet. Semantisch sind C++-Referenzen und Zeiger verwandt, beide sind Verweise und realisieren eine Form der Indirektheit. Referenzen sind in C++ lediglich andere Namen für Variablen (z.B. als aktuelle Parameter). Änderungen an einer Referenz wirken sich immer auch auf den aktuellen Parameter aus. Syntaktisch werden Referenzen aber wie Werteparameter behandelt. Will man z.B. auf die Komponenten einer Struktur oder Klasse über eine Referenz zugreifen, wird hierzu der Punkt verwendet; bei Zeigern wird hierzu der Pfeil verwendet:

```
void kompRef(Vorgang& r){
  r.dauer = 44;
  cout << r.fruehanf;
}
```

```
void kompZeiger(Vorgang* z){
  z->dauer = 44;
  cout << z->fruehanf;
}
```

Die Unterscheidung zwischen C++-Referenzen und -Zeigern fällt erfahrungsgemäß sehr schwer. Hier mag folgende Gedächtnisstütze Abhilfe schaffen:

> Referenzen sind in C++ ständig dereferenzierte Zeiger. Die Dereferenzierung wird bereits zur Compile-Zeit durchgeführt. Während Zeigern auch andere Werte (Adressen) zugewiesen, d.h. *umgebogen* werden können, stellt eine Referenz in C++ einen festen Bezug auf eine Variable dar und wird wie ein zweiter Name (Synonym, Alias) benutzt.
> Referenzen in Java dagegen können auch *umgebogen* werden und entsprechen in dem Punkt den Zeigern in C++. Java-Referenzen entsprechen somit weder vollständig den C++-Zeigern noch den C++-Referenzen. Sie stehen zwischen beiden Konzepten.

4.1.5 Java-Standardtypen als Referenzparameter

Java

Wir haben gesehen, dass Instanzen (Variablen) echter Klassen in Java nur als Referenzen an Funktionen[4] übergeben werden können und Variablen vom Typ `int` oder `double` nur als Werte. Wenn man Arrays an Java-Funktionen übergeben will, ohne dass sie verändert werden sollen, kann man mit der Funktion `System.arraycopy` zunächst eine Kopie von ihnen anlegen und diese dann anstelle der Originale an die Funktion übergeben, die die Inhalte der Arrays verändern könnte; siehe Abschn. 4.2.1.

Wie ist das nun im umgekehrten Fall, wenn man einen Basistyp an eine Funktion übergeben will und diese Funktion den Parameter ändern soll? Wir betrachten z.B. den Fall, dass wir zwei ganze Zahlen `a` und `b` an eine Funktion übergeben und die Funktion `a+b`, `a-b` und `a*b` zurückliefern soll. Das Listing 4.4 stellt links eine falsche und rechts eine richtige Lösung für eine entsprechende Funktion `calc` bzw. `calcWrap` dar.

Die Funktion `test` im linken Teil würde `1 2 3` ausgeben, während die Funktion `testWrap` im rechten Teil die gewünschte Ausgabe `15 5 50` liefert. Die benötigte Hilfsstruktur `Wrap` besteht aus einem einzigen Attribut `val`.

```
public class Wrap{ public int val; };
```

Die Speicherbelegung im Heap- und Stack-Speicher zeigt Bild 4.5. Für die anderen Standardtypen können wir entsprechende Wrapper-Klassen implementieren.[5] Als Alternative zu `Wrap` haben wir in Kap. 2.1 ein `int`-Array benutzt.

Java 4.4: Standardtypen als Referenzparameter

```
public static void
  calc(int a, int b,
       int s, int d, int p) {
  s = a + b;
  d = a - b;
  p = a * b;
 }

public static void test() {
  int a = 10; int b = 5;
  int sum=1, diff=2, prod=3;
  calc(a,b,sum,diff,prod);

  System.out.println(
    sum + " " + diff + " " + prod);
 }
```

```
public static void calcWrap(int a,
    int b, Wrap s, Wrap d, Wrap p) {
  s.val = a + b;
  d.val = a - b;
  p.val = a * b;
 }
public static void testWrap() {
 int a = 10; int b = 5;
 Wrap sum = new Wrap(); sum.val=1;
 Wrap diff= new Wrap(); diff.val=2;
 Wrap prod= new Wrap(); prod.val=3;

 calcWrap(a, b, sum, diff, prod);

 System.out.println(
   sum.val+" " + diff.val+" " +prod.val);
 }
```

(Funk/Klassen/Wrapper.java)

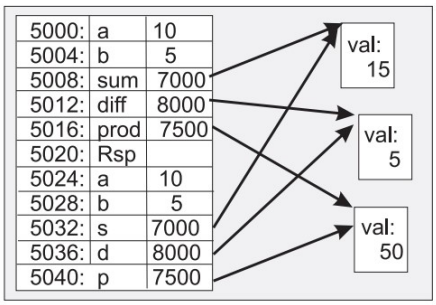

Bild 4.5: Speicherbelegung bei Verwendung einer Wrapper-Klasse für Standardtypen

4.1.6 Java-Hüllklassen für Standardtypen

Java

Hüllklassen, die der im Listing 4.4 benutzten `WrapInt`-Klasse sehr ähnlich sind, stellt Java bereits von Haus aus zur Verfügung:

- `Integer` für den Typ `int`,
- `Double` für den Typ `double`,
- `Character` für den Typ `char` usw.

Die Umwandlung eines Standardtyps in die entsprechende Hüllklasse wird auch als **Boxing** bezeichnet und die entsprechende Rückwandlung mit den Funktionen `intValue`, `doubleValue` usw. als **Unboxing**.

[4] Wir weisen nochmals daraufhin, dass Java streng genommen gar keine Funktionen kennt, sondern nur Methoden. Den Unterschied können wir aber erst in Kap. 6 verstehen. Daher sprechen wir in diesem Kapitel immer von Funktionen.

[5] Der Versuch, diese Definitionen durch ein Template der Art `class WrapT<T> {public T val;}` zusammenzufassen, scheitert daran, dass Java für den Typparameter `T` keine elementaren Datentypen zulässt.

Leider sind diese Standard-Hüllklassen für die Verwendung als Referenzparameter ungeeignet. Ist ein `Integer`-Objekt erst einmal erzeugt, kann der Wert nachträglich nicht mehr verändert werden. Wollen wir z.B. den Inhalt eines `Integer`-Objekts iObj verdoppeln, so müssen wir ein neues Objekt erzeugen:

```
iObj = new Integer(10);
```

```
iObj = new Integer(iObj.intValue() * 2);
```

Deshalb können diese Java-Hüllklassen nicht dazu benutzt werden, um Standardtypen als Referenzen an Funktionen zu übergeben:

```
calcJavaWrap(int a, int b,
    Integer s, Integer d, Integer p) {
   s = a + b;
   d = a - b;
   p = a * b;
  System.out.println(s+" "+d+ " "+p);
}
```

```
public static void testJavaWrap() {
  int a = 10; int b = 5;
  Integer sum = new Integer(1);
  Integer diff = new Integer(2);
  Integer prod = new Integer(3);

  calcJavaWrap(a,b,sum,diff,prod);

  System.out.println(
     sum + " " + diff + " " + prod);
  // Umwandlung in Basistyp
  int n = sum.intValue();
}
```

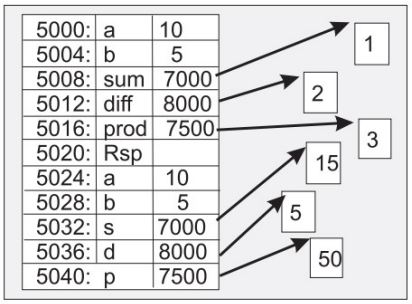

Bild 4.6: Speicherbelegung bei Verwendung der eingebauten Java-Hüllklassen

Die Ausgabe des Programms ist zwar in der aufgerufenen Funktion `calcJavaWrap 15 5 50`, doch dies hat keine Auswirkungen auf die Variablen in der aufrufenden Funktion, die nach wie vor `1 2 3` ausgibt. Bild 4.6 verdeutlicht den Grund dafür: Die Referenzen `sum` und `s` etc. verweisen zunächst auf das gleiche `Integer`-Objekt. Die Anweisung `s = a + b;` führt nun aber ein automatisches *Boxing* durch, d.h. es ist eine abgekürzte Schreibweise für `s = new Integer(a + b);` Die Veränderung der lokalen Referenz `s` (*Umbiegen des Zeigers* würde man in C++ sagen) hat aber keinen Einfluss auf die Stelle, auf die die Referenz `sum` verweist (`sum` wird nicht *umgebogen*).

4.1.7 Zusammenfassung

Verschiedene Arten von Verweisen (Referenzen)

Im Gesamtrahmen der beiden Sprachen C++ und Java gibt es drei verschiedene Arten von Verweisen (Referenzen):

- C++-Zeiger bezeichnen einen Speicherplatz, in dem ein Objekt (Wert, Variable) abgelegt wird. Das Objekt selbst kann als *Inhalt des Speicherplatzes* angesprochen werden, man nennt das *Dereferenzierung*. Ein C++-Zeiger kann in der Regel *umgebogen*, d.h. auf ein anderes Objekt (des gleichen Typs) gerichtet werden.
- C++-Referenzen sind Zweitnamen (Synonyme, Alias) für Objekte, sie verweisen fest auf das Objekt und können nicht *umgebogen*, d.h. nicht auf ein anderes Objekt gerichtet werden.
- Java-Referenzen sind eine Mischung aus den beiden vorangegangenen Fällen: Einerseits können sie wie C++-Zeiger *umgebogen*, d.h. zur Laufzeit mit anderen Objekten verbunden werden. Andererseits bezeichnen sie das Objekt (und nicht den Speicherplatz) und gleichen damit syntaktisch eher den C++-Referenzen.

Der Begriff *Referenz* wird sowohl als Oberbegriff für alle drei genannten Arten von Verweisen verwendet als auch speziell für die C++- und Java-Referenz benutzt.

Werte- und Referenzsemantik

Wertesemantik bedeutet, dass Objekte (Variablen, Werte) *direkt* über ihre Namen benutzt werden. *Referenzsemantik* bedeutet, dass Objekte (Variablen, Werte) *indirekt* über Verweise benutzt werden. Weitere Hintergrundinformationen und Entscheidungshilfen zum Thema *Werte oder Referenzen* liefert der Abschn. 4.3.

Speicherverwaltung

- In C++ können globale Objekte und *Stack-Objekte* sowohl direkt als auch indirekt über Referenzen und Zeiger benutzt werden.
- In Java werden *Stack-Objekte* (d.h. in Java elementare Datentypen) grundsätzlich direkt benutzt.
- In C++ werden *Heap-Objekte* (das können in C++ sowohl elementare als auch strukturierte Objekte sein) indirekt per Zeiger oder auch per Referenz benutzt.
- In Java werden *Heap-Objekte* (d.h. in Java strukturierte Objekte) grundsätzlich indirekt per Referenz benutzt.
- Java verwendet grundsätzlich einen *Garbage Collector (Freispeicher-Sammler)*; C++ verwendet in der Regel (aus grundsätzlichen Erwägungen) keinen *Garbage Collector*, doch gibt es durchaus die Möglichkeit, auch in C++-Systemen einen *Garbage Collector* zu installieren.

4.1.8 Übungen

Übung 4.1: Implementieren Sie eine Funktion `minMaxDurch`, der beliebig viele ganzzahlige Argumente in einem Array übergeben werden und die das maximale und das minimale Arrayelement sowie den Durchschnitt aller Arrayelemente über Zeiger zurückliefert.

Das folgende Listing zeigt die Deklaration der Funktion und eine kleine Anwendung. Vergessen Sie bei Ihrer Implementierung nicht, entsprechende (automatisch ablaufende) Tests für die neue Funktion zu implementieren.

```
void minMaxDurch( int anz, const int arr[],
                  int* pmin, int* pmax, double* pdurch);
int min, max; double durch;
int arr[] = {1, 2, 3, 4, 5, 6};

minMaxDurch(6, arr, &min, &max, &durch);
```

Übung 4.2: Zu welcher Ausgabe führt der Aufruf der Funktionen `mitZeigern` bzw. `mitEchtenRefs`? **C++**

```
void show(Vorgang* z1, Vorgang* z2) {
   cout<<z1->dauer<<" "<<z2->dauer<<endl;
}
void mitZeigern() {
   // Zeiger koennen auch auf andere
   // Objekte gerichtet werden
   Vorgang* z1 = new Vorgang;
   Vorgang* z2 = new Vorgang;
   z1->dauer = 11; z2->dauer = 22;
   show(z1, z2); /* 1 */

   z1 = z2; show(z1, z2); /* 2 */

   z1->dauer = 44; show(z1, z2);

   z2->dauer = 55; show(z1, z2); /* 3 */
}
```

```
void show(Vorgang& r1, Vorgang& r2) {
   cout<<r1.dauer<<" "<<r2.dauer<<endl;
}
void mitEchtenRefs() {
   // Referenzen sind unveraenderbar mit
   // einem Objekt verbunden
   Vorgang& r1 = *(new Vorgang);
   Vorgang& r2 = *(new Vorgang);
   r1.dauer = 11; r2.dauer = 22;
   show(r1, r2); /* 4 */

   r1 = r2; show(r1, r2); /* 5 */

   r1.dauer = 44; show(r1, r2);

   r2.dauer = 55; show(r1, r2); /* 6 */
}
```

(./WerteRef/RefZeigerAufg.cpp)

Veranschaulichen Sie jeweils durch eine eigene Zeichnung die Speicherbelegung zu den Zeitpunkten `/*1*/` bis `/*6*/` auf dem Programm-Stack und -Heap (siehe Bild 4.3).

Übung 4.3:

a) Implementieren Sie eine C++-Funktion `mit30_40Belegen`, sodass deren Aufruf im folgenden Programmausschnitt zu der Ausgabe `vz1: 30, vz2: 40` führt.

```
int main() {
  Vorgang* vz1 = NULL; Vorgang* vz2 = NULL;
  /* ... vorher ... */
  mit30_40Belegen(vz1, vz2);
  /* ... nachher ... */
  cout << "vz1: " << vz1->dauer << ", vz2: " << vz2->dauer << "\n";
```

```
  delete vz1; vz1 = NULL; delete vz2; vz2 = NULL;
  }
```

b) Wie könnte eine entsprechende Lösung in Java für die Funktion aussehen, wenn man in den mit `vorher` und `nachher` auskommentierten Codeabschnitten noch Code vor und nach dem Funktionsaufruf einfügen würde? Der `new`-Aufruf soll allerdings ausschließlich in der aufgerufenen Funktion `mit30_40Belegen` erfolgen.

C++ **Übung 4.4:** Zu welcher möglichen Ausgabe *könnte (!)* der folgende Programmausschnitt führen?

```
int main() {
  int i = 7; int j = 16;
  int* pint = & i;
  *pint = 22;

  cout<< "pint selbst ist " <<pint
    << " *pint ist " <<*pint
    << "  i ist " << i
    << "  j ist " << j <<"\n";
  pint = &j;
  cout << "pint selbst ist " <<pint
    << " *pint ist " <<*pint
    << "  i ist " << i
    << "  j ist " << j <<"\n";
```

```
  int** ppint = & pint;

  *pint = 18;
  cout << "**ppint ist " << **ppint
    << "* pint ist " <<*pint
    << "  j ist " << j<<"\n";

  j = 66;
  cout << "**ppint ist " << **ppint
    << " *pint ist " << *pint
    << "  j ist " << j<<"\n";
  return 0;
  }
```

(./WerteRef/ZeigerAufZeiger.cpp)

4.2 Zeiger und Arrays

4.2.1 Kopieren von referenzierten Objekten

Wenn man ein Objekt über einen Verweis benutzt – in C++ über einen Zeiger, in Java (bei strukturierten Objekten fest vorgegeben) über eine Referenz –, dann gibt es zwei Möglichkeiten des Kopierens. Bild 4.7 veranschaulicht das bildlich:

- *Flach kopieren* bedeutet, der Zeiger bzw. die Referenz wird kopiert, d.h. wird auf das Zielobjekt gerichtet (*umgebogen*). Damit sind dann zwei Verweise auf `Objekt1` gerichtet, und `Objekt2` ist ein verwaistes Objekt im Speicher.
- *Tief kopieren* bedeutet hingegen, dass nicht der Zeiger, sondern das Objekt selbst als Ganzes kopiert wird. Der bisherige Inhalt von `Objekt2` wird durch `Objekt1` überschrieben.

In vielen Fällen ist es sinnvoll, eine echte (tiefe) Kopie eines Objektes, z.B. eines Arrays, anzulegen, denn Arrays werden immer als Referenzparameter an Funktionen übergeben, sodass eine Änderung eines Arrayelements in der Funktion immer auch zu einer Änderung des entsprechenden aktuellen Parameters in der aufrufenden Funktion führt. Die beiden Programmtexte in Listing 4.5 zeigen eine entsprechende Lösung in C++ bzw. in Java, jeweils eingebettet in eine einfache Anwendung:

Der Aufruf der Funktion `arrayKopie1` führt jeweils zu der Ausgabe:

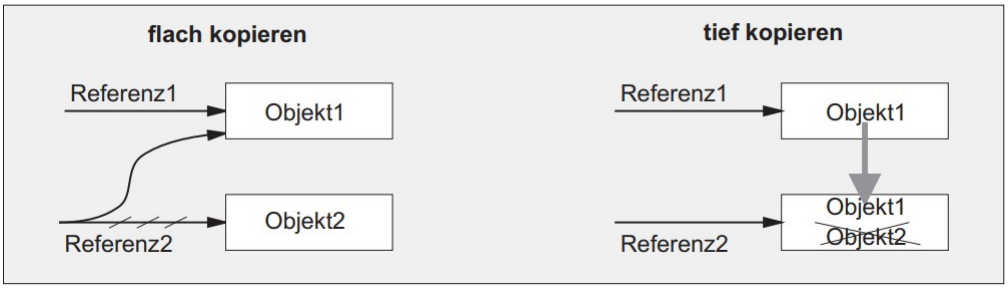

Bild 4.7: Kopieren von referenzierten Objekten

```
0 1 2 3 4 5 6 7 8 9
0 3 6 9 12 15 18 21 24 27
```

Es wurde eine echte (tiefe) Kopie des Arrays `array1` durch Kopieren aller einzelnen Arrayelemente angelegt. Hierbei darf nicht die `new`-Anweisung für das Array `array2` fehlen, da sonst für `array2` kein neuer Speicherplatz zur Verfügung steht. Durch Übergabe von `array2` an die Funktion `vervielfache` ändert sich jetzt nur noch `array2`, auf die Elemente von `array1` hat das nun keinen Einfluss mehr.

Würde man hingegen `int* array2 = new int[GR];` bzw. `int array2[] = new int[GR];` jeweils ersetzen durch `array2 = array1;` so würde der Zeiger bzw. die Referenz `array2` auf das ursprüngliche Objekt gerichtet werden (flache Kopie). Die anschließenden Zuweisungen `array2[j] = array1[j];` würden nichts bewirken, und die Ausgabe wäre dann:

```
0 3 6 9 12 15 18 21 24 27
0 3 6 9 12 15 18 21 24 27
```

Arrays unterschiedlicher Länge

An die Java-Funktionen `vervielfache` und `ausgabe` kann die Übergabe der Arraygröße unterbleiben, denn in Java sind die Arrays selbst als strukturierter Datentyp (Klasse) realisiert und haben damit auch Attribute. Jedes Java-Array verfügt über das Attribut `length`, über das die Länge des Arrays zur Laufzeit ermittelt werden kann. Wir nutzen dieses Wissen, um eine Funktion `kopiereArray` zu implementieren, die eine echte Kopie eines Arrays erzeugt; siehe Listing 4.6.

Statt der `for`-Schleifen zum Kopieren des Arrays `array1` nach `array2` sieht das Kopieren dann wie folgt aus:

```
kopiereArray(array1, array2, 6);
```

```
kopiereArray(array1, array2);
```

In Java können wir statt der selbst implementierten Funktion `kopiereArray` auch die vordefinierte Funktion `System.arraycopy` verwenden. Sie hat 5 Parameter:

- Ausgangsarray (Quelle),
- Startindex im Quellarray,

C++/Java 4.5: Kopieren eines Arrays

```cpp
/** Die anz Elemente des Arrays f
    werden auf dem Bildschirm
    ausgegeben.            */
void ausgabe(const int f[], int anz){
  for (int i=0; i<anz; ++i) {
    cout << f[i] << " ";
  }
  cout << "\n";
}
/** Alle anz Elemente des Arrays f werden
    mit dem Faktor v multipliziert */
void vervielfache(
        int f[], int anz, int v){
  for (int i=0; i<anz; ++i) {
    f[i] *= v;
  }
}
void arrayKopie1(){
  const int GR=10;
  int* array1 = new int[GR];
  for (int i=0; i<GR; ++i) {
    array1[i] = i;
  }
  int* array2 = new int[GR];
  for (int j=0; j<GR; ++j) {
    array2[j] = array1[j];
  }
  vervielfache(array2, GR, 3);
  ausgabe(array1,GR);
  ausgabe(array2,GR);
}
```

(Funk/Kopie/Kopie.cpp)

```java
public class Kopie {
/** Die anz Elemente des Arrays f werden
    werden auf dem Bildschirm ausgegeben. */
static void ausgabe(int f[], int anz){
  for (int i=0; i < anz; ++i) {
    System.out.print(f[i] +" ");
  }
  System.out.println();
}
/** Alle anz Elemente des Arrays f werden
    mit dem Faktor v multipliziert */
static void vervielfache(
        int f[], int anz, int v){
  for (int i=0; i<anz; ++i) {
    f[i] *= v;
  }
}
static void arrayKopie1(){
  final int GR=10;
  int array1[] = new int[GR];
  for (int i=0; i<GR; ++i) {
    array1[i] = i;
  }
  int array2[] = new int[GR];
  for (int j=0; j<GR; ++j) {
    array2[j] = array1[j];
  }
  vervielfache(array2, GR, 3);
  ausgabe(array1,GR);
  ausgabe(array2,GR);
}
```

(Funk/Kopie/Kopie.java)

C++/Java 4.6: Funktion für tiefe Kopie eines Arrays

```cpp
/** Die n Elemente aus dem Array q werden
    in das Zielarray z kopiert. Genuegend
    Speicherplatz muss in z bereits
    vorhanden sein */
void kopiereArray(int q[], int z[], int n){
  for (int i=0; i < n; ++i) {
    z[i] = q[i];
  }
}
```

(Funk/Kopie/Kopie.cpp)

```java
/** Alle Elemente aus dem Array q werden
    in das Zielarray z kopiert. Genuegend
    Speicherplatz muss in z bereits
    vorhanden sein */
static void kopiereArray(int q[], int z[]){
  for (int i=0; i < q.length; ++i) {
    z[i] = q[i];
  }
}
```

(Funk/Kopie/Kopie.java)

- Zielarray,
- Startindex im Zielarray,
- Anzahl der zu kopierenden Arrayelemente.

Die folgenden Java-Zeilen zeigen links nochmals die `for`-Schleife aus Listing 4.5, und rechts sehen wir die Verwendung der Funktion `System.arraycopy`.

```
int array2[] = new int[GR];
for (int j=0; j<GR; ++j) {
  array2[j] = array1[j];
}
```

```
/* NEU */
int array2[] = new int[GR];
System.arraycopy(
   array1, 0, array2, 0, GR);
```

In unserem Beispiel sind die Arrayelemente vom Typ `int`, d.h. elementare Datentypen. Wenn aber die Arrayelemente auch wieder Verweise (C++-Zeiger bzw. Java-Referenzen) auf Objekte sind, dann muss auch wieder zwischen *flacher Kopie* und *tiefer Kopie* der Elemente unterschieden werden; Bild 4.8 stellt den Unterschied bildlich dar, das Thema wird in Kap. 6 vertieft.

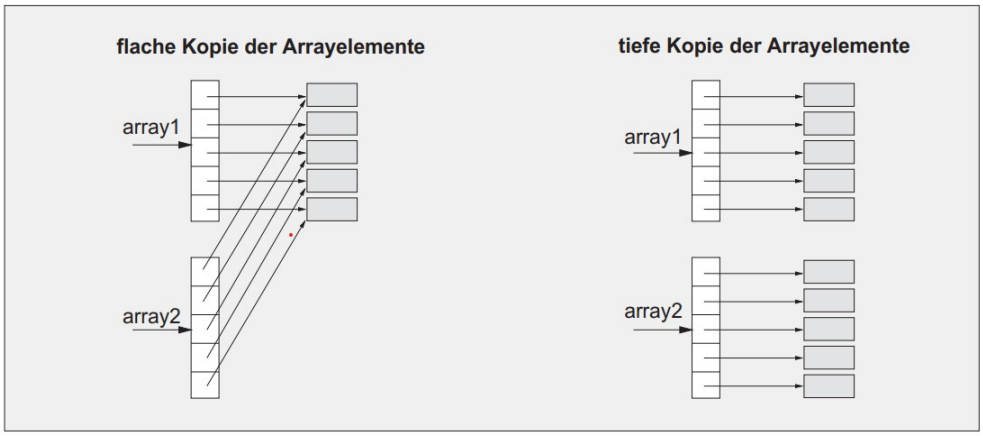

Bild 4.8: Kopieren von referenzierten Arrayelementen

Veränderung der Arraygrößen zur Laufzeit

Nun können wir zwar echte Arraykopien erzeugen, allerdings müssen wir bei der Erzeugung eines Arrays mit dem new-Operator bereits wissen, wie viele Elemente das Array einmal maximal enthalten wird.

Dies ist nicht immer von Anfang an möglich, hier gibt es jedoch eine sehr einfache Lösung, wie das folgende Beispiel zeigt:

```
void varArrayDumm() {
  int* array;
  array = new int[6];
  for (int i=0; i<6; ++i) {
     array[i] = i;
  }
  ausgabe(array, 6);
  /* Wir brauchen mehr Platz */
  array = new int[1000];
  /* alter Inhalt verloren */
}
```

```
static void varArrayDumm(){
  int array[];
  array = new int[6];
  for (int i=0; i<6; ++i) {
     array[i] = i;
  }
  Kopie.ausgabe(array, 6);
  /* Wir brauchen mehr Platz */
  array = new int[1000];
  /* alter Inhalt verloren */
  }
```

Wir haben hierdurch jetzt zwar ein größeres Array erzeugt, dafür ist der ursprüngliche Inhalt des Arrays allerdings verloren gegangen. Deshalb muss der alte Inhalt des Arrays in das vergrößerte Array kopiert werden, bevor mit dem vergrößerten Inhalt weitergearbeitet werden kann, siehe Listing 4.7.

C++/Java 4.7: Kopieren des bisherigen Inhalts bei der Arrayvergrößerung

```
  /* Wir brauchen mehr Platz */
  int* hlp = new int[1000];
  /* bisherigen Inhalt kopieren */
  kopiereArray(array, hlp, 6);
  // Hilfsarray wird grosses Array
  array = hlp;
  // Test der Vergroesserung
  array[6] = 7; array[7] = 8;
  ausgabe(array, 8);
}
```

(Funk/Kopie/VarFeld.cpp)

```
  /* Wir brauchen mehr Platz */
  int hlp[] = new int[1000];
  /* bisherigen Inhalt kopieren */
  System.arraycopy(array,0,hlp,0,6);
  // Hilfsarray wird grosses Array
  array = hlp;
  // Test der Vergroesserung
  array[6] = 7; array[7] = 8;
  Kopie.ausgabe(array, 8);
}
```

(Funk/Kopie/VarFeld.java)

Hierbei müssen wir darauf achten, die Referenz `array` nicht sofort zu überschreiben, da wir sonst keinen Zugriff mehr darauf haben. Deshalb wird das vergrößerte Array zunächst einer *Hilfsreferenz* zugewiesen. Die tiefe Kopie wird mit den zuvor vorgestellten Funktionen `kopiereArray` (siehe Listing 4.6) bzw. `System.arraycopy` durchgeführt.
Die Ausgabe der Funktion ist nach der gezeigten Anpassung dann wunschgemäß:

```
0 1 2 3 4 5
0 1 2 3 4 5 7 8
```

4.2.2 Die Dualität zwischen C++-Zeigern und -Arrays

C++ Das kleine Demoprogramm im Listing 4.8 zeigt am Beispiel des Arrays `a`, dass man auf zwei verschiedene Arten mit solchen Arrays arbeiten kann:

- zum einen durch Verwenden des Subskriptoperators („`[ ]`“) wie in anderen klassischen Sprachen (z.B. Pascal, Modula, Oberon und Ada) und
- zum anderen über das Verwenden von Zeigern, Adressen und *Adress-Arithmetik*.

C++ 4.8: Arbeiten mit Zeigern bzw. Arrays

```
int main() {
  int a[10]; // Array fuer 10 int-Werte

  // Zugriff per Subskriptoperator:

  for (int i = 0; i < 10; i++) a[i] = i * i;
  for (int i = 0; i < 10; i++) cout << a[i];
  cout << endl;
```

(WertRef/Zeiger/AdrZeigVektoren.cpp)

```
  // Zugriff per Zeiger:

  int* pa = &a[0]; //Anfangsadr. von Array a
  for (int i = 0; i < 10; i++) *pa++ = i * i;
  pa = &a[0];
  for (int i = 0; i < 10; i++) cout << *pa++;
  cout << endl;
}
```

Bild 4.9 stellt den Zusammenhang zwischen solchen C-Vektoren und Adressen sowie die jeweiligen Zugriffsmöglichkeiten über Indizierung und Adressrechnung dar.

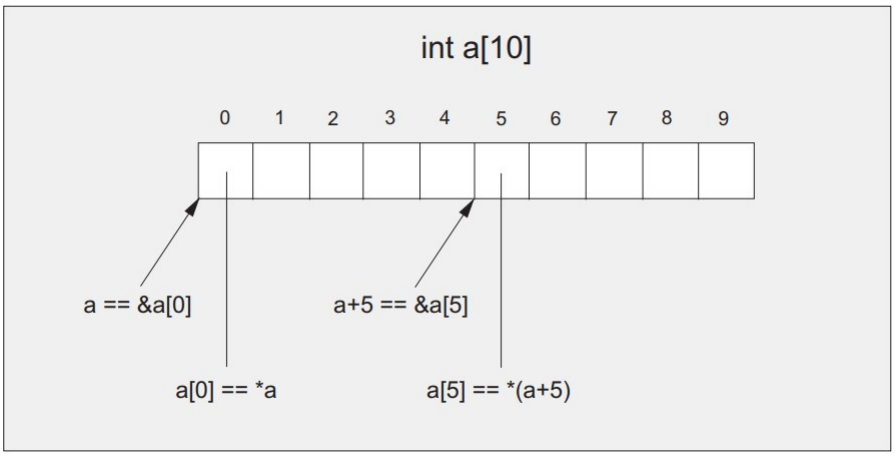

Bild 4.9: Die Dualität zwischen Arrays und Adressen

- Die Vereinbarung `int *pa;` bedeutet: *Der Inhalt von* `pa` *ist vom Typ* `int`, *d.h.* `pa` *selbst ist der entsprechende Zeiger auf diese Werte (die Adresse, unter der der Wert abgelegt ist).*
- Mit `pa = &a[0]` wird der Zeiger auf das erste Element des Arrays a gesetzt.
- Das Gleiche kann durch die Anweisung `pa = a` erreicht werden. Das liegt daran, dass ein Array durch seine Adresse dargestellt wird.
- Somit kann der Zugriff z.B. auf das i-te Element des Arrays `a` entweder über die Notation `a[i]` oder über die Notation `*(a+i)` erfolgen oder, wegen der Zuweisung `pa = a`, natürlich auch über `*(pa+i)`.
- `pa` und `a` sind beide vom Typ *Zeiger auf int*, aber es gibt einen wesentlichen Unterschied: `pa` ist ein *freier* Zeiger, dessen Wert verändert werden kann (z.B. `pa++`), `a` ist dagegen konstant, d.h. fest auf den Anfang des Arrayss gerichtet und kann nicht manipuliert werden.
- Bei der Adressrechnung `a+i`, (z.B. `a+3`) wird die Adresse *typgerecht* um `i` (z.B. hier drei) Schritte, nicht zwangsläufig um drei Byte, erhöht.
- Es wird nicht geprüft, ob ein Index innerhalb des erlaubten (d.h. vereinbarten) Bereiches liegt.
- Adressrechnungen werden weder vom C++-Compiler noch zur Laufzeit überprüft.

4.3 *Werte oder Referenzen*, die Vor- und Nachteile

Der C++-Programmierer muss ständig entscheiden, ob er Objekte direkt verwendet *(Wertesemantik)* oder ob er sie indirekt über Zeiger oder Referenzen verwendet *(Referenzsemantik).*

Der Java-Programmierer braucht sich keine Gedanken zu machen, bei Objekten elementarer Datentypen wird automatisch die *Wertesemantik* verwendet und bei Objekten strukturierter (selbst definierter) Datentypen wird automatisch die *Referenzsemantik* verwendet.

Beide Konzepte sind vernünftig, wenn man den jeweiligen Kontext betrachtet:

- Die Sprache C++ hat ihren Einsatzschwerpunkt in Bereichen wie Systemprogrammierung und eingebetteten Systemen (embedded systems), hier steht Laufzeit- und Speicherplatz-Effizienz im Vordergrund. Der Programmierer kann durch die jeweils richtige Wahl die Effizienz verbessern.
- In Java wird die Wahl jeweils durch das System getroffen, die Einfachheit und Klarheit der Sprache stehen im Vordergrund.

Um nun einerseits das Java-Konzept zu verstehen und andererseits auch in der Lage zu sein, Sprachen wie C++ souverän zu beherrschen, werden die Hintergründe im Folgenden etwas detaillierter erläutert und entsprechende Richtlinien formuliert.

Grobe Richtlinie

Solange bei der Programmierung nicht die Laufzeit- und Speicherplatz-Effizienz im Vordergrund steht, ist es durchaus sinnvoll, auch in C++ die Regel zu befolgen, die in Java fest vorgegeben ist, d.h. elementare Objekte direkt per Wert und strukturierte Objekte indirekt per Zeiger zu verwenden.

Detaillierte Betrachtung

Als Basis für die folgenden Überlegungen rufen wir in Erinnerung:

- Referenzsemantik – also indirekter Zugriff – wird primär auf Heap-Objekte angewendet und hat in der Kombination folgende Vorteile:
  - Objekte werden zur Laufzeit (statt zur Compilezeit) erzeugt und freigegeben, z.B. abhängig von aktuellen Berechnungen oder Eingaben.
  - Objekte können mehrfach referenziert und somit auch gleichzeitig in verschiedenen Containern verwaltet werden, um sie so z.B. nach unterschiedlichen Kriterien zu sortieren.
- Wertesemantik – also direkter Zugriff – ist ausschließlich auf Stack-Objekte möglich, die Objekte können zur Laufzeit nicht beliebig erzeugt und freigegeben werden.

Bei der Referenzsemantik gehört zu jedem Objekt noch eine Referenz (z.B. ein Zeiger). Das Erzeugen und Freigeben kostet etwas Laufzeit, deshalb ist es in der Regel effizienter, sehr kleine Objekte direkt zu benutzen. Bei größeren Objekten fällt dieser Überhang kaum ins Gewicht und die Vorteile der höheren Flexibilität überwiegen bei Weitem. Bild 4.10 zeigt ein Beispiel zum Thema *Werte oder Referenzen*, das die Vorteile der Referenzsemantik anschaulich demonstriert.

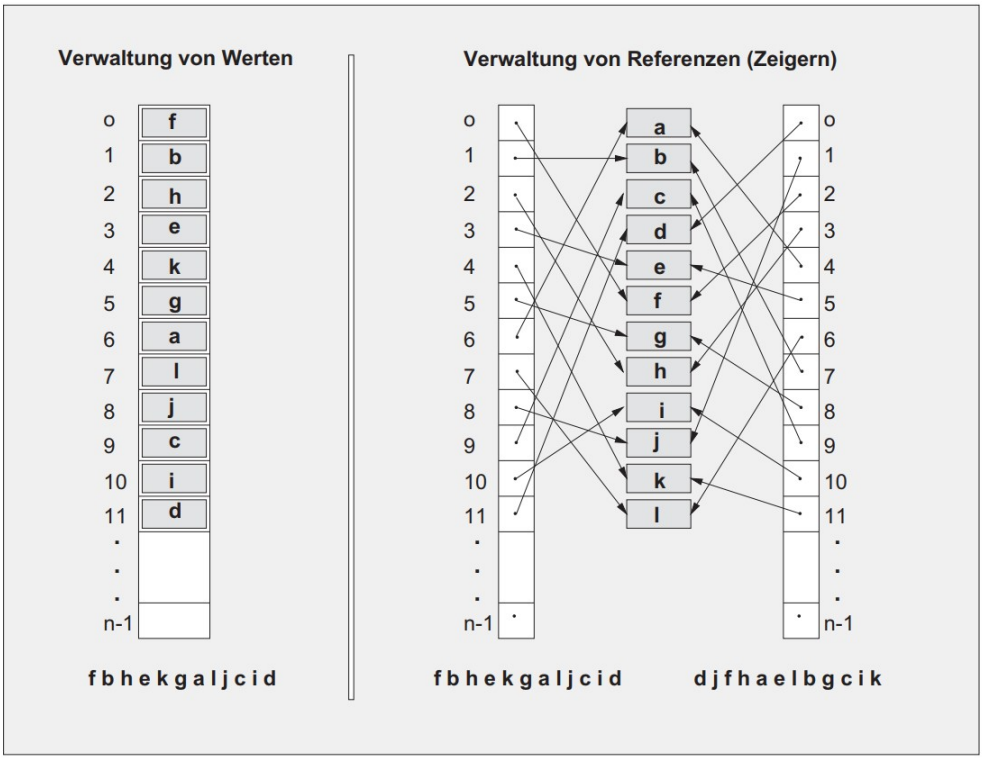

Bild 4.10: Sortieren von Objekten per Wert und per Referenz (Zeiger)

Links zeigt das Bild ein Array der Größe n, das zwölf Objekte als Werte verwaltet. Wir gehen davon aus, dass die Objekte ursprünglich z.B. in alphabetischer Reihenfolge `a b c ... k l` vorlagen und jetzt entsprechend einem inneren Kriterium – z.B. ihrem Preis – in sortierter Reihenfolge vorliegen. Zum Sortieren sind viele Vertauschungen erforderlich, bei denen die Objekte jeweils kopiert werden müssen.

Rechts zeigt das Bild zwei Arrays der gleichen Größe, die jeweils Referenzen (Zeiger) auf Objekte verwalten. Das linke der beiden hat die Objekte in der gleichen Reihenfolge sortiert, und es ist die gleiche Anzahl von Vertauschungen erforderlich, doch werden jeweils nur Referenzen (Zeiger) kopiert. Das rechte Array verwaltet die selben Objekte und hat sie zusätzlich nach einem anderen inneren Kriterium – z.B. nach ihrer Artikelnummer – sortiert.

Wenn wir die beiden hier dargestellten Konzepte *Verwaltung von Werten* und *Verwaltung von Referenzen* vergleichen, so können wir folgende Vorteile für die Referenzen feststellen:

- Die *Lösung mit Referenzen* ist erheblich *schneller*, und zwar etwa um den Faktor *Zeit zum Kopieren eines Objektes* zu *Zeit zum Kopieren einer Referenz* (dabei vernachlässigen wir die Zeit für die Vergleiche).

- Die *Lösung mit Referenzen* ist *flexibler*, weil sie es ermöglicht, die nur einmal vorhandenen Objekte nach verschiedenen Kriterien zu sortieren.
- Die *Lösung mit Referenzen* erfordert in der Regel *weniger Speicherplatz*, es wird nur Speicherplatz für die aktuell vorhandenen Objekte benötigt, bei der *Wertelösung* müssen wir Speicherplatz für die maximal mögliche Anzahl von Objekten vorhalten (den Speicheraufwand für die Referenzen vernachlässigen wir bei dieser Abschätzung).

Tipp

Das Beispiel kann verallgemeinert werden, wann der Einsatz von Referenzen angebracht ist: Immer wenn es ein Objekt `begehrt` gibt, das von mehreren anderen Objekten `nutzer1`, `nutzer2` usw. benötigt wird, ist es im Allgemeinen wenig sinnvoll, wenn sich jeder der Nutzer eine Kopie von dem Objekt `begehrt` anlegt. Das kostet Speicherplatz. Entscheidender ist aber, dass es aufwändig ist, wenn alle Kopien von `begehrt` konsistent gehalten werden müssen. In dem Netzplanungsbeispiel ist es wenig sinnvoll, wenn in jedem Vorgang eine Liste der Nachfolger-Vorgänge existiert und diese Liste die einzelnen Vorgänge jeweils als Wert und nicht als Referenz enthält. Ändert sich z.B. im Laufe der Planung der früheste Anfang eines Vorgangs `v1`, so muss der früheste Anfang bei allen Kopien von `v1` ebenfalls geändert werden. Wird lediglich eine Referenz auf `v1` abgelegt, ist der früheste Anfang an nur einer Stelle anzupassen.

In C++ bedeutet die Verwendung von Referenzen allerdings, dass sich der Entwickler selbst um die Verwaltung des Heap-Speichers kümmern muss. Insbesondere die Freigabe des Speichers mit `delete` ist eine häufige Fehlerquelle, die dann auch noch schwer zu lokalisieren ist. In Java unterstützt der Garbage Collector den Entwickler, Heap-Objekte, die nicht mehr referenziert werden, werden automatisch recycelt. In Kap. 6 werden wir sehen, wie die Verwaltung des Heap-Speichers auch in C++ durch Abstrakte Datentypen unter Verwendung von Destruktoren und durch so genannte intelligente Zeiger vereinfacht werden kann.

Tipp

Entwurfs-Regeln (Design Rules)

Bei der Entscheidung „Wert oder Referenz“ sollten wir uns von folgenden Regeln leiten lassen:

- Im ersten Schritt können die Vorgaben von Java auch in C++ umgesetzt werden, d.h. Wertesemantik für Objekte elementarer Datentypen und Referenzsemantik für Objekte strukturierter Datentypen.
- Im zweiten Schritt können zur Verringerung des Fehlerrisikos (Reduktion der Komplexität) auch strukturierte Datentypen auf dem Programm-Stack angelegt werden (C++), wenn das Objekt nur von einem anderen Objekt benötigt wird.
- In C++ wird in der Regel kein Garbage Collector verwendet. Das bedeutet z.B., dass es nicht sinnvoll ist, einen Zeiger oder eine Referenz auf ein lokal in einer Funktion erzeugtes Objekt zurückzugeben, z.B. in der Form

  ```
  Objekt* f(...) { Objekt* temp = new Object; ... return temp; }
  ```

  oder in der Form

```
Objekt& f(...) { Objekt* temp = new Object; ... return *temp; }
```

So ein Programmierstil provoziert Speicherlecks.

- Beim Programmieren in C++ ist bei der Verwendung von Referenzsemantik besonders darauf zu achten, dass alle reservierten Heap-Objekte auch explizit, d.h. durch Verwenden der Operatoren `delete` und `delete[]`, wieder freigegeben werden.

Im Abschn. 4.5 wird das Thema durch zwei Beispiele vertieft.

4.4 Zeiger auf Funktionen

Zeiger auf Funktionen sind Datentypen *fortgeschrittener, aber nicht objektorientierter* Programmiertechnik. Diese Technik wird deshalb nicht von Java, aber z.B. von Sprachen wie Pascal, Modula2, Oberon und C angeboten. In Kap. 7.2.1 werden Funktionszeiger aber für unsere Motivation der Objektorientierung eine große Rolle spielen, daher empfehlen wir auch dem Leser, der nur an Java interessiert ist, die Lektüre dieses Abschnitts.

Ein Zeiger auf eine Funktion kann auch wieder als Element in einem Array als Komponente in einer Struktur oder als Parameter einer Funktion verwendet werden. Damit ergeben sich interessante Anwendungsmöglichkeiten, die im Folgenden skizziert werden.

Zeiger auf Funktionen in Arrays und Strukturen

Unter Verwendung von `typedef` lassen sich Datentypen als **Zeiger auf Funktionen** definieren, z.B.:

```
typedef void (*func1) (void);
```

Hier wurde der Typ `func1` (ein Zeiger auf eine Funktion ohne Parameter, die nichts zurückgibt) deklariert.

```
typedef int (*func2) (char*, int);
```

Hier wurde der Typ `func2` deklariert, d.h. ein Zeiger auf eine Funktion mit zwei Parametern (`char*` und `int`), die einen `int`-Wert zurückgibt.

```
typedef double (*func3) (double);
```

Hier wurde der Typ `func3` deklariert, d.h. ein Zeiger auf eine Funktion mit einem `double`-Parameter, die ebenfalls einen `double`-Wert zurückliefert.

Unter Verwendung dieser selbst definierten Typen können dann elementare und strukturierte Variablen definiert werden, z.B.:

```
func2 x, y, z;
func3 a[10];
struct {
  func1 f1;
  func2 f2;
  func3 f3;
} xyz;
```

Der folgende C++-Text 4.9 demonstriert die Verwendung von Arrays, deren Elemente Zeiger auf Funktionen sind.

C++ 4.9: Zeiger auf Funktionen, Beispiel

```
double f2(double x) {
 return x*x;}
double f3(double x) {
 return x*x*x;}
double f4(double x) {
 return x*x*x*x;}
double f5(double x) {
 return x*x*x*x*x;}
```

```
void vektorDemo() {
  typedef double (*func) (double);
  func liste[4] = { f2, f3, f4, f5 };
  for (int i=0; i<4; i++) {
    double y = liste[i](5);
    cout << y << endl;
  }
}
```

(WertRef/FunkZeiger/vektor.cpp)

Um Funktionszeiger zu definieren, sind Typdefinitionen mit `typedef` zwar nicht zwingend erforderlich – im obigen Programmtext hätte `liste` auch wie folgt definiert werden können –, allerdings erhöhen die Typdefinitionen erheblich die Lesbarkeit.

```
double (*liste[4])(double) = { f2, f3, f4, f5 };
```

Im C++-Text 4.10 wird die Verwendung von Strukturen, deren Komponenten Zeiger auf Funktionen sind, demonstriert.

C++ 4.10: Zeiger auf Funktionen, komplexes Beispiel

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

int fsum2(int a, int b)  {
  return a+b;
}
int fsum3(int a, int b, int c) {
  return a+b+c;
}
int fsum4(int a, int b, int c, int d) {
  return a+b+c+d;
}
```

```
struct {
  int (*sum2) (int, int);
  int (*sum3) (int, int, int);
  int (*sum4) (int, int, int, int);
} xyz  = { fsum2, fsum3, fsum4 };

void structDemo(void) {
  int i = xyz.sum2(1, 2);
  int j = xyz.sum3(1, 2, 3);
  cout << i << " " << j << " "
    << xyz.sum4(1, 2, 3, 4) << endl;
}
```

(WertRef/FunkZeiger/struktur.cpp)

Zeiger auf Funktionen als Funktionsparameter

Eine fortgeschrittene und in vielen Fällen nützliche Programmiertechnik ist die Übergabe von Zeigern auf Funktionen als Funktionsparameter. Man schreibt z.B. eine Funktion zur Berechnung der Nullstelle einer anderen Funktion, die man dann als Parameter übergibt, z.B.

```
double nullstelle(double x1, double x2, double (*f) (double x));
/* . . . */
double x0 = nullstelle(6, 7, sin); /* x0 == 2 * pi */
```

Man übergibt der Funktion `nullstelle` je einen x-Wert links und rechts der Nullstelle und einen Zeiger auf die Funktion $f(x)$, für die die Bestimmung durchgeführt werden soll.

Ein anderes Beispiel, das in Listing 4.11 dargestellt ist, betrifft eine Sortierfunktion, die eine Liste nach verschiedenen Kriterien sortieren kann. Das jeweils gewünschte Ordnungskriterium wird ihr als aktueller Funktionsparameter übergeben:

- Die Liste wird nacheinander gemäß vier verschiedenen Kriterien sortiert.
- Dafür wird eine einzige Sortierfunktion `sort` verwendet!
- Das Ordnungskriterium wird jeweils über eine Funktion definiert, die zwei Einträge vergleicht.
- Diese Vergleichsfunktion wird der Funktion `sort` als Parameter `comp` (*d.h. compare*) übergeben.
- Der Parameter `comp` hat die Form (den Typ): `int (*comp) (Student, Student);`, d.h. Zeiger auf eine Funktion, die ein `int` zurückliefert und zwei Parameter vom Typ `Student` erwartet.

Die Ausgabe des Programms in die Datei `AUS` ist:

```
8965123 Meier 9 25
9215046 Deng 4 22
8965122 Schulz 9 26
9167111 Xyz 5 21

8965122 Schulz 9 26
8965123 Meier 9 25
9167111 Xyz 5 21
9215046 Deng 4 22

9215046 Deng 4 22
8965123 Meier 9 25
8965122 Schulz 9 26
9167111 Xyz 5 21

9215046 Deng 4 22
9167111 Xyz 5 21
8965123 Meier 9 25
8965122 Schulz 9 26

9167111 Xyz 5 21
9215046 Deng 4 22
8965123 Meier 9 25
8965122 Schulz 9 26
```

Die hier benutzte Übergabe von *Zeigern auf Funktionen* als Funktionsparameter wurde als fortgeschrittene Programmiertechnik bezeichnet. Wir werden Funktionszeiger in Kap. 7.2.1 benutzen, um objektorientierte Konzepte einzuführen. Danach ist der direkte Umgang mit dem komplizierten Konstrukt des Funktionszeigers nicht mehr erforderlich, sondern die objektorientierte Sprache nimmt uns ganz wesentliche Arbeitsschritte ab.

C++ 4.11: Zeiger auf Funktionen, Beispiel Sortierung

```
#include <iostream>
#include <fstream> //Dateiausgabe
#include <cstring> //C-Stringfunktionen
using namespace std;

typedef struct {
  int matrNr;
  char name[12];
  int semester;
  int alter;
} Student;
typedef Student SList[100];

int cmp1 (Student a, Student b);
int cmp2 (Student a, Student b);
int cmp3 (Student a, Student b);
int cmp4 (Student a, Student b);

void sort(SList sl,
   int (*comp) (Student a, Student b));

void printList (ostream& f, SList sl);

void sortDemo() {
  ofstream aus("AUS", ios::out);

  SList li = { 8965123, "Meier", 9, 25,
               9215046, "Deng", 4, 22,
               8965122, "Schulz",9, 26,
               9167111, "Xyz", 5, 21,
               0,     "  ",  0, 0
             };
  printList(aus, li);
  sort(li, cmp1); printList(aus, li);
  sort(li, cmp2); printList(aus, li);
  sort(li, cmp3); printList(aus, li);
  sort(li, cmp4); printList(aus, li);

  aus.close();
}
```

```
int cmp1 (Student a, Student b) {
  return a.matrNr - b.matrNr; }
int cmp2 (Student a, Student b) {
  return strcmp(a.name, b.name); }
int cmp3 (Student a, Student b) {
  return a.semester - b.semester; }
int cmp4 (Student a, Student b) {
  return a.alter - b.alter; }

void sort(SList sl,
  int (*comp) (Student a, Student b)) {
  bool flag;
  Student st;
  do {
    flag = false;
    for (int i=1; sl[i].alter!=0;i++) {
      int n = comp(sl[i-1], sl[i]);
      if (n > 0) {
        flag = true;
        st = sl[i-1];
        sl[i-1] = sl[i];
        sl[i] = st;
      }
    }
  }
  while (flag);
}

void printList(ostream& f, SList sl) {
   for (int i=0; sl[i].alter!=0;i++) {
     f << sl[i].matrNr << " "
       << sl[i].name << " "
       << sl[i].semester << " "
       << sl[i].alter << endl;
   }
   f << endl;
}
```

(WertRef/FunkZeiger/sort.cpp)

4.5 Anwendungsbeispiele

Wir werden in diesem Abschnitt zunächst unsere Netzplanung verfeinern. Anschließend gehen wir am Beispiel von *linear verketteten Listen* auf das Thema *dynamische Datenstrukturen* ein, bei denen die Anzahl der speicherbaren Elemente dynamisch zur Laufzeit verändert wird.

4.5.1 Effizienzsteigerung der Netzplanung

Das Kap. 2 haben wir bzgl. des Netzplanungs-Beispiels mit einer sehr speicherintensiven Lösung beendet. Wir mussten von vornherein wissen, wie viele Vorgänge ein Netzplan maximal haben kann. Wir haben 100 gewählt, womit unser Programm Netzpläne mit mehr als 100 Vorgängen nicht mehr handhaben kann.

Um dieses Problem zu lösen, verwenden wir statt eines Arrays (linke Spalte des folgenden Listing) nun einen Zeiger auf Vorgänge (rechte Spalte) in der Struktur `Netz`:

```
struct Netz {
  int anzahl;
  Vorgang vorg[MAX]; // Knoten
  /* ... */
};
```

```
struct Netz {
  int anzahl;
  Vorgang* vorg; // Knoten
  /* . . . */
};
```

Vor Benutzung des Zeigers `vorg` müssen wir allerdings noch Speicherplatz vom Laufzeitsystem mit dem Operator `new[]` anfordern. Dies geschieht am besten am Anfang der Funktion `initNetzplanAllg`. Am Ende eines jeden Tests wird der Heap-Speicher nicht mehr benötigt und daher mit dem Operator `delete[]` wieder freigegeben. Hierzu erstellen wir eine eigene Funktion `freigabeNetz`, die die Aufräumarbeiten für ein Netz übernimmt und am Ende eines Tests aufgerufen wird:

```
struct Netz {

 /* . . . */

 void initNetzplanAllg(Netz* netz){
   netz->vorg = new Vorgang[netz->anzahl];
   for (int i=0; i < netz->anzahl; ++i) {
     initVorgang(&netz->vorg[i]);
     for (int j=0; j<netz->anzahl;++j)
       netz->nachf[i][j]=false;
   }
 }
```

```
 void freigabeNetz(Netz* netz){
   delete[] netz->vorg;
 }
 bool testFall1() {
   Netz netz;
   initNetzplanTest5(&netz);
   /* . . . */
   freigabeNetz(&netz);
   return ok;
 }
 /* . . . */
};
```

Ein weiteres Problem unserer bisherigen Netzimplementierung ist, dass wir jedes Mal ein sehr dünn besetztes zweidimensionales Array `nachf` anlegen (es sind nur wenige Einträge mit Wert `true` vorhanden), das in unserem Beispiel 10000 Werte enthält – eine enorme Speicherplatzverschwendung. Statt eines zweidimensionalen Arrays `nachf` im `Netz` verwenden wir deshalb in jedem `Vorgang` ein Array, das die Nachfolger des Vorgangs enthält, und ein weiteres Array, das die Vorgänger speichert. Natürlich speichern wir nicht die Vorgänge selbst, sondern einen Zeiger darauf. Eine Implementierung hiervon ist im linken Teil des folgenden Listings gezeigt:

```
struct Vorgang {
  double dauer;
  double fruehanf;
  double spaetend;
  Vorgang* vorgaenger[MAX];
  Vorgang* nachfolger[MAX];
};
```

```
struct Vorgang {
  double dauer, fruehanf, spaetend;
  Vorgang** vorgaenger;
  int anzVorg;
  Vorgang** nachfolger;
  int anzNachf;
};
```

In der Summe erzielen wir mit dieser Lösung aber keine Verbesserung. Wir würden jetzt für jeden Vorgang `2 * MAX` Zeiger auf Vorgänge abspeichern, von denen aber nur wenige wirklich benötigt würden, d.h. maximal die Anzahl der Vorgänge im gesamten Netz. Da unsere Datenstruktur `Vorgang` aber für beliebige Netzgrößen verwendbar sein soll, greifen wir auf den gleichen *Trick* zurück wie oben beim `Netz`: Statt eines Zeiger-Arrays verwenden wir einen Zeiger auf die Zeiger, d.h. `nachfolger` ist vom Typ `Vorgang**` (siehe rechter Teil des Listings).

So richtig überzeugend ist diese Lösung aber immer noch nicht, da wir jetzt zwar *nur* noch `2 * anzahl` Einträge haben, aber die meisten von ihnen werden auch hier gar nicht benötigt, da ja nicht jeder Vorgang ein Nachfolger von jedem anderen Knoten ist. Wir vergrößeren die Arrays, auf die `vorgaenger` bzw. `nachfolger` verweisen, daher jeweils erst beim Einfügen eines neuen Vorgängers bzw. Nachfolgers. Hierzu müssen wir dann jeweils auch die (aktuelle) Anzahl der Vorgänger bzw. Nachfolger im `Vorgang` abspeichern.

Wir führen einen neuen Datentyp `DynVorgangsArray` ein, um die Zeiger auf die Vorgänge und die Anzahl der Vorgänge zusammenzufassen:

```
struct Vorgang; // Vorwaertsreferenz

struct DynVorgangsArray {
  Vorgang** array;
  int anz;
};
```

```
struct Vorgang {
  double dauer, fruehanf, spaetend;
  DynVorgangsArray vorgaenger;
  DynVorgangsArray nachfolger;
};
```

```
#include "DynVorgangsArray.h"
```

Entsprechend können wir nun das Array `nachf` aus `Netz` entfernen. Die Konstante `MAX` wird damit nicht mehr benötigt. An allen Stellen, an denen bisher `nachf` verwendet wurde, verwenden wird nun stattdessen die entsprechenden Arrays `nachfolger` und `vorgaenger` aus `Vorgang`.

Statt direkt die Nachfolger-Beziehung im Array `nachf` einzutragen, verwenden wir hierfür die neu zu implementierende Funktion `setzeNachf`. Die Benutzung von `setzeNachf` zeigt das folgende Listing in der neuen Implementierung von `initNetzplanTest5`:

```
void initNetzplanTest5(Netz* netz){
  /* .. */
  setzeNachf(netz, 0, 1);
  setzeNachf(netz, 0 ,2);
  /* .. */
}
```

Um einen Vorgang mit Namen vv als Nachfolger beim Vorgang vn einzutragen, delegiert das Netz diese Aufgabe an die Funktion setzeNachf mit zwei Vorgängen als Parameter. Sie ruft zweimal die Funktion fuegeHinzu auf:

```
/** Vorgang Nr. vv wird als Nachfolger von Vorgang Nr. vn eingetragen. */
void setzeNachf(Netz* n, int vv, int vn) {
  setzeNachf(&(n->vorg[vv]),&(n->vorg[vn]));
}
void setzeNachf(Vorgang* vv, Vorgang* vn) {
  fuegeHinzu(vv->vorgaenger, vn); fuegeHinzu(vn->nachfolger, vv);
}
```

Die Funktion fuegeHinzu trägt einen Nachfolger oder Vorgänger in das sich dynamisch zur Laufzeit vergrößernde Array ein. Der Speicherplatz auf dem Heap, den arr belegt, wird jeweils um einen Eintrag vergrößert, wie das folgende Listing zeigt:

```
void fuegeHinzu
    (DynVorgangsArray* arr, Vorgang* v) {
  // Speicherplatz fuer einen mehr
  Vorgang** tmp =
        new Vorgang*[arr->anz+1];
  // bisherigen Inhalt retten
  for (int i=0; i < arr->anz; ++i) {
    tmp[i] = arr->array[i];
  }
  // Neuen Eintrag eintragen
  tmp[arr->anz] = v;
  ++arr->anz;
  // bisherigen Speicher freigeben
  delete [] arr->array;
  // Zeiger umbiegen
  arr->array = tmp;
}
```

Nun müssen wir noch den lesenden Zugriff auf das Array nachf modifizieren, um dieses Array endgültig entfernen zu dürfen. Der linke Teil des folgenden Listings zeigt nochmals den bisherigen Zugriff und der rechte Teil die Ersetzung durch den Aufruf der neuen Funktion istVorgVon:

```
void berechneVorwaerts(int v, Netz *netz) {
  /* ... */
    if (netz->nachf[j][v]) {
      passeFruehanfAn(v1, &netz->vorg[j]);
  /* ... */
```

```
void berechneVorwaerts(int v, Netz *netz) {
  /* ... */
    if (istVorgVon(netz, j, v)){
      passeFruehanfAn(v1, &netz->vorg[j]);
  /* ... */
```

Der folgende Codeausschnitt zeigt die Implementierung der Funktion istVorgVon, die ihre Aufgabe wiederum an die Funktion istElem delegiert:

```
bool
 istVorgaengerVon(
    const Vorgang* vv,
    const Vorgang* vn )
{
  return istElem(vn, vv->vorgaenger);
}
```

```
bool istElem(const Vorgang* v,
             const DynVorgangsArray* arr) {
  for (int i=0; i < arr->anz; ++i) {
    if (arr->array[i] == v) { return true;}
  }
  return false;
}
```

Außerdem müssen wir noch die Initialisierung in den Klassen Netz, Vorgang und DynVorgangsArray anpassen:

```
/** Speicheranforderung für die Vorgaenge
auf dem Heap und anschließende
Initialisierung */

void initNetzplanAllg(Netz* netz){
  netz->vorg = new Vorgang[netz->anzahl];
  for (int i=0; i < netz->anzahl; ++i) {
    initVorgang(&netz->vorg[i]);
  }
}
```

```
void initVorgang(Vorgang* v) {
  init(v->vorgaenger);
  init(v->nachfolger);
  v->spaetend = 0; v->fruehanf = 0;
  v->dauer = 0;
}
void init(DynVorgangsArray* arr) {
  arr->anz = 0;
  arr->array = NULL;
}
```

Entsprechendes gilt für die Freigabe des Heap-Speichers:

```
void freigabeNetz(Netz* netz){
  for (int i=0; i < netz->anzahl; ++i) {
    freigeben(&netz->vorg[i]);
  }
  delete[] netz->vorg;
}
```

```
void freigeben(Vorgang* v) {
   freigeben(v->vorgaenger);
   freigeben(v->nachfolger); }
void freigeben(DynVorgangsArray* arr) {
  delete[] arr->array;
  arr->array = NULL; arr->anz = 0; }
```

4.5.2 Dynamische Datenstrukturen

Um Speicherplatz zu sparen, haben wir das Array der Vorgänger bzw. Nachfolger eines Vorgangs im Netzplanungsbeispiel dynamisch bei Bedarf vergrößert. Statt eines Arrays hätte man hierzu auch eine *linear verkettete Liste* verwenden können, die in diesem Abschnitt als Beispiel für verkettete Strukturen vorgestellt wird.

Verkettete Strukturen (Listen, binäre Bäume und andere): Als *dynamische Datenstrukturen* werden Datenstrukturen bezeichnet, die man wie folgt charakterisieren kann:

- Die Strukturen verändern sich hinsichtlich Größe und Form während ihrer Lebensdauer (Wachsen, Schrumpfen, ...).
- Der benötigte Speicherplatz wird auf dem *Programm-Heap* dynamisch verwaltet und dem aktuellen Bedarf angepasst.
- Die einzelnen Strukturelemente sind durch *Zeiger* (in C++) oder durch *Referenzen* (in Java) verknüpft.

Typische Beispiele für solche Strukturen sind *lineare Listen*, *binäre Bäume* und *Vielweg-Bäume*.

Beispiel: Einfach verkettete Listen

Definition eines Knotens: Listen werden z.B. zur Verwaltung von Datensätzen verwendet, die jeweils aus einer Schlüsselinformation und irgendwelchen weiteren Daten bestehen. Jeder Datensatz wird in einem *Knoten (node)* abgelegt. Die Datensätze sind anwendungsspezifisch und werden hier außer Acht gelassen. Als einfachen Stellvertreter für einen beliebigen Datensatz wählen wir der Einfachheit halber ganze Zahlen und nennen sie `key`. Damit ergeben sich folgende Datendefinitionen als Basis für die weiteren Ausführungen:

```
struct Node {
  int key;
  Node* nxt;
};
// Verweis auf Node heißt NodeRef:
typedef Node* NodeRef;
```

```
public class Node {
  int key;
  Node nxt;
};
```

Unterschiede zwischen Java und C++: In C++ wird zwischen dem Datentyp des zu speichernden Datums (Typ `Node`) und dem Verweis darauf unterschieden (Typ `NodeRef` bzw. `Node*`). Java bezeichnet beides mit dem gleichen Typ `Node`. Aus dem Zusammenhang ergibt sich, ob man gerade das zu speichernde Datum im Heap-Speicher (z.B. bei `new`) oder die Referenz auf dem *Programm-Stack* meint, die in den Heap verweist.

Wesentliche Operationen: Wichtige Basisoperationen für das Arbeiten mit Listen sind folgende Funktionen:

- Generieren einer Liste (create);
- Auflösen einer Liste (dispose);
- Einfügen eines Knotens (insert);
- Löschen eines Knotens (delete).

Generieren einer Liste

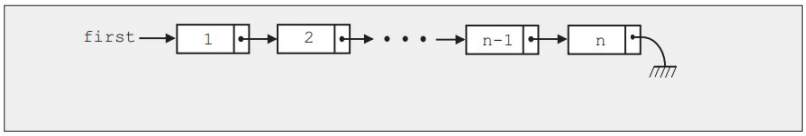

Bild 4.11: Generieren einer Liste

Bild 4.11 zeigt eine verkettete Liste, die die Zahlen `1` bis `n` in aufsteigender Reihenfolge enthält. Der Code zum Generieren dieser Struktur ist so aufgebaut, dass zuerst das erste Element und dann in einer Schleife die Restliste erzeugt wird. Die Liste wird mit einem `NULL`-Zeiger bzw. einer `null`-Referenz abgeschlossen:

```
NodeRef first = new Node; first->key = 1;
NodeRef p = first;
for (int i=2; i < n+1; i++) {
  p->nxt = new Node;
  p = p->nxt; p->key = i;
}
p->nxt = NULL;
```

```
Node first = new Node(); first.key = 1;
Node p = first;
for (int i=2; i < n+1; i++) {
  p.nxt = new Node();
  p = p.nxt; p.key = i;
}
p.nxt = null;
```

Ist die Sonderbehandlung für das erste Element erforderlich, oder geht es auch einfacher? Es gibt eine einfachere Lösung, wenn man die Liste *rückwärts* generiert; es lohnt sich, darüber nachzudenken, warum die Sonderbehandlung hier entfällt:

```
NodeRef first = NULL;
for (int i=n; i >- 1; i--) {
  NodeRef q = new Node;
  q->nxt = first;
  first = q; first->key = i;
}
```

```
Node first = null;
for (int i=n; i >- 1; i--) {
  Node q = new Node();
  q.nxt = first;
  first = q; first.key = i;
}
```

Auflösen einer Liste

Es sei `first` wieder der Zeiger auf den ersten Knoten einer Liste; siehe in Bild 4.11. In C++ wird die Liste durchlaufen und Knoten für Knoten explizit freigegeben. In Java reicht es, den Zeiger `first` auf den Wert `null` zu setzen, der Rest wird dann vom *Garbage Collector* erledigt, da die Liste dann nicht mehr referenziert wird:

```
while (first != NULL) {
  NodeRef hlp = first->nxt;
  delete first;
  first = hlp;
}
```

```
first = null;
```

Aber auch auf die Zuweisung `first = null` kann verzichtet werden, denn der Garbage Collector wird aktiv, wenn der Gültigkeitsbereich von `first` verlassen wird.

Einfügen von Knoten mit Definition der Anforderungen und Tests

Wir wollen eine einfach verkettete Liste entwickeln, die mit den Funktionen *ein Element am Anfang einfügen* (`insBeg`) und *ein Element am Ende einfügen* (`insEnd`) ausgerüstet ist. Bevor wir mit der Implementierung der Liste im Detail fortfahren, wollen wir die Anforderungen zunächst weiter detaillieren. Dies geschieht am einfachsten, indem wir uns geeignete Tests überlegen.

- In eine leere Liste wird am Anfang die Zahl `5` eingetragen. Anschließend ergibt sich die Liste, die nur die `5` enthält.
- In eine Liste mit den Elementen `3, 5, 8` wird am Anfang eine `1` eingetragen. Anschließend ergibt sich die Liste `1, 3, 5, 8`.
- In eine leere Liste wird am Ende die Zahl `5` eingetragen. Anschließend ergibt sich die Liste, die nur die `5` enthält.
- In eine Liste mit den Elementen `3, 5, 8` wird am Ende eine `1` eingetragen. Anschließend ergibt sich die Liste `3, 5, 8, 1`.

Die in den Tests definierten Aktionen lassen sich verallgemeinern, um damit dann die beiden entsprechenden Funktionen `insBeg` und `insEnd` wie folgt zu spezifizieren:[6]

$$\begin{aligned} insBeg(<>, e) &\rightarrow < e > \\ insBeg(< e_1, ..., e_n >, e_0) &\rightarrow < e_0, e_1, ..., e_n > \\ insEnd(<>, e) &\rightarrow < e > \\ insEnd(< e_1, ..., e_n >, e_{n+1}) &\rightarrow < e_1, ..., e_n, e_{n+1} > \end{aligned}$$

Daraus lassen sich unmittelbar Testfunktionen ableiten. Eine der Testfunktionen ist beispielhaft in Listing 4.12 wiedergegeben, aber vorab sind noch einige grundsätzliche Überlegungen zur Implementierung zu beachten.

Implementierung eines kleinen Listen-Programms

Wir könnten die Funktion `insBeg` mit der folgenden Schnittstelle implementieren:

```
void insBeg(NodeRef& st, int val);
```

```
void insBeg(Node st, int val);
```

Der Parameter `st` (für start) soll dann jeweils auf den Listenanfang verweisen. Allerdings ist beim Einfügen am Anfang der Liste der Listenanfang `st` auch zu modifizieren. Diese Lösung scheitert aber in Java, da wir `st` nicht selbst auch noch als Referenz an die Funktion `insBeg` übergeben können. Damit bliebe der Listenanfang nach Beendigung der Funktion immer noch der gleiche wie vor Ausführung der Funktion. Wir können den (möglicherweise veränderten) Listenanfang entweder als Rückgabewert liefern oder über den Referenzparameter wieder über eine Hilfsklasse simulieren.

Wir führen daher einen neuen Datentyp `Liste` ein, der als einziges Attribut den Listenanfang vom Typ `NodeRef` bzw. `Node` enthält. Wir könnten den Datentyp aber auch noch um ein nützliches Attribut `anzahl` zum Zählen der Listenelemente ergänzen.

```
struct Liste{
  NodeRef start;
};
```

```
public class Liste{
  Node start;
}
```

Damit ergibt sich für das Einfügen am Listenanfang:

```
/** val wird in die Liste li am
   Anfang eingefuegt.    */
void insBeg(Liste& li, int val) {
  NodeRef oldSt = li.start;
  li.start = new Node;
  li.start->key = val;
  li.start->nxt = oldSt;
}
```

```
/** Einfuegen von val am Listenanfang */
public static void
       insBeg(Liste li, int val) {
  Node oldSt = li.start;
  li.start = new Node();
  li.start.key = val;
  li.start.nxt = oldSt;
  }
```

[6] In spitzen Klammern geben wir die Elemente einer Liste an. $<>$ bezeichnet die leere Liste, $< e_1, e_2 >$ die Liste mit e_1 als erstem Element, gefolgt von e_2.

In C++ muss der Parameter `li` ein Referenzparameter sein, da andernfalls beim Aufruf von `insBeg` eine Kopie vom Typ `Liste` auf dem *Programm-Stack* erstellt wird und dann lediglich das Attribut `start` der Kopie manipuliert wird. In Java dagegen werden von den komplexen Typen immer Verweise und niemals die Werte selbst auf dem Programm-Stack abgelegt. Die Implementierung einer Testfunktion ist in Listing 4.12 als Beispiel wiedergegeben. Dort werden neben `insBeg` noch die Funktionen `create` und `dispose` zum *Generieren einer Liste* bzw. zum *Auflösen einer Liste* verwendet, deren Realisierung in Listing 4.13 gezeigt wird.

C++/Java 4.12: Testfunktion `testInsBeg1` mit Hilfsfunktion `checkRes`

```
/** Ueberpruefung, ob jedes Element in
der Liste li im Feld exp vorkommt. */
bool checkRes(int exp[], const Liste& li,
                         int count) {
  int i = 0; NodeRef start = li.start;
  while (start != NULL){
    if (exp[i++]!=start->key){
      return false;
    }
    start=start->nxt;
  }
  return i == count;
}
bool testInsBeg1(){
  Liste* lis = create();
  insBeg(*lis, 5);
  int expected[] = {5};
  bool res=checkRes(expected, *lis, 1);
  dispose(lis);
  return res;
}
```

(WertRef/ListClass/Test.cpp)

```
/** Ueberpruefung, ob jedes Element in
der Liste li im Feld exp vorkommt. */
public static boolean
  checkRes(int[] exp, Liste li, int count) {
  int i=0; Node start = li.start;
  while (i < count && start!=null){
    if (exp[i++] != start.key) {
      return false;
    }
    start=start.nxt;
  }
  return true;
}
public static boolean testInsBeg1(){
  Liste lis = Liste.create();
  Liste.insBeg(lis, 5);
  int expected[] = {5};
  boolean res=checkRes(expected, lis, 1);
  return res;
}
```

(WertRef/ListClass/Test.java)

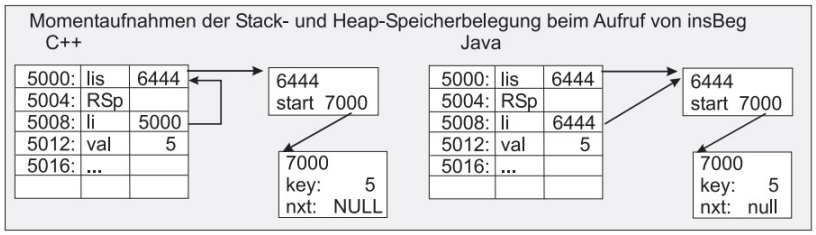

Bild 4.12: Stack- und Heap-Speicherbelegung bei Verwendung des Typs `Liste`

Bild 4.12 zeigt die zugehörige Speicherbelegung in der Funktion `insBeg`. Die unterschiedliche Realisierung der Indirektion einmal über den *Programm-Stack* und einmal über die kopierte Referenz, d.h. zwei gleiche Verweise auf den *Programm-Heap*, ist deutlich zu erkennen.

Wie bereits angedeutet, könnten wir in C++ auch auf den Hilfstyp `Liste` verzichten, wie die folgende Implementierung von `insBeg` zeigt:

C++/Java 4.13: Funktionen `create` und `dispose`

```
/** Erzeugung einer Liste auf
dem Heap und Initialisierung. */
Liste* create() {
  Liste* li = new Liste;
  li->start = NULL;
  return li;
}
/** Freigabe aller Knoten der Liste sowie
der Liste selbst, auf die li zeigt. */
void dispose(Liste* li) {
  NodeRef lauf = li->start;
  while (lauf!=NULL) {
    NodeRef hlp = lauf->nxt;
// Freigabe des Listenknotens
    delete lauf;
    lauf = hlp;
  }
// Freigabe der Liste selbst
  delete li;
}
```

(WertRef/ListClass/Liste.cpp)

```
public static Liste create(){
  Liste li = new Liste();
  li.start = null;
  return li;
  }

// dispose in Java nicht erforderlich
```

(WertRef/ListClass/Liste.java)

```
void insBeg(NodeRef& st, int val) {
  NodeRef oldSt = st; st = new Node;
  st->key = val; st->nxt = oldSt; }
```

Wir erkennen, dass der Parameter `st` als Referenz an die Funktion `insBeg` übergeben wird, d.h. er wird als Ein- **und** Ausgangsparameter verwendet, denn der Listenbeginn ändert sich, wenn am Anfang ein Element eingefügt wird. `st` wird somit in jedem Fall geändert. Die Implementierung des Tests zeigt der folgende Code:

```
bool testInsBeg1(){
  NodeRef start=NULL;
  insBeg(start, 5);
  int expected[] = {5};
  return checkRes(expected, start, 1);
}
```

Bild 4.13 veranschaulicht nochmals die Speicherbelegung auf dem *Programm-Stack* und auf dem *Programm-Heap* beim Eintritt in die Funktion `insBeg` und am Ende.

Ausblick auf andere dynamische Datenstrukturen

Das Spektrum der dynamischen Datenstrukturen ist breit, sie werden vielfach zur Implementierung von *Containern* verwendet, die wesentlicher Bestandteil der zu den Sprachen C++ und Java gehörenden Bibliotheken sind. Wichtige Beispiele für weitere dynamische Datenstrukturen, die z.B. in [30] ausführlich behandelt werden, sind:

- *Doppelt verkettete Listen*: Die im vorangehenden Beispiel behandelten einfach verketteten Listen können zu *doppelt verketteten Listen* erweitert werden, die sich dann leicht in beide Richtungen – vorwärts und rückwärts – durchlaufen lassen.

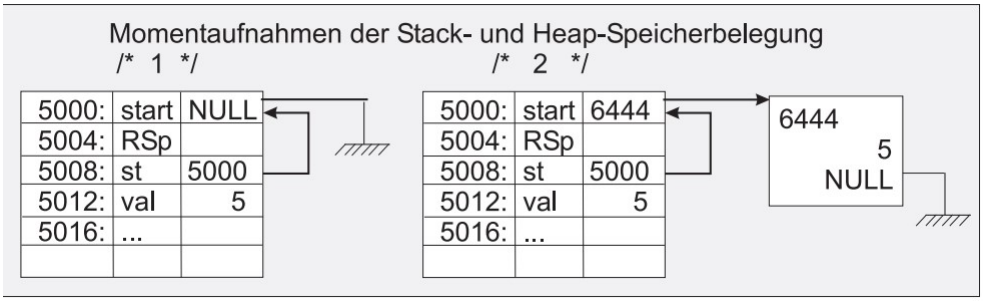

Bild 4.13: Stack- und Heap-Speicherbelegung beim Einfügen am Listenanfang

- *Sortierte Listen*: Listen eignen sich als *sortierte Listen* auch gut zum *Ordnen* von Datenbeständen, solange die Anzahl der Elemente nicht zu groß ist.
- *Bäume*: Zum Ordnen und Suchen in großen Datenbeständen sind z.B. *binäre Suchbäume*, sog. *AVL-Bäume* und *Vielweg-Bäume (B-Bäume)*, effizienter, wobei jede dieser Strukturen ihr spezielles Anwendungsspektrum hat.

4.5.3 Ein dynamischer Vektor

Es gibt viele dynamische Datenstrukturen, die sehr gebräuchlich und nützlich sind. Obwohl ein tiefes Verständnis, d.h. die Fähigkeit, diese Datenstrukturen selbst implementieren zu können, unerlässlich ist, kann der Entwickler doch auf eine Vielzahl von bereits existierenden Datenstrukturen zurückgreifen. Prominenteste Vertreter solcher Bibliotheken sind in C++ die sog. Standard Template Library (`www.sgi.com/tech/stl`) und das Java Collection Framework (`java.sun.com/docs/books/tutorial/index.html`). Beide sind Bestandteil der jeweiligen Sprache. Es gibt zahlreiche Bücher [30, 39, 57], die diese Bibliotheken vorstellen, wir wollen uns exemplarisch nur eine einzige Datenstruktur herauspicken, die wir in weiten Teilen bereits selbst implementiert haben: ein dynamisches Array.

Es kommt häufig vor, dass die Länge eines Arrays nicht von Anfang an bekannt ist. Auf Seite 89 haben wir gesehen, wie wir selbst (durch ein bisschen Programmierarbeit) die Arraygröße zur Laufzeit anpassen können. Sowohl C++ als auch Java bieten durch die Klasse `vector` bzw. `Vector` hier Unterstützung, wie das Beispiel im Listing 4.14 zeigt. Die Funktion `zerlegeInZiffern` zerlegt eine Zahl in ihre Ziffern und gibt die einzelnen Ziffern anschließend auf dem Bildschirm aus.

In C++ ist der Typ der Elemente (im Bsp. `int`), die im Array abgelegt werden sollen, anzugeben. In Java kann dies entfallen. In Java können gleichzeitig verschiedene Typen in dem Vektor abgelegt werden. Einfache Typen (hier `int`) müssen zuvor mit Hüllklassen in Objekte (hier `Integer`) gewandelt werden; siehe Abschn. 4.1.6. Aber auch in Java kann der Typ der Elemente, die im Vektor abgelegt werden dürfen, zur Compile-Zeit bereits festgelegt werden. Dies ist mit der Anweisung `Vector<X> v =`

C++/Java 4.14: Beispiel für `vector` bzw. `Vector`

```cpp
#include <vector>

void zerlegeInZiffern(int zahl) {

  vector<int> vec; // Vektor mit 0 Elem
  int tst = zahl;
  do {
    vec.push_back(tst % 10);
    tst /= 10;
  } while (tst != 0);

  cout << zahl << " = ";
  for (int i = vec.size(); i>0; --i) {
    cout << vec[i-1] << " ";
  }
}
```

(Funk/Klassen/VectorAnw.cpp)

```java
import java.util.Vector;

class VectorAnw {
static void zerlegeInZiffern(int zahl) {
  Vector vec=new Vector(0);//0 Elemente
  int tst = zahl;
  do {
   vec.addElement(new Integer(tst%10));
   tst /= 10;
  } while (tst != 0);

  System.out.print(zahl + " = ");
  for (int i = vec.size(); i>0; --i) {
   System.out.print(vec.elementAt(i-1)+" ");
  }
}
/* ... */
```

(Funk/Klassen/VectorAnw.java)

`new Vector<X>(0)` möglich. `X` darf dann wiederum kein einfacher Typ (z.B. nicht `int`) sein.

Der Zugriff auf einzelne Elemente des Vektors unterscheidet sich in C++ nicht vom Zugriff auf die normalen Arrays. Dies erfolgt in C++ mit dem Index-Operator `[]`. Diese Syntax steht in Java nicht zur Verfügung, weil in Java Operatoren nicht überladen werden können, für Details verweisen wir z.B. auf [30, Abschn. 8.10]. Statt des Index-Operators `[]` wird in Java `elementAt` verwendet. Sowohl der C++-`vector` als auch der Java-`Vector` stellen mit der Methode `size` eine Möglichkeit zur Ermittlung der aktuellen Anzahl der Elemente im Vektor zur Verfügung.

Die Syntax der Anweisung `vec.push_back(tst%10)` entspricht dem Aufruf einer Funktion über einen Funktionszeiger. Wir können uns damit die Methoden als Attribute vom Typ Funktionszeiger in der Struktur `vector` bzw. `Vector` vorstellen; siehe Abschn. 4.4. In Kap. 6 werden wir die Methoden noch im Detail vorstellen.

Einige Methoden für C++- und Java- Vektor-Objekte
`vector<T> vec;` bzw. `Vector vec = new Vector();`

- **C++:** `vec.push_back(elem);` **Java:** `vec.addElement(elem);`
  Fügt das übergebene Element `elem` *hinten am Ende des Vektors an.*
- **C++:** `vec[pos] = elem;` **Java:** `vec.setElementAt(elem, pos);`
  Fügt das Element `elem` *an der Position* `pos` *ein. Der Vektor muss ausreichend groß sein, d.h. an der angegebenen Position muss bereits ein Element vorhanden sein, das überschrieben wird.*
- **Java:** `vec.insertElementAt(elem, pos);`
  Fügt das übergebene Objekt an der Position `pos` *ein. Befindet sich an der angegebenen Position bereits ein Element, werden alle Elemente ab der bisherigen Position* `pos` *um eine Position verschoben und erhalten somit einen um eins verschobenen Index. Eine vergleichbare Methode stellt der C++-vector nicht zur Verfügung.*

⊗ **C++:** `elem = vec.at(pos);` **Java:** `elem = vec.elementAt(pos);`
Liefert das Element an der Position `pos` *zurück. In C++ ist auch der Zugriff mit dem Operator [] (siehe oben) möglich. Im Unterschied zu* `at` *findet dann aber keine Prüfung statt, ob der Index im erlaubten Bereich zwischen* `0` *und* `size() - 1` *liegt.*

⊗ **C++:** `n = vec.size();` **Java:** `n = vec.size();`
Liefert die aktuelle Größe, d.h. wie viele Plätze für Elemente vorhanden sind.

Neben den hier beschriebenen Methoden gibt es noch weitere, z.B. zum gezielten Entfernen von Elementen, zum Sortieren usw.; siehe hierzu auch C++-Kompendium [30, Abschn. 11.5.4 ff].

Achtung: In C++ ist der Rückgabetyp der Funktion `size` vom Typ `vector<T>::size_type`, der meist als `unsigned int` implementiert ist. Das kann zu Problemen führen, wie Listing 4.15 zeigt, denn der Wert der Laufvariablen `j` wird durch den vorzeichenlosen Typ nie negativ und somit nie kleiner als `0`.

C++/Java 4.15: Probleme beim C++-Rückgabetyp von `size` möglich

```cpp
vector<int> vec;
for (int i=0; i<5; ++i) {
  vec.push_back(i);
}
vector<int>::size_type j; // unsigned
/* Verhalten der folgenden Zeilen nicht def.,
   da j als unsigned nie negativ wird,
   siehe Hinweis 'Achtung' im Text */
for (j = vec.size()-1; j>=0; --j) {
  cout << vec[j] << " ";
}
```

(Funk/Klassen/VectorAnw.cpp)

```java
Vector vec=new Vector(0);
for (int i = 0; i<5; ++i) {
  vec.addElement(new Integer(i));
  }

/* Verhalten der folgenden Zeilen auch
   definiert, da size() int zurueckliefert. */
for (int j = vec.size()-1; j>-1; --j) {
  System.out.print(
       vec.elementAt(j) + " ");
}
```

(Funk/Klassen/VectorAnw.java)

Anwendung auf die Netzplanung

Im Abschn. 4.5.1 haben wir zur Verbesserung der Speicherplatzeffizienz unserer Netzplanung die Struktur `DynVorgangsArray` eingeführt. Wir hätten auf diese Struktur verzichten können, wenn wir stattdessen die soeben vorgestellten Container `vector` bzw. `Vector` verwendet hätten; siehe Listing 4.16.

C++/Java 4.16: Struktur `Vorgang` mit `vector` bzw. `Vector`

```cpp
struct Vorgang {
  double dauer;
  double fruehanf;
  double spaetend;
  vector<Vorgang*> vorgaenger;
  vector<Vorgang*> nachfolger;
};
```

(netzplanung/v3a-vector/Vorgang.h)

```java
public class Vorgang {
  double dauer;
  double fruehanf;
  double spaetend;
  Vector<Vorgang> vorgaenger;
  Vector<Vorgang> nachfolger;
}
```

(netzplanung/v3a-vector/Vorgang.java)

Listing 4.17 zeigt die Funktion `initVorgang` zur Initialisierung der Attribute des Vorgangs. Die Funktion `freigeben` kann entfallen. Die in der C++-Bibliothek bzw.

in Java vorhandenen Containertypen sorgen selbst für ein Aufräumen des Speichers. Wir werden in Abschn. 6.2.2 sehen, wie das jeweils in den beiden Sprachen realisiert ist. Die Funktion zum Einfügen von Nachfolgern eines Vorgangs `setzeNachf` lässt sich ebenfalls sehr einfach realisieren.

C++/Java 4.17: Funktion `initVorgang`

```
/** frühester Anfang und spätestes Ende des
Vorgangs werden mit 0 belegt und die Container
werden geleert (hier nicht erforderlich */
void initVorgang(Vorgang* v) {
  v->vorgaenger.clear();
  v->nachfolger.clear();
  v->fruehanf = 0;
  v->spaetend = 0;
  v->dauer = 0;
}

/** Vorgang vv wird als Vorgänger beim
Vorgang vn eingetragen und umgekehrt. */
void setzeNachf(Vorgang* vv, Vorgang* vn) {
  vv->vorgaenger.push_back(vn);
  vn->nachfolger.push_back(vv);
}
```

(netzplanung/v3a-vector/Vorgang.cpp)

```
/** frühester Anfang und spätestes Ende
    des Vorgangs werden mit 0 belegt.
    und zwei leere Container angelegt */
public void initVorgang(Vorgang v) {
  v.vorgaenger = new Vector<Vorgang>(0);
  v.nachfolger = new Vector<Vorgang>(0);
  v.fruehanf = 0;
  v.spaetend = 0;
  v.dauer = 0;
}
/** Vorgang vv wird als Vorgänger beim
Vorgang vn eingetragen und umgekehrt. */
public void setzeNachf
        (Vorgang vv, Vorgang vn) {
  vv.vorgaenger.addElement(vn);
  vn.nachfolger.addElement(vv);
}
```

(netzplanung/v3a-vector/Vorgang.java.sht)

Entsprechend können wir auch in der Klasse `Netz` den Container `vector` bzw. `Vector` verwenden, wie Listing 4.18 zeigt.

C++/Java 4.18: Struktur `Netz` mit `vector` bzw. `Vector`

```
struct Netz {
  double startzeit, endzeit;
  vector<Vorgang> vorg; // Knoten
};
```

(netzplanung/v3a-vector/Netz.h)

```
public class Netz {
  double startzeit,endzeit;
  Vector<Vorgang> vorg; // Knoten
}
```

(netzplanung/v3a-vector/Netz.java)

Für weitere Details verweisen wir auf die vollständige Implementierung der Lösung auf unserer Homepage im Verzeichnis *netzplanung/v3a-vector*. **www**

Übungen

Übung 4.5: Erweitern Sie die Funktionalität der Liste. Implementieren Sie entsprechend `insBeg` die Funktion `insEnd` zum Einfügen eines Elements am Listenende. Implementieren Sie entsprechend `testInsBeg1` die noch fehlenden Tests `testInsBeg2`, `testInsEnd1` und `testInsEnd2`, sodass der folgende Code im Listing 4.19 lauffähig ist.

C++/Java 4.19: Hauptfunktion `main` zur Ausführung der Listentests

```cpp
#include <iostream>
using namespace std;
#include "Test.h"
#include "Liste.h"

int main(){
   if (true == testInsBeg1() &&
     true == testInsBeg2() &&
     true == testInsEnd1() &&
     true == testInsEnd2() ) {
    cout << "Alle Tests erfolgreich.\n";
   }
   else {
    cout << "Tests gescheitert.\n";
   }
}
```

(WertRef/ListClass/main.cxx)

```java
public class ListApp{

static public void main(String[] args){
   if (true == Test.testInsBeg1() &&
     true == Test.testInsBeg2() &&
     true == Test.testInsEnd1() &&
     true == Test.testInsEnd2() ) {
     System.out.println(
          "Alle Tests erfolgreich.");
   }
   else {
     System.out.println(
          "Tests gescheitert.");
   }
}
}
```

(WertRef/ListClass/ListApp.java)

4.6 Zusammenfassung

In diesem Kapitel geht es um die Verwaltung von Daten und um den Zugriff auf Daten im Arbeitsspeicher. In sicherheitskritischen C++-Anwendungen wird häufig durch firmen- oder projektspezifische Entwurfs-Regeln (Design Rules) festgelegt, dass auf die Nutzung von Objekten auf dem Heap-Speicher ganz zu verzichten ist. Zumindest in C++ ist damit aber noch keine Entscheidung gefallen, ob nun ein Wert oder eine Referenz verwendet wird. Referenzen können auch auf Variablen (Objekte) auf dem Programm-Stack verweisen. Referenzen auf ein Objekt (Variable) sind immer dann sinnvoll, wenn es mehrere Nutzer von diesem Objekt gibt.

Dem Java-Entwickler sind in dieser Beziehung bereits viele Entscheidungen abgenommen, das Konzept ist klar und einfach:

- Elementare Daten werden auf dem Programm-Stack als Werte verwaltet;
- selbst definierte Daten (Objekte) werden per Referenz auf dem Programm-Heap verwaltet.

In C++ hat der Programmierer hingegen einen großen Entscheidungsspielraum:

- Daten, ob elementar oder selbst definiert, können generell entweder im Programm-Stack oder im Programm-Heap oder auch global vereinbart werden.
- Der Zugriff auf Daten im Programm-Heap erfolgt per Zeiger oder per Referenz (d.h. indirekt).
- Der Zugriff auf globale Daten und auf Daten im Programm-Stack erfolgt wahlweise per Wert (d.h. direkt) oder per Zeiger oder Referenz (d.h. indirekt).

Zwischen der Semantik von Referenzen in C++ und in Java besteht ein wesentlicher Unterschied: In C++ sind Referenzen[7] fest an ein Objekt gebunden, in Java dagegen

[7] Die C++-Referenzen im engeren Sinne, nicht die Zeiger, die im weiteren Sinne auch Referenzen sind.

kann eine Referenz zur Laufzeit jederzeit an ein anderes Objekt gebunden werden, d.h. sie verhält sich diesbezüglich wie ein Zeiger in C++.

Effiziente Datenstrukturen und Algorithmen sind ohne den Einsatz von Zeigern und Referenzen nicht denkbar, sie stellen somit einen sehr wichtigen Aspekt der Programmierung dar.

Kapitel 5

Fehlersuche und -behandlung

In diesem Kapitel geht es um Maßnahmen, die wir ergreifen können, um die Fehlersuche bei *unerwarteten Fehlern* zu verkürzen (Abschn. 5.1 und 5.2). Dazu besprechen wir sowohl Mittel, die wir bereits *vor* dem Eintreten des Fehlers nutzen können, um ihn rasch zu erkennen, als auch Techniken, die uns *nach* Eintritt eines Fehlers bei einer Diagnose unterstützen.

Aber gleichgültig, wie viel Mühe wir uns auch geben, alle Fehler werden wir nicht verhindern können. Oft müssen wir Fehlfunktionen geradezu einkalkulieren, bspw. einen vollen Hauptspeicher, unerlaubte Funktionsargumente oder nicht auffindbare Dateien. Auf diese Fehler müssen wir *vorbereitet* sein, ihre Behandlung wird uns daher in Abschn. **??** beschäftigen.

5.1 Strategien für die Fehlersuche

Ein realistischer Albtraum: Mehrere 1000 Zeilen Code, die Sie noch nie gesehen haben, die unter bestimmten Umständen abstürzen – und Sie sollen den Fehler finden und korrigieren. Wie gehen Sie vor?

Wissenschaftlich formuliert, läuft die Fehlersuche in fünf Schritten ab [63]:

1. Hypothese erstellen (*wenn lediglich die Vorgangsnamen im Programm verändert werden und damit (nur) der Zeitpunkt der Berechnung der verschiedenen frühesten und spätesten Endzeitpunkte, hat das keinen Einfluss auf das Ergebnis*);
2. Vorhersage (*Die Tests sind nach wie vor durchführbar und es wird* `Alle Tests erfolgreich!` *auf dem Bildschirm ausgegeben*);
3. Experiment durchführen (*Es werden die gezeigten Änderungen im Programm vorgenommen, sodass sich die Umbenennung der Vorgänge ergibt*);
4. Beobachtung (*Die Bildschirmausgabe ist* `Tests fehlgeschlagen!`);

5. Folgerungen für die Hypothese (*In diesem Fall ist die Hypothese zu verwerfen, und damit hängt das Ergebnis von der Reihenfolge ab, in der die Vorgänge präsentiert werden*).

Nun wiederholt sich der Ablauf. Die erste Hypothese war noch einfach zu bilden, nun benötigen wir eine Hypothese für die Ursache des Fehlschlags. Dabei versuchen wir natürlich nicht, den Grund für den Fehler exakt zu erraten, sondern nähern uns dem Problem an, indem wir mit allgemeinen Hypothesen beginnen (*Es hat etwas mit diesem Modul zu tun*) und sie im weiteren Verlauf verfeinern (*Es liegt an der Initialisierung der Datenstruktur X in Funktion f dieses Moduls*). Wenn wir beim Generieren der Hypothesen schlecht sind, durchlaufen wir diese Schritte viele Male.

Drei Dinge machen die Fehlersuche kompliziert:

- **Lokalisierung:** Wo man den Fehler beobachtet, liegt selten die Ursache, da der Fehler meistens nicht sofort zu einem Absturz führt, sondern das Programm läuft eine ganze Weile weiter, bis durch eine scheinbar harmlose Operation ein Fehlverhalten offenbart wird.
- **Code-Komplexität:** Der Umfang und die Komplexität des Codes ist hoch – und der Fehler könnte überall sein.
- **Problem-Komplexität:** Viele Verarbeitungsschritte des Programms können erforderlich sein, ehe der Fehler endlich auftritt. Diese Schritte müssen mit einem Debugger (evtl. auch mehrfach) durchlaufen werden.

Dieser vielschichtigen Problematik begegnet man nur durch ein ganze Reihe von (zum Teil präventiven) Maßnahmen.

Der erste Punkt betrifft die rasche **Lokalisierung von Fehlern**: Wenn ein Fehler aufgetreten ist, wollen wir diesen Umstand so schnell wie möglich erkennen – und nicht aus Nebeneffekten erraten müssen. Je mehr wir über den Fehler wissen, desto leichter fällt das Aufstellen guter Hypothesen für den Grund des Fehlers. An dieser Stelle hilft der intensive Einsatz von Zusicherungen und Invarianten, die wir in Abschn. 5.1.1 besprechen.

Der zweite Punkt spricht die **Komplexität des Codes** an. Wenn wir uns eine binäre Funktion vorstellen, die zehn (binäre) Eingangsvariablen und eine (binäre) Ausgabe besitzt, dann gibt es potenziell $2^{10} = 1024$ verschiedene Testfälle, die zu prüfen sind. Wenn die Realisierung der Funktion aber aus Bausteinen für Konjunktion, Disjunktion und Negation aufgebaut ist, kann jede dieser viel einfacheren Komponenten mit wenig Aufwand getestet werden (je 4 bzw. 2 Testfälle). Funktionieren diese Grundbausteine, kann der Fehler in der Gesamtfunktion *nur noch* in der Verschaltung der Grundfunktionen liegen. Ein Fehler in der Verschaltung äußert sich in der Regel aber nicht nur in einem der 1024 Testfälle, sondern in mehreren. Wenn wir sicherstellen, dass die kleineren, leichter zu testenden Bausteine funktionieren, dann wird ein Fehler in deren Komposition folglich leichter zu entlarven sein. Als Konsequenz sollten nicht nur das Gesamtsystem, sondern bereits die kleineren Einheiten getestet werden (Unit-Tests). Wir werden darauf in Abschn. 5.1.2 zurückkommen.

Der dritte Punkt betrifft die Eingabe selbst, d.h. die **Komplexität der Fehlersituation**. In einer Folge von 100 Kommandos, nach denen ein Absturz erfolgte,

können sicher einige Kommandos weggelassen werden, sodass der Fehler immer noch auftritt. Die Fehlersuche wird vereinfacht, wenn die Eingabe so weit reduziert werden kann, dass jede verbleibende Eingabe zwingend für das Auftreten des Fehlers notwendig ist. Diesen Aspekt beleuchten wir in Abschn. 5.1.3.

5.1.1 Frühwarnungen durch Zusicherungen

In diesem Abschnitt behandeln wir eine sehr einfache Strategie: Bei der Erkennung eines Fehlerzustandes brechen wir das Programm einfach ab. (Solche Programme werden **robuste Programme** genannt.) Diese radikale Vorgehensweise mag nicht besonders elegant erscheinen, wirkt aber sehr effektiv bei der **Lokalisierung** des Fehlers – denn nun erlauben wir es dem fehlerhaften Programmzustand nicht, durch Nebeneffekte noch weitere unerklärliche Phänomene zu generieren, sondern haben gleich die Quelle allen Übels gefunden.

Eine Spezifikation einer Funktion (vgl. Seite **??**) ist nicht nur nützlich, wenn wir jemandem mitteilen möchten, welche Funktionalität wir von ihm erwarten. Viele Fehler enstehen, weil eine Funktion für einen (Sonder-) Fall aufgerufen wird, der ursprünglich gar nicht vorgesehen war. Wenn wir die Vorbedingung aus der Spezifikation vor der Ausführung einer Funktion überprüfen, können wir solche Fälle erkennen.

Weitere Fälle, in denen wir Fehler leicht erkennen können, sind **Invarianten**. Invarianten sind unveränderliche Tatsachen; so handelt es sich bspw. bei `(1<=month)&&(month<=12)` um eine Daten-Invariante, weil eine gültige Monatsangabe immer zwischen 1 und 12 liegt. Schleifen-Invarianten sind Bedingungen, die in allen Schleifendurchläufen wahr bleiben, so darf sich etwa die Anzahl der Listenelemente beim Sortieren einer Liste nicht verändern.

C++/Java 5.1: Fehlererkennung mit Zusicherungen

```cpp
#include <cassert>
int kWoche(int ta, int mo) {
  assert(ta>=1 && ta<=31 &&
        mo>=1 && mo<=12);
  return 1 + ((mo-1) * 30 + ta) / 7;
}
void test1(){
   cout << "KW: "<< kWoche(2,1) << endl;
   cout <<"KW: "<< kWoche(14,44)<< endl;
   cout << "KW: "<< kWoche(2,1) << endl;
}
```

(Funk/Ausnahme/Assert.cpp)

```java
public class Assert{
static public int
            kWoche(int ta, int mo) {
  assert ta>=1 && ta<=31 &&
     mo>=1 && mo<=12 : "Falsches Datum";
  return 1 + ((mo-1) * 30 + ta) / 7;
  }
public static void main(String[] args){
  System.out.println("KW "+kWoche(2, 1));
  System.out.println("KW "+kWoche(14,44));
  System.out.println("KW "+kWoche(2, 1));
  }
}
```

(Funk/Ausnahme/Assert.java)

Man nennt solche Bedingungen, die an bestimmten Stellen im Code wahr sein müssen, auch **Zusicherungen**. In C++ gibt es das Konstrukt `assert(...)`, um solche Zusicherungen in den Code aufzunehmen. (Dazu muss `<cassert>` eingebunden werden.) In Java gehört `assert` zur Sprache. Wird der an `assert` übergebene Boolesche

Ausdruck zur Laufzeit zu `false` ausgewertet, wird das Programm beendet (C++) oder eine Ausnahme geworfen (Java, Ausnahmen behandeln wir im folgenden Abschn. **??**). Das Listing 5.1 demonstriert die Verwendung von `assert`; dabei wird hier vereinfachend angenommen, dass jeder Monat genau 30 Tage hat.

Nachdem mit `assert` erstmals ein Ausdruck mit Wert `false` ermittelt wird, werden einige Lokalisierungs-Informationen an den Anwender übergeben. Das C++-Beispiel könnte z.B. zur folgenden Ausgabe führen (die genaue Fehlermeldung hängt jedoch vom verwendeten C++-Compiler ab):

```
KW: 1
Assertion failed: ta>=1 && ta<=31 && mo>=1 && mo<=12,
file assert.cpp, line 8
```

Das entsprechende Java-Programm meldet z.B.:

```
KW 1
Exception in thread "main" java.lang.AssertionError: Falsches Datum
        at Assert.kWoche(Assert.java:9)
        at Assert.main(Assert.java:14)
```

Die verletzte Bedingung wird in Java nicht ausgegeben, sondern nur der Text nach dem Doppelpunkt. Text und Doppelpunkt dürfen auch weggelassen werden.

Tipp Eine viel verwendete Strategie besteht darin, die Überprüfungen mit `assert` nur im Testbetrieb zu verwenden und sie im Normalbetrieb auszuschalten. Hierzu muss in C++ vor dem Einbinden von `<cassert>` der Makroname `NDEBUG` definiert werden. In Java wird das Programm ohne die Interpreter-Option `-ea` (für *enable assertions*) aufgerufen. In Java sind die Überprüfungen somit standardmäßig abgeschaltet und in C++ eingeschaltet.[1]

5.1.2 Unit-Tests

Dass Tests maßgeblich zur frühen Erkennung von Fehlern beitragen, ist spätestens seit der Lektüre von Kap. **??** bekannt. Mit dem Begriff *Unit-Test* ist ein Test von *Software-Einheiten* oder -Bausteinen gemeint – im Gegensatz zum Test der Software als Ganzes (Komponententest). Während die korrekte Funktionalität des Gesamtsystems auch ein Endnutzer testen könnte, sind die einzelnen Software-Bausteine nicht direkt zugänglich und können nur vom Entwicklerteam selbst getestet werden. Die wichtigsten Punkte, die wir im Zusammenhang mit Tests bereits erwähnt haben, seien hier noch einmal kurz zusammengefasst:

- Damit Tests nach einer Änderung **automatisch** ausgeführt werden können (sog. **Regressionstests**), müssen sie (wie normaler Code) implementiert werden und

[1] Zusicherungen per `assert` gehören erst seit Java 1.4 zur Sprache. Deshalb müssen die Zusicherungen auch für den Compiler zunächst mit `-source 1.4` aktiviert werden. Das Programm wird deshalb mit

```
javac -source 1.4 Assert.java
```

übersetzt. Anschließend kann es ausgeführt werden mit:

```
java -ea Assert
```

– ganz wichtig – selbst entscheiden, ob ein Fehlerfall vorliegt oder nicht (keine *Sichtprüfung* durch den Programmierer mehr).

- **Tests** werden **gegen Schnittstellen** geschrieben, damit sich die Realisierung hinter der Schnittstelle komplett ändern kann, ohne dass unser Testcode neu geschrieben werden muss. Das heißt, dass der Testcode bspw. nur die Header-Datei kennt, nicht aber irgendwelche internen (globalen) Statusvariablen des Moduls, die in der Schnittstelle (Header) nicht aufgeführt sind.
- Es ist eine gute Idee, die Tests zuerst zu schreiben, noch bevor die Schnittstelle endgültig verabschiedet wird. In diesem sog. **test-first**-Ansatz merken wir frühzeitig, wenn der Schnittstelle wichtige Elemente fehlen, um sie überhaupt testen zu können. Unterlaufen beim Design der Schnittstelle solche Fehler, können wir den vorigen Punkt nicht erfüllen – und eine spätere Änderung der Schnittstelle ist immer aufwändiger als eine Korrektur *vor* der Implementierung.

Tipp

Je dichter das Netz von Tests für kleinere Software-Bausteine, desto unwahrscheinlicher wird es, dass Fehler des Gesamtsystems erst beim Endnutzer auffallen. Durch die Popularität des *extreme Programming* (ein Vorgehensmodell zur Software-Entwicklung; siehe folgendes Kap. **??**) sind eine ganze Reihe von Test-Suiten entwickelt worden (*JUnit* für Java, *CppUnit* für C++, u.v.a.m.)[2]. Diese Werkzeuge unterstützen den Entwickler bei wiederkehrenden Routinearbeiten, wenn er viele Tests schreibt und diese periodisch ablaufen lassen will.

Als Beispiel betrachten wir einen *JUnit*-Test für eine Funktion, die die reelle (größere) Lösung einer quadratischen Gleichung $x^2 + p \cdot x + q = 0$ berechnet, sofern es sie gibt. Der Quelltext zu dieser Funktion ist im folgenden Listing links zu sehen. Auf der rechten Seite befindet sich die Test-Suite, die alle Testfälle zusammenfasst. In diesem Fall haben wir nur eine einzige Klasse mit Testfällen (`QuadTest`). Wären es mehrere, wäre die Zeile `suite.addTestSuite(QuadTest.class)`, entsprechend angepasst, mehrfach einzufügen. Alle Funktionen,[3] die mit `public void test...` beginnen, werden dann automatisch bei Ausführung der Test-Suite ausgeführt. Hauptaufgabe eines Tests ist es, Soll- und Ist-Werte zu vergleichen. Diese Vergleiche werden bei *JUnit* durch verschiedene `assertEquals`-Aufrufe ausgeführt. Im Beispiel vergleichen wir im ersten Fall den erwarteten Wert 3.0 mit der Lösung, die uns der Funktionsaufruf `Quad.f(-5,6)` liefert. Weichen die Werte um mehr als die angegebene Genauigkeit von 0.001 (Kompensation etwaiger Rundungsfehler) voneinander ab, so gilt das als Fehlschlag des Tests. *JUnit* protokolliert und zählt die Fehler, stellt sie grafisch dar und ermöglicht anschließend die Navigation durch alle Tests.

[2] *JUnit* (`http://www.junit.org`) ist in der Entwicklungsumgebung Eclipse bereits fest integriert (`http://www.eclipse.org`). Informationen zu *CppUnit* finden sich unter `http://cppunit.sourceforge.net/cppunit-wiki`.

[3] Der aufmerksame Leser wird bemerken, dass hier das Schlüsselwort **static** bei der Funktion `testQuad` fehlt. Wir werden diesen Umstand in Kap. 6 erklären, bei der Benutzung ändert sich für den Moment nichts.

```
public class Quad {
  /**
   * Liefert größere Lösung von x^2+px+q=0,
   * Existenz vorausgesetzt.
   */
  static double f(double p,double q) {
    double w,l;
    p = -p/2.0;
    w = p*p - q;
    assert w>=0:"Lösbarkeit vorausgesetzt";
    l=p+Math.sqrt(w);
    return l;
  }
}
```

```
import junit.framework.Test;
public class AllTests {
  public static Test suite() {
    TestSuite suite = new TestSuite("Tests");
    suite.addTestSuite(QuadTest.class);
    return suite;
  }
}
```

```
import junit.framework.TestCase;
public class QuadTest extends TestCase {
  public void testQuad() {
    assertEquals(3,Quad.f(-5,6),0.001);
    assertEquals(2,Quad.f(0,-4),0.001);
  }
}
```

Durch eine enge Integration in die Entwicklungsumgebung (integrated development environment (IDE)) werden dem Entwickler alle automatisierbaren Schritte abgenommen, sodass die Organisation und Einrichtung der Tests möglichst wenig Overhead bedeutet. Für weiterführende Informationen verweisen wir den Leser auf die zahlreichen Online-Quellen.

5.1.3 Problemvereinfachung

Wenn eine Funktion fehlerhaft ist, dann produziert sie den Fehler selten bei jedem Aufruf, sondern nur in bestimmten Situationen, z.B. bei bestimmten Argumentkonstellationen. Die Bedingungen, unter denen der Fehler auftritt, genau zu kennen, hilft bei der Erkennung der Fehlerursache sehr. Wenn die Eingabe in eine Funktion unterschiedlich lang sein kann (z.B. ein Array oder eine Liste), dann nähert man sich dem Kern des Fehlers umso leichter, je kürzer und damit einfacher die Eingabe ist. Das folgende C++-Programm erfüllt keine sinnvolle Funktion. Es dient uns nur zu Demonstrationszwecken. Die Funktion, die uns den Fehler liefert, ist in diesem Fall der C++-Compiler selbst. Wir erhalten je nach verwendetem Compiler und vorhandener Laufzeitumgebung z.B. folgende Meldung:

```
line 11: Error: multiple types in one declaration
```

Mit einem geübten Auge und etwas Erfahrung wird schnell klar, was den Compiler stört, aber für den Anfänger ist es schwer, aus dieser Meldung die richtigen Schlüsse zu ziehen. Der Compiler gibt Zeile 11 als Fehlerursache an (Markierung ***), aber wir wissen nicht, was wir in dieser Zeile falsch gemacht haben sollen. In diesem einfachen Beispiel ist die Eingabe (der Quelltext) nicht sehr lang, aber in realistischen Situationen wird der Quelltext viel größer sein. Ohne eine Idee, wo der Fehler liegt, können wir versuchen, Teile der Eingabe zu löschen: Verschwindet dabei der Fehler, muss die Ursache im gelöschten Teil zu suchen sein. Wenn wir z.B. Zeile 15 löschen (Markierung **), tritt der Fehler weiterhin auf, an dieser Zeile lag es also nicht.

```
class MyClass {
public:
  void funk2() { double d=44.2; }
}

class YourClass {
  int i;
public:
  void funk1() { int i=44; }
};                      // ***

int main() {
  MyClass *c = new MyClass();
  c->funk2();           // **
  return 0;
}
```

```
class MyClass {


}

class YourClass {



};

int main() {


  return 0;
}
```

Nun ist es mühsam, alle Zeilen einzeln zu entfernen. Stattdessen können wir versuchen, die Eingabe in größeren Teilen zu bearbeiten: Entfernen wir alle Code-Zeilen aus `main`, so bleibt der Fehler dennoch erhalten. Selbst wenn wir alle Anweisungen aus beiden Klassen löschen (im Beispiel rechte Spalte), besteht der Fehler fort. Entfernen wir im nächsten Schritt die Klasse `YourClass` komplett, bleibt der Fehler, entfernen wir hingegen `MyClass`, so verschwindet er. Also musste der Rest von `MyClass` die Ursache für den Fehler enthalten. Bei einer derart reduzierten Klasse ist es nun einfach, den Grund zu finden: Das abschließende Semikolon wurde vergessen.

Diese Strategie beim Umgang mit dem Compiler kann man ebenso bei der Vereinfachung der Eingabe durchführen, die eine Funktion zum Absturz oder zu einer Fehlfunktion bringt (z.B. Sortierfunktion sortiert Array nicht richtig). Durch Reduktion der Eingabe und Beobachtung, unter welchen Umständen die Funktion weiterhin abstürzt, kann man eine minimale Eingabe konstruieren, die den Fehler noch produziert. Für diese Art des **delta debugging** (Analyse der Eingabe-Unterschiede bei verschiedenen Aufrufen) gibt es ebenfalls Werkzeug-Unterstützung, wir verweisen den Leser hierzu auf [63].

5.2 Ablaufverfolgung durch Logging

Ist der Fehler durch Tests erkannt oder durch Zusicherungen lokalisiert worden, müssen die genauen Ursachen herausgefunden werden. Den Prozess der Fehlersuche (Debugging) unterstützen moderne Entwicklungsumgebungen (IDE) meist sehr gut durch interaktive Debugger. Wird auf verschiedenen Plattformen gleichzeitig gearbeitet, stehen sie aber nicht immer konsistent zur Verfügung. Manchmal müssen auch mehrere Informationen zusammengesucht werden, was sich in einer Debugger-Sitzung dann langwierig gestalten kann. Sehr häufig wird man Ausgabeanweisungen in sein Programm einfügen, um überhaupt erst einmal herauszufinden, in welcher Funktion das Programm abgestürzt ist bzw. in welcher Funktion das Programm sich zum ersten Mal nicht entsprechend den Erwartungen verhalten hat.

Der Code sieht anschließend vielleicht wie folgt aus:

```
void funk1() {
  cout << "funk1 betreten";
  int i=44;
  /*... */
  cout << "i ist " << i;
  funk2();
  /*... */
  cout << "funk1 verlassen";
  }
```

```
public static void funk2() {
  System.out.println("funk2 betreten");
  double d=44.2;
  /*... */
  System.out.println("d ist "+d);

  /*... */
  System.out.println("funk2 verlassen");
  }
```

Nun wird man die Software erneut übersetzen, vielleicht noch weitere Ausgabe-Kommandos hinzufügen usw., um das Problem immer weiter einzukreisen. Nachdem der Fehler behoben ist, wird man den Code wieder bereinigen, indem die Debug-Ausgaben entfernt werden. Später – bei der Suche nach der Ursache eines anderen Problems – werden die gleichen Debug-Ausgaben vielleicht abermals eingefügt und wieder gelöscht. Um das wiederholte Löschen und Einfügen etwas praktikabler zu gestalten, existieren Software-Bibliotheken.

Mit den hier vorgestellten Bibliotheken können die Debug-Ausgaben im Code verbleiben. Sie können ohne erneute Übersetzung durch eine Konfigurationsdatei aus- und wieder eingeschaltet werden. Beim Programmstart wird aus einer ASCII-Datei ausgelesen, welche Ausgaben in der Log-Ausgabe enthalten sein sollen. *LogTrace* (für C++, auf den Webseiten zum Buch) arbeitet mit einer so genannten Id-Datei: Jede Ausgabe bekommt einen String (als Id) zugeordnet. Es erfolgen nur Ausgaben, deren Ids in der Konfigurationsdatei vorkommen. Selten ist es sinnvoll, für jede Ausgabe eine neue Id zu vergeben. Praktischer ist es, die Ausgaben bspw. klassenweise ein- und abschalten zu können. Mit *Log4J* (für Java, siehe `www.log4j.org`, seit Java 1.4 auch in leicht veränderter Form im Sprachumfang von Java vorhanden) legt man typischerweise für jede Klasse eine `Logger`-Variable an, über die dann die Ausgaben dieser Klasse erfolgen. Wieder regelt eine Konfigurationsdatei (property-Datei), für welche Klassen eine Ausgabe erfolgen soll.

www

Tipp

Ein Beispiel soll dies verdeutlichen. Der obige Quellcode (je eine Funktion mit *LogTrace* und *Log4J*) sieht dann wie folgt aus:

```
void funk1() {
  LOG_FUNCTION("Nsp", "funk1", "");
  int i=44;
  /*... */
  TRACE("i ist " << i );
  funk2();
  /*... */
  }
```

```
public static void funk2() {
  logger.entering("MyClass","funk2");
  double d=44.2;
  /*... */
  logger.info("d ist "+d);
  /*... */
  logger.exiting("MyClass","funk2");
  }
```

Wenn die Konfigurationsdatei den String `Nsp` oder `funkt1` bzw. `MyClass` enthält, so enthält die Log-Datei beim Betreten der Funktion `funk1` bzw. `funk2` eine Zeile, die den Eintritt in die Funktion und eine Zeile, die das Verlassen der Funktion markiert. *LogTrace* liefert die letztere Ausgabe automatisch, unabhängig davon, wie viele `return`-Anweisungen die Funktion enthält bzw. ob die Funktion durch eine Ausnahme abgebrochen wird. Wenn somit die Funktion irgendwie verlassen wird,

so kann man sicher sein, dass dies auch in der Log-Datei vermerkt ist. In diesem Abschnitt stellen wir nur die Funktionalität des `LOG_FUNCTION`-Kommandos vor. Im Abschn. 6.2.5 werden wir im Zusammenhang mit Konstruktoren und Destruktoren dann lernen, wie man `LOG_FUNCTION` in C++ selbst realisieren könnte. Spätestens dort wird auch klar werden, warum *Log4J* den Service der automatischen Protokollierung des Verlassens der Funktion nicht bieten kann; hier muss die Meldung durch einen `exiting`-Aufruf explizit erzeugt werden.[4]

Das *LogTrace*-Kommando `TRACE` entspricht im Wesentlichen der Ausgabe über z.B. `cout`. Dem gegenüber steht der Aufruf von `logger.info(...)` bei *Log4J*. `TRACE` erzeugt allerdings nur eine Ausgabe, wenn beim letzten Aufruf von `LOG_FUNCTION`[5] eine Ausgabe erfolgte, d.h. die entsprechenden Ids in der Id-Datei gesetzt sind.

Bei *Log4J* wird die Ausgabe nicht durch das umschließende `entering`/`exiting`-Paar bestimmt, sondern durch die Variable `logger`, die wie folgt angelegt werden kann:

```
protected static Logger logger = Logger.getLogger("MyClass");
```

Das Argument der `getLogger(..)`-Funktion entspricht einer Id bei *LogTrace*.

Enthielte die Konfigurationsdatei somit z.B. den String `Nsp` und `MyClass`, so wären alle Debug-Ausgaben aktiviert und der Inhalt der Trace-Datei folgender:

```
> func "Nsp::funk1()"
  i ist 44
  > func "Nsp::funk2()"
    d ist 44.2
  < func "Nsp::funk2()"
< func "Nsp::funk1()"
```

```
<YourClass.funk1>
  i ist 44
  <MyClass.funk2>
    d ist 44.2
  </MyClass.funk2>
</YourClass.funk1>
```

Würde die Steuerdatei nur den String `MyClass` enthalten, wären die Ausgaben in Funktion `funk1` entsprechend deaktiviert und der Inhalt der Trace-Datei weniger umfangreich:

```
> func "Nsp::funk2()"
  d ist 44.2
< func "Nsp::funk2()"
```

```
<MyClass.funk2>
  d ist 44.2
</MyClass.funk2>
```

Genau wie man Einrückungen zur Strukturierung eines Programms und zur Verbesserung der Lesbarkeit einsetzt, strukturiert auch *LogTrace* seine Ausgaben durch Einrückungen. Jedes `LOG_FUNCTION`-Kommando führt zu einer weiteren Einrückung um drei Leerzeichen. Beim Verlassen des Gültigkeitsbereichs des `LOG_FUNCTION`-Kommandos (Verlassen der Funktion) wird entsprechend wieder ausgerückt. *Log4J* rückt nicht entsprechend der Aufruftiefe ein, kann aber um dieses Feature ergänzt werden. Ein Beispiel finden Sie auf der Webseite zum Buch (`bsp/Debug/LogDemo.java`). **www**

Um die Funktionalität der *LogTrace*-Bibliothek nutzen zu können, muss die Header- **C++**

[4] Die Erzeugung dieser Ein- und Austritts-Meldungen kann durch aspektorientierte Programmierung, die uns in diesem Buch aber nicht weiter beschäftigen soll, fast völlig automatisch generiert werden. Java-Entwicklern sei [8] empfohlen.

[5] bzw. `LOG_METHOD`, `LOG_CONSTRUCTOR`, `LOG_DESTRUCTOR` oder `LOG_BLOCK`

Datei `logtrace.h` eingebunden und das Programm mit dem Makro `LOGTRACE_DEBUG` übersetzt werden. [6] Wird ohne Makro `LOGTRACE_DEBUG` übersetzt, werden alle *LogTrace*-Makros zu leeren Anweisungen und erzeugen damit (fast) keinen Laufzeit-Overhead mehr, d.h. das Programm wird so übersetzt, als ob die Ausgaben für *LogTrace* nicht existent wären. *LogTrace* ist im Namensraum `logtrace` vereinbart. Daher muss jede Datei, die die *LogTrace*-Kommandos verwenden will, noch die Anweisung `using namespace logtrace;` enthalten (vgl. auch folgendes Listing). *LogTrace* muss einmalig zu Beginn des Programms durch das Kommando `LOGTRACE_INIT` initialisiert werden.

```
#include "logtrace.h"
using namespace logtrace;

int main() {
  // Angabe Log-Datei und Id-Datei
  LOGTRACE_INIT("netz.log","netz.ids");
  LOG_FUNCTION("Nsp", "main", "");
  /*... */
  return 0;
  }
```

```
import java.util.logging.Logger;

class MyClass {
 protected static Logger logger =
  Logger.getLogger(MyClass.class.getName());

 public static void funk1() {
   logger.entering("YourClass","funk1");
   /*... */
   logger.exiting("YourClass","funk1");
 }
}
```

Java Ein *Log4J*-ähnlicher Logging-Mechanismus ist in Java fest eingebaut, sodass keine weitere Bibliothek erforderlich ist. Allenfalls müssen die entsprechenden Klassen aus dem Paket `java.util.logging` importiert werden. In welche Log-Datei die Ausgabe erfolgen und wie detailliert Log-Meldungen ausgegeben werden sollen, steht in einer sog. Property-Datei, z.B. `log.properties`. Eine Zeile `MyClass.level=WARNING` bedeutet, dass der mit `MyClass` initialisierte Logger nur Meldungen ausgeben soll, deren Dringlichkeit wenigstens die Stufe „Warnung“ erreicht. Weitere Stufen, die *Log4J* unterscheidet, sind bspw. Info, Error und Fatal. Die Konfigurationsdatei muss zur Ausführung des Programms durch eine Kommandozeilenoption `-Djava.util.logging.config.file=log.properties` bekanntgegeben werden. Es empfiehlt sich, mit dem Beispiel (`bsp/Debug/LogDemo.java`) ein wenig zu experimentieren.

www

5.3 Zusammenfassung

Fehler sind schnell passiert, aber oft nur in einem langwierigen Prozess zu entfernen. Es lohnt sich, etwas Zeit im Vorfeld zu investieren, um die künftig noch zu erwartenden Fehler schneller beseitigen zu können. In diesem Kapitel haben wir als stärkste *Waffen* gegen den Fehlerteufel Zusicherungen, Tests, Logging und Eingabe-Vereinfachung identifiziert. Ihre intensive Nutzung ist in größeren Projekten unerlässlich.

[6] Der gesamte Code muss mit den Präprozessor-Anweisungen `#define HAVE_CONFIG_H` und `#define LOGTRACE_DEBUG` übersetzt werden. Außerdem muss sich das Verzeichnis `liblogtrace/logtrace` im Include-Pfad befinden. Bei Verwendung der gcc-Compiler-Serie ist z.B. die folgende Anweisung zur Übersetzung erforderlich:

```
g++ -DHAVE_CONFIG_H -DLOGTRACE_DEBUG -I../liblogtrace/logtrace
```

Nicht immer ist die Tatsache, dass die Ausführung eines Programms nicht korrekt beendet werden konnte, auf einen Programmierfehler zurückzuführen. Einige Ausnahme-Situationen lassen sich vorhersehen und erfordern eine Berücksichtigung im Programmfluss – so soll der Nutzer, wenn sich eine Datei nicht öffnen lässt, das Programm nicht neu starten müssen, sondern seinen Tippfehler korrigieren dürfen. Wir haben Exceptions als besten Ansatz kennengelernt, die Meldung einer Ausnahmesituation von ihrer Behandlung zu trennen, ohne viel administrativen Aufwand für den Transport des Fehlerreports vom Enstehungs- zum Behandlungsort treiben zu müssen.

Abschließend ist folgende Bemerkung angebracht: Debugging ist zeitaufwändig und frustrierend, gerade für Anfänger. Aber es hat auch einen hohen Lerneffekt! Wer seine Programme selbst entwanzen kann, gewinnt Erfahrungen, wie sich typische Fehler äußern und wo sie gerne auftreten. Selbst wenn wir so nicht verhindern, dass uns derselbe Fehler ein weiteres Mal unterläuft, so finden wir ihn beim zweiten Mal doch schneller. Nicht unterschätzen sollte man auch das persönliche Gespräch: Wenn der Entwickler die Problematik einem Kollegen *erklärt*, lösen sich erstaunlich häufig Denkblockaden, ohne dass überhaupt eine Reaktion des Kollegen nötig gewesen wäre.

Kapitel 6

Abstrakte Datentypen: Einheit von Daten und Funktionalität

In diesem Kapitel werden wir die fundamentale Bedeutung von Schnittstellen kennenlernen und untersuchen, wie sie sauber umgesetzt werden können. Die Lösung besteht in so genannten *abstrakten Datentypen*, die wir zunächst *von Hand* in einer prozeduralen Sprache (C) umsetzen. Mit diesem Verständnis ist die Übertragung auf die Konzepte objektorientierter Sprachen nur noch ein kleiner Schritt.

6.1 Die Bedeutung von Schnittstellen

6.1.1 Kapselung von Komplexität

Bei komplexen Software-Systemen sind unzählige große und kleine Entscheidungen zu treffen, die direkt oder indirekt Einfluss auf den Erfolg haben. Man sagt, dass selbst ein geschulter Mensch niemals mehr als fünf bis sieben Dinge gleichzeitig verfolgen und berücksichtigen kann, für mehr sind die Wahrnehmung und das menschliche Gehirn nicht ausgelegt. Wenn die menschlichen Fähigkeiten derart beschränkt sind, ist der Komplexität von Systemen eine Art natürliche Grenze gesetzt, könnte man meinen. Dass dem nicht so ist, erleben wir im Alltag immer wieder. Ein Auto ist (heute mehr denn je) ein komplexes System – aber trotzdem benutzen wir es tagtäglich, ohne uns Gedanken über seine Komplexität zu machen. Mehr noch: Wir können in fast jeden beliebigen Pkw einsteigen und unabhängig vom Hersteller oder vom Fahrzeugtyp das Auto fahren!

Der Schlüssel dazu liegt offensichtlich in der standardisierten Bedienung: Wir müssen nichts von Verbrennungsmotoren, Schaltgetrieben oder Katalysatoren verstehen, um ein Auto bedienen zu können. Es reichen einige Pedale, Lenkrad und Zündschlüssel. Diese *Schnittstelle* verbirgt die Komplexität vor dem Bediener. Dieses Prinzip funk-

tioniert im Alltag tausendfach (Radio, Fernseher, Funkuhr, Kaffeemaschine, ...) und ist auch der Schlüssel für die Bewältigung der Komplexität großer Software-Systeme.

Wir können also Komplexität reduzieren, indem wir komplexe Sachverhalte hinter einer einfachen Schnittstelle verbergen. Im Beispiel der Netzplanung können wir vor dem Anwender beispielsweise verbergen, dass für eine Planung eine Vorwärts- und eine Rückwärtsberechnung erforderlich sind. Auch die Art und Weise, wie wir den Netzgraphen intern im Speicher halten, spielt für die korrekte Anwendung der Netzplanung keine Rolle.

Durch Schnittstellen reduzieren wir nicht nur die Komplexität für den Bediener (hier: den Kollegen, mit dem wir gemeinsam Software entwickeln), sondern standardisieren auch die Bedienung und erreichen damit noch einen anderen, wichtigen Punkt: Standardisierte Teile sind leichter austauschbar. Wenn Ihre Kaffeemaschine nicht richtig arbeitet (weil sie defekt ist, zu langsam kocht, der Kaffee nicht richtig schmeckt), können Sie sich einfach eine neue Maschine besorgen und die alte ersetzen. Der Austausch vollzieht sich dank einheitlicher Schnittstelle reibungslos.

Wenn wir in der Praxis Software entwickeln, dann wählen wir in dem Moment, in dem wir einen Datentyp benötigen, diejenige Implementierung aus, die uns zum Zeitpunkt der Auswahl am geeignetsten scheint. Wir können bei einem Start-up-Unternehmen etwa davon ausgehen, dass wir niemals mehr als 100 Kunden speichern müssen und entscheiden uns daher für eine Implementierung einer Kundenverwaltung über ein Array. Später kann sich die Annahme aber als falsch herausstellen. Was ist nun zu tun? Wenn sich der Zugriff auf das Array im ganzen Programm widerspiegelt (wie in der Netzplanung in Kap. **??**), dann zieht der Austausch einen großen Änderungsaufwand nach sich. Wenn wir aber eine *saubere Schnittstelle* für die Kundenverwaltung erstellt haben, die von der Art der Speicherung abstrahiert, dann schreiben wir einfach eine neue Kundenverwaltung, die eine beliebige Anzahl von Kunden verwalten kann, halten die Schnittstelle ein und können die Software leicht austauschen.

Der Haken bei der Sache ist, dass der erste Entwurf einer Schnittstelle vielleicht nicht der beste ist, was wir aber erst beim Austausch merken. Die Bedienung eines Autos ist heute zwar weitestgehend standardisiert, war im ersten *Ford T* aber sicher noch anders als heutzutage. Muss eine Schnittstelle geändert werden, verlieren wir alle Vorteile: alle Kunden müssen umlernen bzw. alle Stellen im Code, wo wir die Schnittstelle benutzen, müssen mühsam von Hand geändert werden – bei Software ein fehleranfälliger Prozess.

> **Schnittstellen** reduzieren Komplexität und erleichtern den reibungslosen Austausch von Software-Bausteinen (wobei der typische Fall darin besteht, einen fehlerhaften oder ineffizienten gegen einen besseren Software-Baustein auszutauschen). Ihre Definition will aber wohl überlegt sein, nachträgliche Änderungen an Schnittstellen schmälern die Kosten/Nutzenbilanz erheblich.

Wie aber sieht eine gute Schnittstelle aus?

6.1.2 Datenkapselung: Abstrakte Datentypen

Potenziell ist alles, was öffentlich zugänglich ist, Bestandteil der Schnittstelle (vielleicht decken deshalb immer mehr Autohersteller den Motorraum mit einem großen Plastikdeckel ab). Betrachten wir einmal eine Alternativ-Implementierung der Netzplanung: Jedes `Vorgang`-Objekt ist selbst dafür verantwortlich, welche Vorgänger und Nachfolger er hat (also Adjazenzlisten statt Adjazenzmatrix). In die Schnittstelle gehören dann Funktionen zum Erzeugen eines `Vorgang`-Objektes, zum Hinzufügen von Nachfolgern und zur Abfrage der Vorgangsdauer.

Wir erklären diesen Teil zur *öffentlichen* Schnittstelle, indem wir ihn in die Header-Datei schreiben. Während es legitim ist, in einer Header-Datei Datenstrukturen zu definieren, so ist doch unmittelbar klar, dass wir damit ein Implementierungsdetail der Umsetzung verraten. Betrachten wir den Inhalt der Header-Datei als verbindlichen *Vertrag*, an den wir uns langfristig halten wollen, so liefern wir somit eine Zusage, uns an diese Datenstrukturen dauerhaft zu binden.

```
typedef struct {
  Vorgang* nachfolger[];
  int anzNachfolger;
  double dauer;
} Vorgang;

Vorgang* erzeugeVorgang(double d);
void addNachfolger(Vorgang*);
Vorgang* getNachfolger(int nr);
...
```

Ist das problematisch? Dieser Einblick in die Implementierung ermöglicht dem Nutzer eine direkte Eingriffsmöglichkeit in unsere Datenstrukturen. Warum sollten wir verhindern, dass die verwendeten Datenstrukturen so offengelegt werden, wie in diesem Beispiel? Wenn unsere Schnittstelle eine bestimmte Funktionalität nicht bietet, so hat die Offenlegung der Datenstrukturen doch den Vorteil, dass der Nutzer diese Funktionalität selbst „nachrüsten“ kann, allerdings nur deshalb, weil er den Aufbau der Datenstrukturen kennt.

Dieser Vorteil wird unter dem Gesichtspunkt der Austauschbarkeit aber eben zu einem Nachteil: Denn nun ist der Datentyp-spezifische Code nicht mehr zentral an einer Stelle gesammelt, sondern über viele Stellen verstreut. Eine später notwendig gewordene Änderung hat nicht nur lokale Auswirkungen auf alle Dateien, die diese Schnittstelle implementieren, sondern wir müssen potenziell *den ganzen Code des Projektes* nach Zugriffen auf die Datenstrukturen durchforsten, den Code nachvollziehen und ggf. notwendige Anpassungen vornehmen. Der Vorteil der einfachen Austauschbarkeit wird somit leichtfertig vergeben.

Nehmen wir für unser Beispiel an, dass nach einiger Zeit zum Vorgang eine Funktionalität `getVorgaenger` hinzukommen soll. Im Gegensatz zur *Änderung von Funktionen* in der Schnittstelle ist die *Hinzunahme von neuen Funktionen* zu einer Schnittstelle als ein unkritischer Schritt anzusehen, weil daraus kein Änderungsbedarf außerhalb des Datentyps entsteht. Wenn wir nun die Funktion `getVorgaenger` imple-

mentieren, könnten wir auf die Idee kommen, den Vorgang selbst auch um eine Vorgängerliste zu erweitern:

```
typedef struct {
  Vorgang* nachfolger[];
  int anzNachfolger;
  Vorgang* vorgaenger[];
  int anzVorgaenger;
  double dauer;
} Vorgang;
```

Das neue Vorgänger-Array können wir trotzdem allein mit Hilfe der Funktion `addNachfolger` aktuell halten (wir gehen im Beispiel vereinfachend davon aus, dass die Arrays immer ausreichend groß sind):

```
void addNachfolger(Vorgang *dieser,Vorgang *nachf) {
  dieser->nachfolger[anzNachfolger]=nachf; ++anzNachfolger;
  nachf->vorgaenger[nachf->anzVorgaenger]=dieser; ++(nachf->anzVorgaenger);
}
```

Was geschieht aber, wenn in der Zwischenzeit jemand – unter Ausnutzung der offengelegten Datenstrukturen – eine Funktion *„lösche alle Nachfolger mit einer Dauer kleiner eine Stunde“* geschrieben hat? Aus der Schnittstelle sind keine Elemente entfernt oder verändert worden, daher wird der Code noch übersetzt, es gibt keine Fehlermeldung. Weil sich aber die *interne Repräsentation* eines Vorgangs geändert hat (wir führen nun zusätzlich eine Vorgängerliste), überführt der externe Code unsere Vorgänge nun in einen inkonsistenten Zustand. Alle Nachfolger mit einer Dauer kleiner als eine Stunde sind zwar aus den Nachfolger-Arrays entfernt, aber in den Vorgänger-Arrays sind sie noch enthalten! Während das Hinzufügen neuer Funktionen (unter Beibehaltung aller alten Funktionen in Syntax und Semantik) in Schnittstellen unproblematisch ist, gilt dies offensichtlich **nicht** für Datenstrukturen. Darum sollte die Repräsentation nicht offengelegt werden, damit kein Anwender überhaupt die Möglichkeit hat, die Repräsentation auszunutzen.

Ein **abstrakter Datentyp** besteht aus einer öffentlichen Schnittstellendefinition (Syntax) und der Spezifikation der Bedeutung der Funktionen (Semantik), alle Implementierungsdetails (insbesondere Datenstrukturen) bleiben verborgen *(Geheimnisprinzip*, engl.: *information hiding)*. Als Konsequenz müssen Änderungen an Daten immer über Funktionen erfolgen (so genannte getter- und setter-Funktionen zum Lesen und Schreiben von Attributen). Damit wird es im Gegensatz zur Bekanntgabe der Datenstrukturen nun auch möglich, den Lese- und Schreibzugriff auf einzelne Daten (etwa durch den Wegfall einer set-Methode) getrennt zu regeln.

Weil nur über Funktionen auf die Daten zugegriffen wird, ist ein abstrakter Datentyp implementierungsneutral und kann auch mehrfach in verschiedener Gestalt (polymorph) umgesetzt werden (siehe Kap. 7). Offenbar kommt der Klärung der Semantik bei abstrakten Datentypen eine besondere Rolle zu: Notfalls konnten wir bisher immer in die Implementierung einer Funktion schauen, um zu verstehen, was

sie macht. Bei abstrakten Datentypen ist es dagegen wichtig, die Semantik unabhängig von einer Implementierung festzulegen, denn eine konkrete Implementierung legt im Allgemeinen viele Details fest, die nicht unbedingt Anforderungen an den Datentyp darstellen. Es muss *ohne eine Referenzimplementierung* möglich sein, über die Korrektheit einer Datentyp-Implementierung zu entscheiden. Dies wird uns in Abschn. 6.2.4 noch näher beschäftigen.

Eigenschaften guter Schnittstellen sind:

- Sie enthalten keine Datenstrukturen.
- Sie sind *schmal*, d.h., sie bestehen aus so wenig Funktionen wie möglich (was sie einfacher durchschaubar macht und Komplexität reduziert).
- Sie enthalten Funktionen mit klarer Bedeutung – zu viele Fälle mit einer Funktion erschlagen zu wollen, ist gefährlich: wird etwas übersehen und eine Änderung ist nötig, so ist das *teurer* als das Hinzufügen einer neuen Funktion.
- Sie sind nicht zu sehr auf eine bestimmte Art der Implementierung ausgerichtet, sondern orientieren sich nur am Bedarf des Schnittstellen-Nutzers. (Beispiel: Wenn in der Schnittstelle ohne zwingenden Grund der wahlfreie Zugriff auf das n-te Element gefordert ist, benachteiligt das z.B. Listen gegenüber Arrays.)

Netzplanung mit abstrakten Datentypen

Wie können wir mit den Mitteln einer prozeduralen Sprache überhaupt abstrakte Datentypen realisieren? Dazu müssen wir in C nichts weiter tun, als die Definition des Datentyps `Vorgang` aus der Header-Datei in die Quellcode-Datei zu verschieben. Damit die in der Header-Datei verbleibenden Funktionsprototypen weiterhin vom Compiler akzeptiert werden, müssen wir eine so genannte Vorwärtsdeklaration des `struct Vorgang` vornehmen, d.h., wir geben bekannt, dass es ein `struct Vorgang` gibt, geben aber nicht vor, wie es aussieht. Listing 6.1 und 6.2 zeigen, was in den Header-Dateien von `Vorgang` und `Netz` sichtbar bleibt.

C/C++ 6.1: Schnittstelle Vorgang, H-Datei (netzplanung/v4-adt/Vorgang.h)

```
typedef struct Vorgangsdaten Vorgang;

Vorgang* erzeugeVorgang(double d);
double getDauer(const Vorgang* v);
double getFruehAnf(const Vorgang* v);
double getSpaetEnd(const Vorgang* v);
void zerstoereVorgang(Vorgang *v);
```

Ein Zugriff des Benutzers auf die Daten unter Umgehung der Zugriffsfunktionen kann mit dieser Lösung verhindert werden. Sogar die Instanziierung eines Vorgangs unter Umgehung eines Aufrufs der Funktion `erzeugeVorgang` ist unterbunden, weil nach dem Einbinden der Header-Datei der Compiler die Größe einer Variablen vom Typ `Vorgang` nicht kennt und den Versuch einer Speicherallokation für `Vorgang` mit einer entsprechenden Fehlermeldung quittiert.

C/C++ 6.2: Schnittstelle `Netz`, Header-Datei (netzplanung/v4-adt/Netz.h)

```
typedef struct Netzdaten Netz;

Netz* erzeugeNetz(double startzeit,double endzeit);
Netz* fuegeHinzu(Netz* n,Vorgang *v);
Netz* setzeNachfolger(Netz *n,const Vorgang *v,const Vorgang *w);
Vorgang* getVorgang(const Netz*,int i);
bool plane(Netz *n);
void zerstoereNetz(Netz *n);
```

Es fällt auf, dass wir in der Schnittstelle keine Funktion vorgesehen haben, die das *Setzen des frühesten Anfangszeitpunktes* eines Vorgangs ermöglicht. Dies ist mit Bedacht geschehen, weil nur die Netzplanung selbst (über die Funktion `plane`) das Setzen dieser Werte vornehmen soll. Dem Anwender soll es nicht möglich sein, die Planungswerte zu *verändern*. Nichtsdestotrotz werden wir, wie in Listing 6.3 gezeigt, in der Source-Datei eine Funktion `setFruehAnf` vorsehen, die für den internen Gebrauch durch Netzplanungs-Funktionen gedacht ist. Diese Funktion ist kein Bestandteil der öffentlichen Schnittstelle, aber Teil des Vertrages zwischen den eng kooperierenden Datentypen `Vorgang` und `Netz`. In der Source-Datei werden die Vorgangs-Funktionen implementiert, und dort wird auch die Definition der Datenstruktur `Vorgang` durch Implementierung von `Vorgangsdaten` nachgeholt:

C/C++ 6.3: Implementierung `Vorgang`, Quellcode-Datei

```
struct Vorgangsdaten {
  double dauer,fruehanf,spaetend;
  int id;
};

/** Neuen Vorgang mit Dauer d anlegen. */
Vorgang* erzeugeVorgang(double d) {
  Vorgang *v = new Vorgang();
  v->dauer = d;
  v->id = -1;
  return v;
}
```

```
/** Einfache Zugriffsfunktionen */
double getDauer(const Vorgang* v) {
  return v->dauer; }
double getFruehAnf(const Vorgang* v) {
  return v->fruehanf; }
void setFruehAnf(Vorgang* v,double fa) {
  v->fruehanf=fa; }
double getSpaetEnd(const Vorgang* v) {
  return v->spaetend; }
void setSpaetEnd(Vorgang* v,double se) {
  v->spaetend=se; }
int getId(const Vorgang* v) {
  return v->id; }
void setId(Vorgang* v, int id) {
  v->id=id; }
void zerstoereVorgang(Vorgang* v) {
  delete v; v=NULL; }
```

Bei der Realisierung des Vorgangs haben wir nicht den obigen Ansatz (jeden Vorgang seine Nachfolger selbst speichern zu lassen) verfolgt, sondern bleiben dichter an der bereits bekannten Realisierung. Allerdings haben wir ein neues Attribut im `Vorgang` eingeführt, das Attribut `id` (vgl. Listing 6.3). Wir haben als eine Eigenschaft guter Schnittstellen hervorgehoben, dass sie nicht viel über die Realisierung selbst verraten. In bisherigen Realisierungen der Netzplanung haben wir die Vorgänger/Nachfolger-Beziehung direkt in der Adjazenzmatrix des Netzgraphen eingetragen. Das könnten wir natürlich beibehalten, aber dazu müsste der Anwender wissen, welche laufende Nummer ein `Vorgang` in der Adjazenzmatrix hat. Damit bürden wir dem Anwender

der Netzplanung zusätzliche (Buchführungs-)Arbeit auf, die Bedienung wird komplexer. Besser ist es, wenn wir dieses Detail ebenfalls verbergen, d.h. mehr Komplexität kapseln.

Dazu benutzen wir die `id` im `Vorgang`: Beim Hinzufügen eines Vorgangs zu einem Netz vergeben wir dem `Vorgang` eine laufende Nummer. Beim *Verschalten* der Nachfolger über die Funktion `setNachfolger` erwarten wir nur Referenzen auf Vorgänge, d.h. wir muten dem Anwender nicht auch noch zu, die korrespondierenden laufenden Nummern zu den Vorgängen zu kennen. In die Adjazenzmatrix tragen wir dann an der richtigen Stelle die Verbindung ein, indem wir die beteiligten Vorgänge nach ihren laufenden Nummern fragen. Das Attribut `id` im `Vorgang` ist nur dem Zusammenspiel zwischen `Vorgang` und `Netz` und einer guten Schnittstelle geschuldet, was in diesem Fall Ausdruck der engen Kopplung beider Datentypen ist. (Eine alternative Lösung wäre die Implementierung über Adjazenzlisten gewesen.)

Listing 6.4 zeigt die Implementierung zu der Schnittstelle in Listing 6.2.

C/C++ 6.4: Implementierung `Netz`, Quellcode-Datei

```
struct Netzdaten {
  double startzeit,endzeit;
  int anzahl;
  Vorgang *vorg[MAX]; // Knoten
  bool nachf[MAX][MAX];// Adjazenzmatrix
  int reihenfolge[MAX];
};

/** Neues Netz mit Start-/Endzeit anlegen */
Netz* erzeugeNetz(double sz,double ez) {
  int i,j;
  Netz *n = new Netz();
  n->startzeit = sz;
  n->endzeit = ez;
  n->anzahl = 0;
  for (i=0;i<MAX;++i) {
    for (j=0;j<MAX;++j) {
      n->nachf[i][j]=false;
    }}
  for (i=0;i<MAX;++i) n->reihenfolge[i]=i;
  return n;
}
/** Knoten #i aus Netz zurueckliefern. */
Vorgang* getVorgang(const Netz* n, int i){
  return n->vorg[i];
}
```

```
/** Knoten v zu Netz netz hinzufügen. Ein
Vorg. darf nur einem Netz zugeordnet sein.*/
Netz* fuegeHinzu(Netz* netz,Vorgang *v) {
  if (getId(v)!=-1)
    cout << "Fehler!";
  setId(v,netz->anzahl);
  netz->vorg[getId(v)]=v;
  netz->anzahl++;
  netz->reihenfolge[getId(v)]=getId(v);
  return netz;
}

/** Angabe einer Vorgaenger-Nachfolger
 -Beziehung im Netz n. */
Netz* setzeNachfolger(Netz *n,
     const Vorgang *v, const Vorgang *w){
  n->nachf[getId(v)][getId(w)]=true;
  return n;
}

/** Loeschen des Netzes. */
void zerstoereNetz(Netz* n) {
  delete n; n=NULL;
  }
```

6.2 Klassen als abstrakte Datentypen

Die Datenkapselung und der Zugriff über Schnittstellenfunktionen ist ein wesentlicher Bestandteil objektorientierter Programmierung. In gewissem Sinne haben wir in den Listings 6.1–6.4 in einer prozeduralen Sprache Code objektorientiert umgesetzt. Moderne objektorientierte Sprachen unterstützen eine Umsetzung in diesem

Sinne durch leicht erweiterte Funktionalität. Eine **Klasse** ist zunächst nichts weiter als ein `struct`, in dem neben den Variablen (nun Attribute genannt) auch gleich die Funktionen der Schnittstelle (nun Methoden genannt) deklariert werden können. Damit wird die inhaltliche Zusammengehörigkeit der Methoden zur Datenstruktur (nunmehr *Klasse*) deutlich gemacht, Daten und Funktionen bilden eine Einheit.

Zunächst leisten objektorientierte Sprachen einen Dienst bei der Tipparbeit. Betrachten wir einmal die Funktionen zur Datenstruktur `Vorgang` in Listing 6.2: Sie haben alle als erstes Argument eine Variable vom Typ `Vorgang*`, das den *aktuellen* `Vorgang` referenziert, auf den sich die Funktionalität bezieht. Dieses erste Argument ist offenbar bei allen Funktionen, die sich auf eine bestimmte Vorgangsinstanz beziehen, notwendig. Daher kann der Compiler das Notieren dieses ersten Arguments selbst übernehmen. Beim Aufruf der Funktion müssen wir die Referenz auf den Vorgang aber natürlich mit angeben, sonst weiß die Funktion nicht, auf welche Datenstruktur sich der Aufruf bezieht. Damit wir nun aber nicht bei Funktionsdeklaration und -aufruf unterschiedliche Signaturen haben (bei Deklaration ohne, bei Aufruf mit erstem `Vorgang*`-Argument), wurde bei objektorientierten Sprachen die Schreibweise für den Aufruf von Methoden gegenüber dem Aufruf von Funktionen geändert: Bei einem gegebenen Zeiger auf einen `Vorgang v` verschieben wir das erste Argument *vor* den Funktionsaufruf. Statt

```
double d = getDauer(v);
```

schreiben wir

```
double d = v.getDauer();
```
oder
```
double d = v->getDauer();
```

Und zwar benutzen wir in Java immer die linke Syntax, in C++ analog zum Zugriff auf `struct`'s bei Wertesemantik die linke, bei Zeigersemantik die rechte Syntax. In allen Fällen wird für den Vorgang `v` (oder den von `v` referenzierten Vorgang) die Methode `getDauer` aufgerufen. (Man beachte die Ähnlichkeit mit dem Aufruf einer Funktion, die als Funktionszeiger `getDauer` in einer Struktur `v` gespeichert ist, vgl. Abschn. 4.4.)

Um in den Methoden trotzdem auf dieses automatisch eingefügte Argument zugreifen zu können, müssen wir dessen Namen kennen. Hier hat sich in beiden Sprachen `this` als Name durchgesetzt. Aus der Anweisung `return v->dauer` der `getDauer`-Funktion wird die Anweisung `return this->dauer` der `getDauer`-Methode.

6.2.1 Sichtbarkeit

Wir haben am Beispiel von `Vorgang` und `Netz` gesehen, dass es drei verschiedene Arten von Funktionen gibt, wenn es um ihre *Sichtbarkeit nach außen* geht. Dafür gibt es beim Arbeiten mit Klassen in C++ und in Java jeweils eine klare sprachliche Unterstützung, wie sie in Listing 6.5 am Beispiel der Klasse `Vorgang` zum Ausdruck kommt.

Zunächst haben wir **öffentliche** Methoden: Das sind alle Methoden, die Bestandteil des (abstrakten) Datentyps sind, über die der Anwender (ohne Kenntnis der Repräsentation) mit dem Datentyp umgehen soll. In einer objektorientierten Sprache

C++/Java 6.5: Eine Vorgangs-Klasse

```cpp
class Vorgang {
 private:
  double dauer,fruehanf,spaetend;
  int id;

 public:
  Vorgang(double d);
  ~Vorgang();
  double getDauer() const;
  double getFruehAnf() const;
  double getSpaetEnd() const;

 private:
  void setFruehAnf(double fa);
  void setSpaetEnd(double se);
  int getId() const;
  void setId(int id);

 friend class Netz;
};
```

(netzplanung/v4-adt-cpp/Vorgang.h)

```java
class Vorgang {
 private double dauer,fruehanf,spaetend;
 private int id;

 public Vorgang(double d) {
  dauer = d; id = -1; }
 public double getDauer() {
  return dauer; }
 public double getFruehAnf() {
    return fruehanf; }
 public double getSpaetEnd() {
    return spaetend; }

 protected void setFruehAnf(double fa) {
    fruehanf=fa; }
 protected void setSpaetEnd(double se) {
    spaetend=se; }
 protected int getId() {return id;}
 protected void setId(int id) {this.id=id;}
};
```

(netzplanung/v4-adt-java/Vorgang.java)

kann man Elemente einer Klasse (Attribute wie Methoden) mit dem Schlüsselwort **public** als öffentlich deklarieren. Ist ein Element öffentlich deklariert, kann jeder darauf zugreifen (d.h. Methoden aufrufen, Attribute lesen oder schreiben).

Wir haben auch gesehen, dass es **nicht-öffentliche** oder **interne** Elemente (Methoden oder Daten) gibt, zum Beispiel die Methoden zur Vorwärts- oder Rückwärtsrechnung beim Netz. Diese Methoden wurden eingeführt, um die Durchführung der Planung gemäß schrittweiser Verfeinerung sauber zu strukturieren, aber von außen sollen sie nicht aufgerufen werden. Sie sind nur für den internen Gebrauch bestimmt, dienen u.a. der besseren Nachvollziehbarkeit des Quelltextes. Solche Elemente (Attribute wie Methoden) werden in objektorientierten Sprachen mit dem Schlüsselwort **private** gekennzeichnet. Wir wissen bereits: Im Sinne von abstrakten Datentypen sollten *immer alle Attribute privat deklariert werden*! **Tipp**

Und dann gab es noch eine dritte Art von Methoden im Listing 6.5, deren Existenz sich dadurch begründet, dass manche Datenstrukturen enger miteinander zusammenarbeiten als andere (sog. starke Kopplung). Die Datentypen `Vorgang` und `Netz` können wir unabhängig voneinander als abstrakte Datentypen auffassen. Das ergibt Sinn, wenn wir in der Lage sein wollen, die interne Repräsentation des einen Datentyps unabhängig vom anderen Datentyp zu ändern. Dann dürfen wir aber auf Attribute der Datentypen nur über öffentliche Zugriffsfunktionen zugreifen. Wir hatten uns entschieden, allen Vorgängen, die an einem Netz beteiligt sind, eine laufende Nummer zu geben. Die Intention ist, dass diese Nummer vom Datentyp `Netz` gesetzt wird, sobald ein Vorgang dem Netz hinzugefügt wird. Wir wollen aber auf keinen Fall, dass jemand anderes als der Datentyp `Netz` diese Nummer verändert, weil dann die Konsistenz und damit das reibungslose Zusammenspiel der Datentypen nicht mehr gewährleistet ist. Hier haben wir Methoden, die wir **besonders schüt-**

zen müssen, damit sie nur von ausgewählten Datentypen benutzt werden können. Die Lösung dieses Problems ist je nach Programmiersprache unterschiedlich, in Java wird sie durch die Organisation von zusammenarbeitenden Klassen in Paketen und dem Schlüsselwort **protected** gelöst, in C++ durch *befreundete Klassen* (**friend**).

Tabelle 6.1: Wer darf auf Attribute und Funktionen zugreifen?

	C++	Java
public	Zugriff von überall möglich	Zugriff von überall möglich
protected	Zugriff nur aus abgeleiteten Klassen (vgl. Kap. 7)	Zugriff für alle Klassen im gleichen Paket und abgeleitete Klassen auch in anderen Paketen
private	Zugriff nur aus derselben Klasse	Zugriff nur aus derselben Klasse
friend	Ist eine Klasse explizit als **friend** deklariert worden, darf sie auch auf **protected**- und **private**-Elemente zugreifen.	Java kennt kein **friend**-Konstrukt, alle Klassen im gleichen Paket sind *befreundet* und erlauben Zugriff auf der Ebene von **protected**.

Während in Java die Sichtbarkeit für jedes Attribut und jede Methode neu notiert wird, gilt in C++ die notierte Sichtbarkeit, bis etwas anderes gefordert wird. Im Listing 6.5 sind jeweils alle Attribute privat, nur einige Methoden sind öffentlich, die anderen sind entweder protected (Java), oder es sind befreundete Klassen angegeben (C++).

Wenngleich niemand auf die verwendeten Datenstrukturen zugreifen kann (private), so sind sie im Beispiel doch bekannt – und das widerspricht dem Geheimnisprinzip, auch wenn der Effekt, den wir vermeiden wollten, nun nicht mehr auftreten kann. Es besteht auch die Möglichkeit, diese Informationen zunächst noch nicht preiszugeben und später erst Klassen bereitzustellen, die die tatsächlichen Implementierungen der Funktionen vorhalten (und ggf. mit eigenen Attributen und Methoden ergänzen), vgl. dazu Kap. 7. Ansonsten enthält die Header-Datei nur die Schnittstellen-Definition, die Methoden werden in der Source-Datei implementiert. Java kennt keine Unterscheidung in Header- und Source-Datei. Aber die Möglichkeit, Schnittstellen zu definieren, war den Java-Entwicklern sogar ein eigenes Schlüsselwort wert: **interface**. Die Schreibweise ist identisch mit der von Klassen, nur dass in einem Interface automatisch nur öffentliche Methoden deklariert werden können und keine Attribute.

Der in der Klasse `Vorgang` (siehe Listing 6.5) enthaltene Konstruktor `Vorgang(double d)` sowie der Destruktor (nur C++) `~Vorgang(double d)` entsprechen den Funktionen `Vorgang* erzeugeVorgang(double d)` bzw. `void zerstoereVorgang(double d)` der prozeduralen Lösung.

Das Listing 6.6 zeigt, wie die Schnittstelle `IVorgang`[1] definiert wird, die aus drei Methoden besteht. Man sagt, die Vorgangsklasse *implementiert* die Schnittstelle, wenn sie für alle Schnittstellen-Methoden Realisierungen bereitstellt. Dies wird zum Ausdruck gebracht (und danach vom Compiler überprüft), indem nach dem Klassennamen die Schlüsselworte `implements IVorgang` folgen.

Java 6.6: Eine Klasse implementiert eine Schnittstelle `IVorgang`

```
interface IVorgang {
 double getDauer();
 double getFruehAnf();
 double getSpaetEnd();
};
```

(netzplanung/v4-adt-java/IVorgang.java)

```
class Vorgang implements IVorgang {
 private double dauer,fruehanf,spaetend;
 private int id;

 public Vorgang(double d) {dauer=d; id=-1;}
 public double getDauer() { return dauer; }
 public double getFruehAnf() { ... }
 public double getSpaetEnd() { ... }
 . . .
}
```

(netzplanung/v4-adt-java/VorgangImpIVorgang.java)

Auf den ersten Blick mag diese Aufteilung der Trennung in Header- und Source-Datei ähneln, aber der Unterschied ist deutlich: Die Schnittstelle wird hier völlig unabhängig von der Klasse definiert, die diese Schnittstelle implementiert. In der Variante Header/Source-Datei stand nur die Schnittstelle einer bestimmten Klasse. Wir werden uns mit diesem Unterschied in Kap. 7 intensiv beschäftigen.

6.2.2 Standardfunktionalität in der Klassen-Schnittstelle

Wenn man konsistent alle neuen Datentypen, die man benötigt, als abstrakte Datentypen implementiert, stellt sich heraus, dass eine kleine Menge von Methoden immer und immer wieder auftaucht. Das sind Funktionen zum Erzeugen und Löschen von Instanzen, zum Kopieren oder Zuweisen sowie Vergleichsfunktionen. Es ergibt Sinn, diese Operationen bei allen Datentypen auf die gleiche Weise zu definieren statt durch Bezeichnervielfalt (`erzeugeVorgang`, `neuerVorgang` usw.) für Verwirrung zu sorgen. Die folgenden Abschnitte diskutieren diese Standardfunktionalität.

Konstruktoren

Bevor wir mit einem abstrakten Datentyp etwas anfangen können, müssen wir zuerst eine Instanz desselben anlegen. Dazu standen uns Funktionen `erzeugeVorgang` oder `erzeugeNetz` zur Verfügung.

[1] Der führende Buchstabe „I“ vor `Vorgang` soll bereits im Namen andeuten, dass es sich um ein Interface handelt.

C++ 6.7: Erzeugerfunktion, Quellcode-Datei

```
/** Neues Netz mit Start-/Endzeit anlegen */
Netz* erzeugeNetz(double sz,double ez) {
  int i,j;
  Netz *n = new Netz();
  n->startzeit = sz;
  n->endzeit = ez;
  n->anzahl = 0;
  for (i=0;i<MAX;++i) {
    for (j=0;j<MAX;++j) {
      n->nachf[i][j]=false;
    }}
  for (i=0;i<MAX;++i) n->reihenfolge[i]=i;
  return n;
}
```

Zur Erzeugung und Initialisierung eines Netzes war bisher die folgende C++-Anweisung erforderlich:

```
Netz* pn = erzeugeNetz(0, 7);
```

Die standardisierte Version einer solchen Erzeugerfunktion heißt Konstruktor und kann als Funktion ohne Rückgabewert (nicht einmal `void`) aufgefasst werden, deren Methodenname mit dem Klassennamen identisch ist. Wir können den Konstruktor auch als namenlose Funktion ansehen, die eine Instanz des Datentyps zurückliefert.

Aus der Erzeugerfunktion in Listing 6.7 wird der Konstruktor in Listing 6.8.[2]

C++/Java 6.8: Konstruktoren

```
Netz::Netz(double sz,double ez) {
  this->startzeit = sz;
  this->endzeit = ez;
  this->anzahl = 0;
  for (int i=0;i<MAX;++i)
    for (int j=0;j<MAX;++j)
      this->nachf[i][j]=0;
  for (int k=0;k<MAX;++k)
    this->reihenfolge[k]=k;
}
```

(netzplanung/v4-adt-cpp/Netz.cpp)

```
public Netz(double sz,double ez) {
  this.startzeit = sz;
  this.endzeit = ez;
  this.anzahl = 0;
  for (int i=0;i<MAX;++i)
    for (int j=0;j<MAX;++j)
      this.nachf[i][j]=false;
  for (int i=0;i<MAX;++i)
    this.reihenfolge[i]=i;
}
```

(netzplanung/v4-adt-java/Netz.java)

Ein Ausschnitt aus der Schnittstelle von `Netz` ist zur besseren Übersichtlichkeit im Listing 6.9 zusammen mit den Attributen angegeben. In Java erfolgt die Implementierung des Konstruktors direkt in der Klasse, in C++ separat in der entsprechenden Quellcode-Datei (jeweils Listing 6.8).

Die Erzeugung eines Netzes mit Hilfe des Konstruktors unterscheidet sich kaum von der Erzeugung mit der Funktion `erzeugeNetz`:

```
Netz* pn = new Netz(0, 7); // auf Heap
```

```
Netz pn = new Netz(0, 7);
```

Im Konstruktor werden entsprechend `erzeugeNetz` die Attribute initialisiert. Um die geringen Unterschiede zur Funktion `erzeugeNetz` in Listing 6.7 zu betonen, wurde hier der `this`-Zeiger explizit benutzt, im Allgemeinen wird er aber weggelassen.

[2] Das Java-Beispiel verschweigt, dass der Speicher für die Arrays (`nachf`, `reihenfolge`) erst angelegt werden muss. Diese Initialisierung kann man auch direkt bei den Attributdefinitionen vornehmen, was im Listing 6.9 zugunsten der Gegenüberstellung C++/Java-Konstruktoren durchgeführt wurde.

C++/Java 6.9: Attribute der Netz-Klasse mit Konstruktor

```cpp
class Netz {
 private:
  enum {MAX=100};
  double startzeit,endzeit;
  int anzahl;
  Vorgang *vorg[MAX]; // Knoten
  bool nachf[MAX][MAX]; // Adjazenzmatrix
  int reihenfolge[MAX];
 public:
  Netz(double startzeit,double endzeit);
  //...
```

(netzplanung/v4-adt-cpp/Netz.h)

```java
class Netz {
 private static final int MAX = 100;
 private double startzeit,endzeit;
 private int anzahl;
 private Vorgang vorg[]=new Vorgang[MAX];
 private boolean nachf[][]
              =new boolean[MAX][MAX];
 private int reihenfolge[]=new int[MAX];

 /// Konstruktor
 public Netz(double sz,double ez) {
```

(netzplanung/v4-adt-java/Netz.java)

In C++ ist es auch möglich, Instanzen der Klasse auf dem Programm-Stack zu erzeugen. Die Syntax entspricht dabei der Definition und Initialisierung von Variablen elementarer Datentypen:

```cpp
Netz n(0, 7); // Objekt auf dem Programm-Stack
double d(17.45); // Definition und Initialisierung von d
```

C++/Java 6.10: Aufbau eines Netzes

```cpp
  Vorgang *v0 = new Vorgang(1);
  Vorgang *v1 = new Vorgang(3);
  Vorgang *v2 = new Vorgang(2);
  Vorgang *v3 = new Vorgang(5);
  Vorgang *v4 = new Vorgang(2);

  netz->fuegeHinzu(v0);
  netz->fuegeHinzu(v1);
  netz->fuegeHinzu(v2);
  netz->fuegeHinzu(v3);
  netz->fuegeHinzu(v4);

  netz->setzeNachfolger(v0,v1);
  netz->setzeNachfolger(v0,v2);
  netz->setzeNachfolger(v1,v4);
  netz->setzeNachfolger(v3,v4);
```

(netzplanung/v4-adt-cpp/NetzTest.cpp)

```java
  Vorgang v0 = new Vorgang(1);
  Vorgang v1 = new Vorgang(3);
  Vorgang v2 = new Vorgang(2);
  Vorgang v3 = new Vorgang(5);
  Vorgang v4 = new Vorgang(2);

  netz.fuegeHinzu(v0);
  netz.fuegeHinzu(v1);
  netz.fuegeHinzu(v2);
  netz.fuegeHinzu(v3);
  netz.fuegeHinzu(v4);

  netz.setzeNachfolger(v0,v1);
  netz.setzeNachfolger(v0,v2);
  netz.setzeNachfolger(v1,v4);
  netz.setzeNachfolger(v3,v4);
```

(netzplanung/v4-adt-java/NetzTest.java)

Ein Beispiel für die Benutzung des Konstruktors von `Vorgang` zeigt die Anpassung unserer Testfälle in Listing 6.10 an die Klassenschnittstellen von `Vorgang` und `Netz`. Es werden Vorgänge erzeugt, beim Netz angemeldet und die Nachfolgerbeziehungen gesetzt.

Einige Konstruktoren haben besondere Namen:

- Ein Konstruktor ohne Argument heißt **Standard-** oder **Default-Konstruktor**. Ist überhaupt kein Konstruktor angegeben, wird ein Default-Konstruktor vom Compiler erzeugt, der (nur in Java) alle Attribute auf ihre Defaultwerte setzt. Sobald der Entwickler aber einen Konstruktor selbst angibt, wird vom Compiler kein Konstruktor mehr automatisch erzeugt (auch nicht, wenn der Konstruktor des Entwicklers Argumente besitzt).

- Erzeugt ein Konstruktor eine Kopie einer anderen Instanz, so spricht man von einem **Kopierkonstruktor**; er besitzt nur ein Argument desselben Typs als Referenzparameter.
- Erzeugt ein Konstruktor ein Objekt, indem er ein übergebenes Argument konvertiert (z.B. ein Konstruktor für String mit einem `double`-Argument, der aus dem `double`-Wert eine Textrepräsentation erzeugt), so nennt man ihn einen **Umwandlungskonstruktor**. Alle Konstruktoren mit genau einem Argument sind deshalb Umwandlungskonstruktoren. Sie wandeln den Typ des Arguments in die Klasse um. Der Kopierkonstruktor ist hier gewissermaßen der Sonderfall der Umwandlung der Klasse in sich selbst.

Destruktoren

Wir erinnern uns, dass jede Variable, die in C++ über den `new`-Operator im Heap angelegt wurde, von uns wieder freigegeben werden muss. Diesen Vorgang übernimmt standardmäßig der **Destruktor** einer Klasse. Analog zu den Konstruktoren ist die Bedeutung der Destruktoren in beiden Sprachen sehr unterschiedlich. Der Destruktor wird *automatisch* vom Compiler aufgerufen, wenn eine Klasseninstanz auf dem Stack angelegt wurde (was nur in C++ möglich ist). Ein im Heap angelegtes Objekt `obj` muss in C++ durch den expliziten Destruktor-Aufruf `delete obj;` entfernt werden (Arrays: `delete[] obj`). Eine wesentliche Aufgabe von Destruktoren ist es, nicht nur den Speicher, den das Objekt selbst belegt, freizugeben, sondern möglicherweise auch den Speicher von referenzierten Objekten. Weil es manchmal nicht einfach zu entscheiden ist, ob ein referenziertes Objekt von keinem anderen Objekt mehr referenziert wird (denn nur dann sollte es freigegeben werden), ist das Vermeiden von Speicherlecks manchmal recht aufwändig. Von dieser Last sind Java-Nutzer befreit, sie können einen Destruktor-Aufruf nicht erzwingen, der Garbage Collector entscheidet selbst, wann er den Speicher für nicht mehr benötigte Objekte freigibt.

Das folgende Beispiel zeigt die Implementierung eines C++-Destruktors für die Klasse Netz, bei der (zu Demonstrationszwecken) alle referenzierten Vorgänge ebenfalls freigegeben werden. Das sollte nur erfolgen, wenn wir *wissen*, dass außerhalb des Netz-Objektes niemand eine Referenz auf die Vorgänge hält. Generell gilt aber die
Tipp Regel, dass derjenige, der ein Objekt erzeugt hat, es auch wieder freigeben sollte. Da die Netz-Klasse die Vorgänge nur verwaltet, aber nicht erzeugt, wäre ein solcher Destruktor schlechte Praxis und damit ein schlechtes Design.

```
Netz::~Netz() {
  for (int i=0;i<anzahl;++i)
    delete vorg[i];
}
```

Eine weitere Aufgabe für Destruktoren ist es, andere benutzte Ressourcen wieder freizugeben, bspw. eine geöffnete Datei auch wirklich zu schließen. Derartige Funktionalität kann in Java in der Methode `finalize` untergebracht werden, die beim Zerstören eines Objekts aufgerufen wird – aber wie gesagt, der Garbage Collector entscheidet, wann das erfolgt.

Kopieren/Clonen

Das Kopieren von Objekten ist eine relativ häufig benötigte Operation, die über verschiedene Datentypen/Klassen vereinheitlicht werden sollte. In C++ wird dafür ein Kopierkonstruktor verwendet, in Java eine Kopiermethode `clone`. Der Kopier*konstruktor* hat dabei für C++ eine stärkere Bedeutung als die Kopiermethode in Java, weil er in vielen Fällen implizit vom System verwendet wird, z.B. in den beiden folgenden Funktionen `f1` und `f2`:

```
void f1(Netz n) {
  . . .
}

Netz f2() {
  Netz temp;
  . . .
  return temp;
}
```

Die Funktion `f1` hat einen Werteparameter `Netz n`, d.h. der Wert der Variablen (des Objektes) `n` wird in die Funktion hineinkopiert (call-by-value), und dazu verwendet das System den Kopierkonstruktor. Entsprechend verwendet das System in der Funktion `f2` den Kopierkonstruktor, um den Wert von `temp` zurückzugeben.

Die beschriebene Situation kann es in Java nicht geben, weil strukturierte Datentypen niemals in Wertesemantik zurückgegeben werden. Die Integration der Kopierfunktion in die Sprache ist daher nicht so stark. Der Standardweg für das Kopieren von Objekten geht über eine Methode `clone`, die in einer Schnittstelle `Cloneable` wie folgt definiert ist:

```
interface Cloneable {
  Object clone();
}
```

Der Kopierkonstruktor wird wie jeder andere Konstruktor aufgerufen, die `clone`-Methode wie jede andere Methode:

```
Vorgang v1;

Vorgang v2(v1); // Kopierkonstruktor
```

```
Vorgang v1 = new Vorgang();

Vorgang v2 = (Vorgang) v1.clone();
```

Wenn eine Klasse (wie `Vorgang`) nun diese Schnittstelle einhält (`implements Cloneable`), so bedeutet das, dass ein `Vorgang` die Methode `clone` zur Verfügung stellt. Weil diese Schnittstelle aber zu einem Zeitpunkt definiert wurde, als noch niemand an unsere Vorgangs-Klasse gedacht hat, kann die Schnittstelle als Rückgabewert naturgemäß nicht `Vorgang` liefern (sondern allgemeiner ein Objekt, vgl. Kap. 7). Trotzdem ist der zurückgegebene Typ identisch mit dem des kopierten Objekts, es ist aber eine explizite Typumwandlung (type-cast) vorzunehmen.

Listing 6.11 implementiert die Java-`clone`-Methode (Java) und den C++-Kopierkonstruktor.

C++/Java 6.11: Kopierkonstruktor/Clone-Methode von `Vorgang`

```
class Vorgang {
 private:
  double dauer,fruehanf,spaetend;
  int id;
 public:
  Vorgang(const Vorgang& v);
  ...
}

Vorgang::Vorgang(const Vorgang& v) {
  dauer = v.dauer;
  fruehanf = v.fruehanf;
  spaetend = v.spaetend;
  id = v.id;
}
```

(netzplanung/v4-adt-cpp/VorgangCC.cpp)

```
class Vorgang implements Cloneable {
 private double dauer,fruehanf,spaetend;
 private int id;

 public Vorgang(double d) {
   dauer = d; id = -1; }
 public Object clone() {
   Vorgang v = new Vorgang(dauer);
   v.fruehanf = fruehanf;
   v.spaetend = spaetend;
   v.id = id;
   return v;
 }
 ...
}
```

(netzplanung/v4-adt-java/CloneBeispiel.java)

Sowohl in Java als auch in C++ gibt es Mechanismen für eine Default-Kopierfunktionalität. Wenn in C++ eine Klasse keinen Kopier-Konstruktor explizit definiert, generiert das System automatisch eine Default-Implementierung. In Java können wir durch `super.clone()` eine Default-Kopie anlegen (siehe Listing 6.13). In beiden Fällen wird eine komponentenweise Kopie der Klasseninstanz erzeugt – d.h. ein Objekt, das Zeiger bzw. Referenzen beinhaltet, wird **flach kopiert**: Es werden die Zeiger, aber nicht die damit verbundenen Speicherbereiche im Heap kopiert!

Die Default-Implementierung kann in der Regel nur dann korrekt sein, wenn die Klasse keine dynamisch verwalteten Daten enthält. Wenn für eine unabhängige Kopie weitere Objekte kopiert werden müssen (und der Kopierkonstruktor dies durchführt), spricht man von einer **tiefen Kopie**.

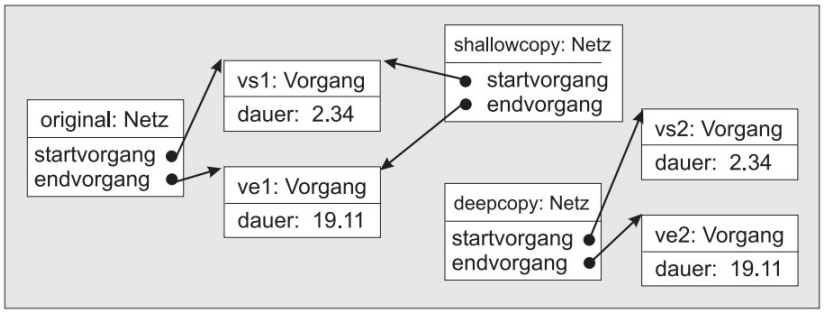

Bild 6.1: Tiefe und flache Kopie eines Originals.

Unsere Klasse `Netz` ist ein Beispiel für eine Klasse, in der weitere Vorgänge referenziert werden. Kopieren wir nur die Referenzen/Zeiger, dann zeigt die Kopie auf *dieselben* Objekte und die Kopien sind nicht unabhängig voneinander. Im Beispiel in Bild 6.1 wurden eine flache (Objekt `shallowcopy`) und eine tiefe Kopie (Objekt `deepcopy`) des Objekts `original` erzeugt: Wird bei `shallowcopy` die Dauer des Start-

vorgangs geändert, so ändert sich gleichzeitig auch die Dauer des Startvorgangs im Objekt `original`. Nur bei einer tiefen Kopie sind beide Netz-Instanzen hinterher wirklich unabhängig voneinander.

Das folgende Beispiel in den Listings 6.12–6.13 zeigt nochmals die Attribute der Netz-Klassen und die Umsetzungen einer tiefen Kopie.

C++/Java 6.12: Attribute von `Netz`, die vom Kopierkonstruktor bzw. von der Clone-Methode zu berücksichtigen sind

```
class Netz {
 private:
  enum {MAX=100};
  double startzeit,endzeit;
  int anzahl;
  Vorgang *vorg[MAX]; // Knoten
  bool nachf[MAX][MAX]; // Adjazenzmatrix
  int reihenfolge[MAX];
```

(netzplanung/v4-adt-cpp/Netz.h)

```
class Netz {
 private static final int MAX = 100;
 private double startzeit,endzeit;
 private int anzahl;
 private Vorgang vorg[]=new Vorgang[MAX];
 private boolean nachf[][]
              =new boolean[MAX][MAX];
 private int reihenfolge[]=new int[MAX];
```

(netzplanung/v4-adt-java/Netz.java)

C++/Java 6.13: Kopierkonstruktor/Clone-Methode `Netz`

```
Netz::Netz(const Netz& n) {
  startzeit = n.startzeit;
  endzeit = n.endzeit;
  anzahl = n.anzahl;
  for (int i=0;i<MAX;++i)
    vorg[i]=new Vorgang(n.vorg[i]);
  for (int i=0;i<MAX;++i)
    for (int j=0;j<MAX;++j)
      nachf[i][j]=n.nachf[i][j];
  for (int i=0;i<MAX;++i)
    reihenfolge[i]=n.reihenfolge[i];
}
```

(netzplanung/v4-adt-cpp/VorgangCC.cpp)

```
public Object clone() {
 Netz n = super.clone(); // flache Kopie
 n.vorg = (Vorgang[]) vorg.clone();
 for (int i=0;i<MAX;++i)
     n.vorg[i] = (Vorgang) vorg[i].clone();
 n.nachf = (int[][]) nachf.clone();
 n.reihenfolge = (int[]) reihenfolge.clone();
}
```

(netzplanung/v4-adt-java/CloneBeispiel.java)

Zuweisungsoperator

Manchmal sollen die Attribute eines Objekts `Netz x` identisch auf alle Attribute eines Objekts `Netz y` gesetzt werden. Ist in Java eine solche Zuweisungsfunktionalität gefordert, so bringt man sie in einer Methode `set(Netz n)` unter. In C++ entsteht der Bedarf für eine solche Funktionalität aber auch indirekt durch eine Zuweisung `x=y` (in Wertesemantik).[3] Darum sollte in C++ der Zuweisungsoperator `=` überladen werden:

```
Netz& operator=(const Netz& n) {
  ...
}
```

[3] In Java wird diese Anweisung nur die Referenz `x` auf dasselbe Objekt zeigen lassen, auf die Referenz `y` zeigt.

Tipp

Der Zuweisungsoperator selbst sollte als Erstes immer überprüfen, ob es sich um eine Selbstzuweisung handelt – in dem Fall kann einfach `*this` zurückgegeben werden. Andernfalls sollten die Attributwerte einzeln kopiert werden, wobei dynamische Datenstrukturen ggf. in ihrer Größe vorher angepasst werden müssen:

```
Netz& Netz::operator=(const Netz& n){
 if (this != &n) {
  startzeit = n.startzeit;
  endzeit = n.endzeit;
  anzahl = n.anzahl;
  for (int i=0;i<MAX;++i) {
    delete vorg[i];// Heap−Speicherfreigabe
    vorg[i]=new Vorgang(n.vorg[i]);
    }
  for (int i=0;i<MAX;++i) {
    for (int j=0;j<MAX;++j) {
      nachf[i][j]=n.nachf[i][j];
      }
    reihenfolge[i]=n.reihenfolge[i];
    }
  }
 return *this;
}
```

Zu beachten ist beim Zuweisungsoperator im Unterschied zum Kopierkonstruktor, dass das Objekt, auf das `this` verweist, bereits existiert und damit alle Attribute sinnvolle Werte haben (sollten). Falls sie in den Heap-Speicher verweisen, muss der gegebenenfalls freigegeben werden (`delete vorg[i]`).

Vergleich

Eine weitere häufige Operation ist der Vergleich von Objekten. Der Vergleich zweier Zeiger oder Referenzen `a==b` liefert `true`, wenn die Zeiger identisch sind. Oftmals wird aber die Frage nach Wertgleichheit gestellt, d.h. ob zwei verschiedene Objekte den gleichen Zustand haben (zwei Vorgänge bspw. die gleiche Dauer, früheste Start- und späteste Endzeit). In C++ kann das Ergebnis des Vergleichs `a==b` (`a` und `b` sind Referenzen oder Werte) speziell für neue Typen durch eine globale Funktion überladen werden:

```
bool operator==(const Vorgang& v1,const Vorgang& v2) { ... }
```

In Java ist für diese Zwecke die Methode

```
public boolean equals(Vorgang v) { ... }
```

vorgesehen, sodass ein Wertevergleich dann durch `a.equals(b)` durchgeführt wird (`a==b` vergleicht in Java immer nur die Referenzen, nie die referenzierten Objekte).

Während in C++ analog dazu auch die Operatoren `!=`, `<`, `>`, `<=` und `>=` umdefiniert werden können, wird in Java üblicherweise das Interface `Comparable` benutzt, das eine Methode `compareTo(..)` voraussetzt.

```
interface Comparable {
 int compareTo(Object obj);
}
```

Aus denselben Gründen wie schon bei Cloneable (siehe Seite 141) tritt hier der allgemeinere Typ `Object` (statt `Vorgang`) als Argument auf. Das Argument muss in einer Implementierung für Vorgangsvergleiche auf den Typ `Vorgang` gecastet werden. Der Rückgabewert ist eine ganze Zahl, wobei

- ein negativer Wert bedeutet, dass das aktuelle Objekt (`this`) kleiner ist als `obj`;
- ein Wert 0 bedeutet, dass beide Objekte (wertmäßig) identisch sind;
- ein positiver Wert bedeutet, dass das aktuelle Objekt (`this`) größer ist als `obj`.

Dasselbe Konzept ist aber auch in C/C++ üblich, etwa bei der Funktion `strcmp` der Standardbibliothek.

Minimale Standardschnittstelle

Die letzten Absätze haben die am häufigsten benötigte Standardfunktionalität diskutiert, die wir im Umgang mit Objekten benötigen.

- **Konstruktor**: Ohne Konstruktor kann der Lebenszyklus eines Objekts nicht beginnen. Darum bieten C++ und Java einen Default-Konstruktor an. Um immer für eine geeignete Initialisierung eines Objekts zu sorgen, ist die Ersetzung des Default-Konstruktors durch einen Konstruktor mit Parametern sinnvoll. Als Konsequenz der Wertesemantik muss in C++ ein parameterloser Default-Konstruktor bestehen bleiben, wenn es möglich sein soll, ein Array durch `new MyClass[10]` anzulegen. Während in Java hier nur Platz für 10 Referenzen auf `MyClass` angelegt wird, reserviert C++ schon Platz für 10 `MyClass`-Objekte, die über den Default-Konstruktor initialisiert werden.
- **Destruktor**: Ebenso wie der Konstruktor markiert der Destruktor einen entscheidenden Punkt im Objektlebenszyklus. In C++ sollte er also auf keinen Fall fehlen, um Speicherlecks zu vermeiden. In Java übernimmt diese Aufgabe der Garbage Collector.
- **Clonen**: Die Möglichkeit zur Duplikation von Objekten ist nicht immer gewünscht, oft muss sie sogar verhindert werden (vgl. Entwurfsmuster *Singleton* im Abschn. **??**). Dass die Existenz eines Kopierkonstruktors für C++ trotzdem empfohlen wird, liegt daran, dass er in einer unscheinbaren Zeile `MyClass temp = instance` implizit aufgerufen wird, um von `instance` die Kopie `temp` anzulegen. Wird statt Werte- nur Referenzsemantik benutzt, wird bei dieser Anweisung nur die Referenz kopiert, aber keine Kopie angelegt. In Java spielt das Clonen eine geringere Rolle als der Kopierkonstruktor in C++.
- **Zuweisungsoperator**: Das Beispiel aus dem letzten Absatz lässt sich direkt auch auf den Zuweisungsoperator übertragen. In der C++-Anweisungsfolge `MyClass temp; temp = instance;` wird nun zuerst der Default-Konstruktor und dann der Zuweisungsoperator aufgerufen, folglich sollte ein Zuweisungsoperator in C++ standardmäßig definiert werden. Der Java-Entwickler hat es einfacher, weil hier wieder nur Referenzen kopiert werden, ein spezieller Zuweisungsoperator für Objekte (in Wertesemantik) existiert nicht.
- **Vergleichsfunktionen**: Man muss Objekte nicht vergleichen können, wenn man sie aber in einen Container einzufügen gedenkt, sind Vergleichsfunktionen äußerst hilfreich, um die Objekte auch wiederzufinden.

Liegt für eine C++-Klasse die explizite Implementierung von (Default-)Konstruktor, Destruktor, Kopierkonstruktor und Zuweisungsoperator vor, spricht man von ei-

ner Klasse mit *minimaler Standardschnittstelle*. In Java genügt hierfür `clone()`-Funktionalität und ein (Default-)Konstruktor. Sind darüber hinaus Vergleichsfunktionen definiert, wird die Klasse auch als *Nice-Class* bezeichnet.

Tipp

In einigen Fällen kann es z.B. aus Effizienzgründen sinnvoll sein, auf die Implementierung der minimalen Standardschnittstelle zu verzichten und stattdessen die Default-Implementierung der Sprache zu verwenden. Ausführlich werden die Vor- und Nachteile der expliziten Implementierung in [13, 14] diskutiert. Gerade der Anfänger sollte jedoch schon aus Übungsgründen für jede von ihm neu entwickelte Klasse eine minimale Standardschnittstelle vorsehen oder den Aufruf der betreffenden Methoden durch Deklaration als `private` unterbinden.

6.2.3 Klassenglobale Attribute und Methoden

Normalerweise sind die in einer Klasse definierten Attribute und Methoden *objektbezogen*, d.h. zu jedem Objekt der Klasse gehört ein entsprechender Satz der in der Klasse definierten Attribute, und jedem Objekt sind die in der Klasse definierten Methodenfunktionen zugeordnet:

```
class X {
   int x, y, z;
  public:
   void f1() {...}
   void f2() {...}
   void f3() {...}
};
. . .
X x1, x2, x3;
x1.f1(); x2.f1(); x3.f1(); x1.f2(); . . .
```

```
class X {
  private int x, y, z;

  public void f1() {...}
  public void f2() {...}
  public void f3() {...}
};
. . .
X x1=new X(), x2=new X(), x3=new X();
x1.f1(); x2.f1(); x3.f1(); x1.f2(); . . .
```

Jedes Objekt der Klasse `X` hat seinen individuellen Satz von Variablen und belegt damit einen Speicherbereich, der dem für drei Variablen vom Typ `int` entspricht:[4]

`sizeof(x1)` ≡ `sizeof(x2)` ≡ `sizeof(x3)` ≡ `3*sizeof(int)`

Für besondere Fälle gibt es aber auch die Möglichkeit, Attribute und Methoden innerhalb einer Klasse zu definieren, die *klassenbezogen* oder *klassenglobal* sind. Sie werden mit dem Bezeichner `static` spezifiziert.

Das *Klassenattribut* `xyz` im Listing 6.14 *belastet* die Objekte nicht, d.h. sie belegen den gleichen Speicherplatz (`3*sizeof(int)`) wie oben! Statische Elemente sind der Klasse (dem Typ) zugeordnet, sie sind objektübergreifend vorhanden und verhalten sich wie globale Variablen bzw. wie globale Funktionen, deren Geltungsbereich (scope) auf die Klasse beschränkt ist, und werden deshalb außerhalb der Klasse in C++ über den Bereichsoperator `::` (`X::xyz`) bzw. in Java über den Klassennamen `X.xyz` angesprochen. Sie unterliegen den gleichen Zugriffsrechten wie die üblichen nichtstatischen Methoden und Attribute. Statische Methoden können auch *objektbezogen*

[4] Zusätzliche administrative Daten, die pro Objekt möglicherweise gespeichert werden müssen, ignorieren wir hier.

C++/Java 6.14: Statische Attribute und Methoden der Klasse `X`

```cpp
class X {
   int x, y, z;
 public:
   static int xyz;
   void f1() {...}
   void f2() {...}
   void f3() {...}
   static void g() {...}
};
. . .
X x1, x2, x3;
x1.f1(); x2.f1(); x3.f1(); x1.f2(); . . .
. . .
X::g(); . . . int i = X::xyz; . . .
```

(ADT/X.cpp.sht)

```java
class X {
  private int x, y, z;
  public static int xyz;

  public void f1() {...}
  public void f2() {...}
  public void f3() {...}
  public static void g() {...}
};
. . .
X x1=new X(), x2=new X(), x3=new X();
x1.f1(); x2.f1(); x3.f1(); x1.f2(); . . .
. . .
X.g(); . . . int i = X.xyz;
```

(ADT/X.java.sht)

aufgerufen werden, statt `X::g()`/`X.g()` auch `x1.g()`, was aber der Klarheit wegen vermieden werden sollte.

6.2.4 Spezifikation von abstrakten Datentypen

Ein abstrakter Datentyp, oder noch allgemeiner eine Schnittstelle, kapselt (trennt) die Details der Implementierung von der Benutzung des Datentyps. Dabei besteht die Spezifikation einer Schnittstelle zum einen aus den Funktionsprototypen (Syntax) – ein Verstoß gegen die Syntax wird vom Compiler mit Syntax-Fehlermeldungen geahndet – und zum anderen aus der Semantik. Natürlich sollen für alle Methoden selbsterklärende Namen gewählt werden, aber es ist illusorisch zu glauben, dass solche Namen die Funktionalität eindeutig charakterisieren. Die schwierige Aufgabe der Spezifikation besteht darin, ohne schon die Programmierung vorzunehmen eindeutig die Semantik der Operationen zu beschreiben, sodass einerseits der Benutzer weiß, wie die Funktionen korrekt anzuwenden sind, und andererseits der Entwickler weiß, wie die Funktionen richtig zu implementieren sind.

Wir wollen hier nur eine Art der Spezifikation ansprechen, und zwar die **axiomatische Spezifikation**. Betrachten wir als Beispiel die Lehrbuch-Spezifikation des abstrakten Datentyps für einen Stapel[5] in Tabelle 6.2. Hier steht `base` für einen Basisdatentyp, von dem Elemente im Stack gespeichert werden sollen. Es gibt fünf Operationen: `empty`, `push`, `pop`, `top`, `isEmpty`. Die Syntax kann dem Abschnitt **operators** aus der Spezifikation entnommen werden. Das Symbol $\perp$ steht für ein Fehlersymbol, dessen Entsprechung in den betrachteten Programmiersprachen in einem Null-Zeiger gesehen werden kann. Die mit **axioms** eingeleiteten Gleichungen stellen Bedingungen dar, die für alle Stacks s und Basiselemente e gelten müssen.

[5] Ein `Stapel` ist eine Datenstruktur, in der die Daten wie die Teller im Schrank verwaltet werden. Teller werden mit der Operation `push` oben auf dem (Teller-)`Stapel` im Schrank abgelegt, und anschließend wird der zuletzt abgelegte Teller mit der Operation `pop` wieder vom (Teller-)`Stapel` entfernt.

Tabelle 6.2: Axiomatische Spezifikation eines Stapels

adt	stack	
sorts	stack, base, bool	
operators	empty : $\rightarrow$ stack	
	push : stack $\times$ base $\rightarrow$ stack	
	pop : stack $\rightarrow$ stack $\cup\{\perp\}$	
	top : stack $\rightarrow$ base $\cup\{\perp\}$	
	isempty : stack $\rightarrow$ bool	
axioms	isempty(empty) = true	(1)
	isempty(push(s,e)) = false	(2)
	pop(empty) = $\perp$	(3)
	pop(push(s,e)) = s	(4)
	top(empty) = $\perp$	(5)
	top(push(s,e)) = e	(6)

Der einzige Operator, den wir anwenden können, ohne bereits einen Stack zu haben, ist `empty`. Dieser Operator entspricht einem Konstruktor. Der Name des Operators `isEmpty` suggeriert eine Abfrage, ob der Stack leer ist. Aber auf diese sprachliche Beschreibung wollen wir uns nicht verlassen, die Axiome definieren die Bedeutung mit zwei Gleichungen ganz genau: `isempty(empty)=true`, d.h. für einen mit `empty` erzeugten Stack soll `isEmpty` wahr sein. Wenn wir für einen beliebigen Stack `s` (beispielsweise `s=empty()`) `push(s,e)` aufrufen, dann ist der Stack nicht mehr leer (zweites Axiom). Wenn wir aber versuchen, von einem leeren Stack ein Element zu entfernen (`pop(empty)`), dann gelangen wir in einen Fehlerzustand (drittes Axiom). Es ist nicht einfach, diese Menge von Gleichungen zu finden, aber in diesem Fall spezifizieren sie die Bedeutung der Operatoren ganz genau, ohne sich auf sprachliche Ungenauigkeiten einzulassen.

Umsetzung in einer Programmiersprache: Die Notation der Funktionen entspricht unserer prozeduralen Realisierung; auch hier ist das erste Argument jeder Funktion der Stack, auf den wir die Operation anwenden wollen. Aus diesem Argument wird in objektorientierten Sprachen das Objekt der Methode (**this**). Eine Übertragung in ein Java-Interface sähe für den Datentyp `Base` damit etwa wie folgt aus:

```
interface Stack {
  // Stack();
  Stack push(Base b);
  Stack pop();
  Base top();
  boolean isEmpty();
}
```

Die Spezifikation der Semantik über Axiome kann direkt in Tests umgesetzt werden. Dieser Test prüft die Axiome der Spezifikation und kann für alle Implementierungen

der Schnittstelle Test verwendet werden. Im Beispiel ist in den Kommentaren das benutzte Axiom referenziert.[6]

```
boolean test1(Base b) {
  Stack s = new Stack();
  if (!s.isEmpty()) return false; // (1)
  s.push(b);
  if (s.isEmpty(e)) return false; // (2)
  Base x = s.top();
  if (x!=b) return false; // (6)
  s.pop();
  if (!s.isEmpty()) return false; // (4) (1)
  x = s.top();
  if (s!=null) return false; // (5)
  Stack s1 = new Stack();
  try {s1.pop(); return false;} catch { /* nichts, alles ok */ } // (3)
  return true;
}
```

Wie sähe eine solche axiomatische Spezifikation für die Klassen `Vorgang` oder `Netz` aus? Wir gehen im folgenden Listing zum Beispiel davon aus, dass ein mit Dauer 4.0 erzeugter `Vorgang` hinterher auch diese Dauer liefert (1), dass ein `Netz` zu Anfang keine Vorgänge enthält (2), dass ein `Vorgang`, der in ein leeres Netz eingefügt wurde, an Index 0 wiedergefunden werden kann (3) usw. Das Schreiben von Tests ist im Grunde nichts anderes als ein Beitrag zur axiomatischen Spezifikation von Datentypen. Gegenüber der Spezifikation in Textform (oder im Kommentar wie in Abschn. ??) haben die Tests den Vorteil, dass wir ihre Gültigkeit automatisch prüfen lassen können.

```
boolean test1(Base b) {
  Vorgang v = new Vorgang(4.0);
  if (v.getDauer()!=4.0)) return false; // (1)
  Netz n = new Netz(0,10);
  if (n.getVorgaenge()!=0) return false; // (2)
  n.fuegeHinzu(v);
  if (n.getVorgang(0)!=v) return false; // (3)
  return true;
}
```

Entsprechend sind wir auch bereits im Abschn. 4.5.2 bei der Spezifikation der Liste vorgegangen. Weitere Beispiele für eine axiomatische Spezifikation finden Sie bspw. in [48].

6.2.5 Anwendung: Implementierung von *LogTrace*

C++

Wir haben gesehen, dass der Java-Entwickler stärker von der Laufzeitumgebung bei der Verwaltung des Heap-Speichers unterstützt wird. Er muss sich nicht um die

[6] Ein großer Vorteil von Tests ist, dass hier die letzten Feinheiten geklärt werden *müssen*. Die mathematisch-axiomatische Schreibweise sagt in Axiom 6 nichts darüber aus, ob `pop(push(s,e))` wertmäßig identisch `e` ist (`pop(push(s,e)).equals(e)`) oder ob es sich wirklich um das gleiche Objekt handelt (`pop(push(s,e))==e`). Erst der Test macht es unmissverständlich.

Freigabe von nicht mehr benötigten Variablen auf dem Heap kümmern. Umgekehrt hat der Entwickler aber keinen direkten Einfluss darauf, wann eine Variable vom Heap endgültig entfernt wird, d.h. wann der Garbage Collector die Methode `finalize` aufruft.

Diese Eigenschaft von C++ werden wir uns aber im Folgenden bei der Implementierung der Logging-Funktionalität von *LogTrace* zu Nutze machen. Im Abschn. ?? haben wir die *LogTrace*-Bibliothek kennengelernt, um das Betreten und vor allem das Verlassen von Funktionen einfach protokollieren zu können.

Die Protokollierung des Betretens der Funktionen `funk1` bzw. `funk2` erfolgt jeweils zu Beginn der Funktion durch Aufruf der `LOG_FUNCTION`-Anweisung.

```
void funk1() {
  LOG_FUNCTION("Nsp", "funk1", "");
  int i=44;
  /*... */
  if (...) return
  funk2();
  /*... */
  }
```

```
void funk2() {
  LOG_FUNCTION("Nsp", "funk2, "");
  double d=44.2;
  /*... */
  if (...) throw X(...);
  if (...) return;
  /*... */
  }
```

Die Funktion enthält keinen expliziten Code zur Protokollierung des Verlassens, und trotzdem erfolgt dies, sobald die Funktion verlassen wird. Hierbei ist es gleichgültig, wie viele `return`-Anweisungen die Funktion enthält – mehr noch, die Protokollierung erfolgt sogar, wenn die Funktion durch Werfen einer Ausnahme verlassen wird. Wie können wir das in C++ implementieren?

Wir nutzen hier die Eigenschaft von C++, dass das Laufzeitsystem beim Verlassen eines Blocks, z.B. einer Funktion, automatisch immer für alle auf dem Programm-Stack erzeugten Variablen die Destruktoren aufruft, d.h. wir müssen in der `LOG_FUNCTION`-Anweisung eine Instanz einer Klasse (z.B. `LogFunc`) anlegen, die über einen geeigneten Konstruktor und einen geeigneten Destruktor verfügt. Eine mögliche Implementierung von `LogFunc` zeigt Listing 6.15.

Der Konstruktor prüft zunächst, ob eine Protokollierung der Funktion gewünscht wird (Methodenaufruf von `inIdDateiVorhanden`), d.h., ob sich die übergebenen Ids in der Id-Datei befinden, die der `LOGTRACE_INIT`-Anweisung bei der Initialisierung übergeben wurden. Ist das der Fall, wird der Name der Funktion geeignet in der Trace-Datei protokolliert und der Name der Funktion in einem Attribut von `LogFunc` abgespeichert. Im Destruktor brauchen wir dann nur noch den gespeicherten Funktionsnamen ausgeben. Der Destruktor der Klasse `LogFunc` wird automatisch aufgerufen, sobald die Funktion, d.h. der Gültigkeitsbereich der Instanz von der Klasse `LogFunc`, verlassen wird.

Die `LOG_FUNCTION`-Anweisung wird als Makro implementiert, die zu einer leeren Anweisung expandiert wird, sofern die Präprozessor-Variable `LOGTRACE_DEBUG` nicht definiert ist. Damit ist es möglich, die *LogTrace*-Bibliothek beim Debuggen einzusetzen. Wenn es aber anschließend auf Laufzeiteffizienz ankommt, wird ohne die Präprozessor-Anweisung `LOGTRACE_DEBUG` übersetzt, und die beim Debuggen genutzten Vorteile verursachen nun keinen Laufzeit-Overhead.

C++ 6.15: Implementierung von *LogTrace*

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

extern ofstream* trDatei; //Trace-Datei
class LogFunc {
public:
  LogFunc(string ns,string cl,string fn);
  ~LogFunc();
  bool istAusgabeAktiv() const
    {return ausgabeAktiv};
private:
  string name;
  bool ausgabeAktiv;
  bool inIdDateiVorhanden(
    const string& ns,
    const string& cl,
    const string& fn) const;
  void ausgabe(const string& s) const;
};
```

(ADT/LogFunc/LogFunc.h)

```
LogFunc::LogFunc
 (
 string ns,
 string cl,
 string fn
 ) {
  ausgabeAktiv = false;
  if (inIdDateiVorhanden(ns,cl,fn)) {
    ausgabeAktiv = true;
    name = ns + ":" +cl +":" +fn;
    ausgabe(name);
    }
  }

LogFunc::~LogFunc(){
  if (true == ausgabeAktiv) {
  ausgabe(name);
  }
}
```

(ADT/LogFunc/LogFunc.cpp)

```
#ifdef LOGTRACE_DEBUG
# define LOG_FUNCTION(ns, fn, rest) \
  LogFunc logf(ns, "", fn);    \
  if (logf.istAusgabeAktiv() {   \
    *trDatei << rest;          \
    }
```

```
#else
# define LOG_FUNCTION(ns, fn, rest) \
  /* do nothing */
#endif
```

Die `LOG_METHOD`-Anweisung

Entsprechend dem Makro `LOG_FUNCTION` zum Einsatz in Funktionen stellt die *LogTrace*-Bibliothek die `LOG_METHOD`-Anweisung zur Protokollierung des Betretens und Verlassens von Methoden zur Verfügung. Sie verfügt über einen weiteren Parameter:

```
LOG_METHOD(ns, cl, meth, command)
```

Für den zusätzlichen, zweiten Parameter `cl` sollte der Name der Klasse, zu der die Methode gehört, als String eingesetzt werden. Wenn sich mindestens eine der drei Zeichenketten `ns`, `cl` bzw. `meth` in der Id-Datei befindet, erfolgt die Ausgabe `ns::cl::meth` in die Trace-Datei, gefolgt von der in runden Klammern eingeschlossenen Ausgabe, die `command` bewirkt.

Die `LOG_BLOCK`-Anweisung

Entsprechend gibt es die `LOG_BLOCK`-Anweisung, die zur Protokollierung des Betretens und Verlassens von Verbundanweisungen (Blöcken) verwendet wird. Das Makro hat nur einen Zeichenketten- und einen Kommando-Parameter:

```
LOG_BLOCK(blockid, command)
```

Wenn sich die Zeichenkette `blockid` in der Id-Datei befindet, erfolgt die Ausgabe `blockid` in die Trace-Datei, gefolgt von der in runden Klammern eingeschlossenen Ausgabe, die `command` bewirkt.

C++ 6.16: Beispiel für `LOG_METHOD` und `LOG_BLOCK`

```
void Netz::fuegeHinzu(Vorgang *v) {
  LOG_METHOD("NP","Netz","fuegeHinzu",
     "this=" << this);
  if (v->getId()!=-1)
    /* Vorgang darf nur einem
       Netz zugeordnet sein */
    cout << "Fehler!" << endl;
  v->setId(this->anzahl);
  this->vorg[v->getId()]=v;
  this->anzahl++;
  this->reihenfolge[v->getId()] =
                      v->getId();
}
```

(netzplanung/v4-adt-cpp-LogTrace/Netz.cpp)

```
bool Netz::plane() {
  LOG_METHOD("NP","Netz","plane",
     "this=" << this);

  bestimmeReihenfolge();
  for (int k=0;k<this->anzahl;++k) {
    LOG_BLOCK("pl-loop", "k=" << k)
    int v = this->reihenfolge[k];
    this->berechneVorwaerts(v);
  }
  for (int i=this->anzahl-1;i>-1;--i) {
    LOG_BLOCK("pl-loop", "i=" << i)
    int v = this->reihenfolge[i];
    this->berechneRueckwaerts(v);
  }
  return istDurchfuehrbar();
}
```

Der Code im Listing 6.16 veranschaulicht die Verwendung der beiden Makros. Eine mögliche Ausgabe könnte z.B. sein:

```
      > func "NP::Netz::fuegeHinzu(this=00325F48)"
      < func "NP::Netz::fuegeHinzu(this=00325F48)"
    ...
      < func "NP::Netz::fuegeHinzu(this=00325F48)"
      > func "NP::Netz::plane(this=00325F48)"
        > block "k=0 [pl-loop]"
        < block "k=0 [pl-loop]"
        > block "k=1 [pl-loop]"
        < block "k=1 [pl-loop]"
       ...
        > block "k=4 [pl-loop]"
        < block "k=4 [pl-loop]"
        > block "i=4 [pl-loop]"
        < block "i=4 [pl-loop]"
       ...
        > block "i=0 [pl-loop]"
        < block "i=0 [pl-loop]"
      < func "NP::Netz::plane(this=00325F48)"
```

Die `LOG_CONSTRUCTOR`- und `LOG_DESTRUCTOR`-Anweisungen

Die `LOG_CONSTRUCTOR`-Anweisung dient zur Protokollierung des Betretens und Verlassen von Konstruktoren:

```
LOG_CONSTRUCTOR(ns, cl, command)
```

Syntax und Semantik entsprechen zunächst der `LOG_METHOD`-Anweisung. Allerdings muss die Id-Datei zusätzlich noch die Zeichenkette `Constructor` enthalten. Bei vorhandener Zeichenkette `Constructor` in der Id-Datei erfolgt allerdings in jedem Fall am Ende des Programms eine Protokollierung der Anzahl von Konstruktor- bzw. `LOG_CONSTRUCTOR`-Aufrufen.

Entsprechend gibt es die `LOG_DESTRUCTOR`-Anweisung:

```
LOG_DESTRUCTOR(ns, cl)
```

Wenn sich die Zeichenkette `Destructor` in der Id-Datei befindet, erfolgt eine Protokollierung der Anzahl der aufgerufenen Destruktoren. Werden somit `LOG_CONSTRUCTOR` und `LOG_DESTRUCTOR` konsequent in allen Konstruktoren und Destruktoren eingesetzt, so kann man am Ende des Programms sehr einfach ermitteln, ob bestimmte auf dem Heap angelegte Klasseninstanzen nicht wieder freigegeben wurden.

Eine mögliche Trace-Dateiausgabe könnte z.B. wie folgt aussehen:

```
> idusage
         Constructor/20     Destructor/4
               NP/44              main/1          pl-loop/40
< idusage
```

In diesem Fall wurden 16 Destruktoren zu wenig aufgerufen, oder aber es fehlen `LOG_DESTRUCTOR`-Anweisungen im Code.

Eine vollständige Implementierung der hier vorgestellten *LogTrace*-Bibliothek kann auf unserer Homepage im Verzeichnis *netzplanung/liblogtrace* heruntergeladen werden. **www**

6.2.6 Übungen

Übung 6.1: Erweitern Sie die Vorgangs-Klasse, sodass die Anzahl der vorhandenen Vorgangs-Objekte gezählt wird. Im Konstruktor wird bei Erzeugung eines Vorgangs jeweils die Anzahl der nun vorhandenen Vorgangs-Objekte inkrementiert, im Destruktor dekrementiert.

Außerdem soll eine Methode `getAnzVorgaenge` von `Vorgang` implementiert werden, die die Anzahl der aktuell vorhandenen Vorgänge liefert. Beim Programmstart sollte diese Methode 0 zurückliefern. Vergessen Sie nicht, geeignete Tests zu implementieren.

Übung 6.2: Gegeben ist die folgende Schnittstelle der Klasse `Vorgang` sowie ein kleines Anwendungsprogramm dazu. **C++**

```
class Vorgang {
 public:
  Vorgang(double d);
  Vorgang(const Vorgang& v2);
  ~Vorgang();
  double getDauer() const;
  void setDauer(double d);
 private:
  double dauer;
};
```

(./ADT/Plus/Vorgang.h)

```
Vorgang plus1(Vorgang va, Vorgang vb);

int main() {
  Vorgang v1(4), v2(5);
  Vorgang v3(0);
  cout << "Vor plus\n";
  v3 = plus1(v1, v2);
  cout << "Nach plus\n";
  }
```

a) Implementieren Sie zunächst die Quellcode-Datei der Klasse `Vorgang` und die Funktion `plus1`, die die Dauer der Vorgänge addiert.
b) Wie viele Konstruktoren und Destruktoren von `Vorgang` werden zwischen den beiden Bildschirmausgaben aufgerufen? Begründen Sie Ihre Entscheidung. Weisen Sie die Richtigkeit Ihrer Behauptung nach. Verwenden Sie zum Nachweis die *Log-Trace*-Bibliothek.
c) Verändern Sie die Schnittstelle der Funktion `plus1` und ggf. auch den Aufruf, sodass weniger Instanzen von `Vorgang` erzeugt werden.

C++ **Übung 6.3:** Was gibt das folgende Programm in die Datei `konstrDestr.txt` aus?

```
#include <fstream>
using namespace std;
ofstream dat("konstrDestr.txt", ios::out);
class Z {
public:
  Z(int a=9):w(a) {dat<<"+Z"<<w <<" ";}
  Z(const Z& z):w(z.w)
    {dat<<"+CZ"<<w<< " ";}
  ~Z() {dat << "-Z" << w << " ";}
private:
  int w;
};
void funk1() {
  Z z1(1), z2(2);
}
void funk2() {
  Z z1[3];
}
```

(./ADT/KonstrDestruktor/main.cpp)

```
void funk3() {
  Z* pz1 = new Z(4);
  Z* pz2 = new Z(5);
  delete pz1;
}
void funk4() {
  Z z1(4);
  Z z2(z1);
  Z* pz1 = new Z(z2);
  Z* pz2 = &z1;
  delete pz1;
}
int main() {
  dat << "f1: "; funk1();
  dat << "\nf2: "; funk2();
  dat << "\nf3: "; funk3();
  dat << "\nf4: "; funk4();
  }
```

C++ **Übung 6.4:** Implementieren Sie für die Klasse `A` eine minimale Standardschnittstelle, d.h. die folgenden Methoden und Operatoren:

- Standardkonstruktor,
- Destruktor,
- Kopierkonstruktor,
- Zuweisungsoperator und
- Vergleichsoperator.

```
class A {
public:
  A(const char* nam="");
  ~A();
  A(const A& a);
  const A& operator= (const A& a);
  void print(ostream& str) const;
private:
  char* name;
  int len1;
  friend bool operator==
    (const A& a1, const A& a2);
  };
```

(./ADT/NiceClass/A.h)

Vergessen Sie nicht, geeignete Tests zu implementieren.

Übung 6.5: Implementieren Sie die beiden Methoden `inIdDateiVorhanden` und `ausgabe` der Klasse `LogFunc`. **C++**

Zur Lösung der Aufgabe werden Sie die Klasse `LogFunc` um einige Attribute erweitern müssen. Eine einfachere Implementierung wird sich mit den in Abschn. 6.2.3 vorgestellten klassenglobalen Attributen und Methoden ergeben.

Übung 6.6: Implementieren Sie die beiden Makros `LOG_METHOD` und `LOGTRACE_INIT` der *LogTrace*-Bibliothek. **C++**

Übung 6.7: Implementieren Sie die beiden Makros `LOG_CONSTRUCTOR` und `LOG_DESTRUCTOR` der *LogTrace*-Bibliothek. **C++**

6.3 Generische Programmierung, 2. Teil

In Abschn. 2.3 haben wir die generische Programmierung am Beispiel von Schablonenfunktionen vorgestellt. Ganz entsprechend ist es möglich, sowohl in Java als auch in C++ generische Datentypen/Klassen zu implementieren.

6.3.1 Generische Datentypen

Wir zeigen die Implementierung eines generischen Datentyps am Beispiel eines Stapels (engl. *Stack*); siehe Listing 6.17.

Die Algorithmen des Datentyps `Stapel` brauchen unter Verwendung von Schablonenklassen somit nur einmal unabhängig vom tatsächlich zu speichernden Datentyp implementiert zu werden. Wie bei den typparametrisierten Funktionen wird wiederum anstelle eines konkreten Datentyps, z.B. `int`, ein Platzhalter, z.B. `T`, verwendet. Die Datenstruktur wird dann unter Verwendung des Platzhalter-Typs formuliert.

Beim Erzeugen einer Instanz der Klasse `Stapel` müssen wir nun genau festlegen, welche Typen im `Stapel` gespeichert werden sollen. Dieses wird in spitzen Klammern hinter dem Klassennamen angegeben; siehe Listing 6.18.

Für Java gilt, dass die aktuellen Parameter, die für den Platzhalter `T` eingesetzt werden, Referenztypen sein müssen, d.h. `int`, `double` etc. sind nicht möglich. Wir können diese Standardtypen jedoch zuvor in Referenztypen umwandeln, indem wir auf die Hüllklassen (*Wrapper*) `Integer`, `Double` etc. zurückgreifen, die wir im Abschn. 4.1.6 vorgestellt haben.

C++/Java 6.17: Schnittstelle der generischen Klasse `Stapel`

```
template <typename T>
class Stapel{
private:
  T* data;
  int top, size;
public:
   Stapel (int si) {
     data = new T [si];
     size = si;
     top = 0;
   }
   ~ Stapel () {delete[] data;}
   void push  (T x) {
        data[top++]=x;
     }
   T   pop   () {return data[--top];}
   bool isEmpty () {return top==0;}
};
```

(Poly/Stack/Stapel.h)

```
public class Stapel<T> {
  T[] data;
  private int top, size;

  public Stapel(int s, T x) {
    data = (T[]) new Object[s];
    size = s; top = 0;
  }
  public void push(T x) {
    data[top++] = x;
  }
  public T pop() {
    return data[--top];
  }
  public boolean isEmpty () {
    return top == 0;
  }
}
```

(Poly/Stack/Stapel.java)

C++/Java 6.18: Anwendungsprogramm zur generischen Klasse `Stapel`

```
#include<iostream>
using namespace std;
#include"Stapel.h"

int main() {
  Stapel<double> s1(3);
  Stapel<int>* ps2 = new Stapel<int>(4);
  /*Stapel<Vorgang> s3(9); */

  for (int i=0; i<3; i++) {
    s1.push(i*2.5);
  }
  while ( ! s1.isEmpty() ) {
    cout << s1.pop() << " ";
  }
    // jetzt Syntaxfehler
    // string n="AB"; s1.push(n);
  /* . . . */
  delete ps2; //explizite Freigabe noetig
}
```

(Poly/Stack/DemoTemp.cpp)

```
public class DemoTemp {
  public static void main(String[] args) {
   Stapel<Double> s1=new Stapel<Double>(3);
   Stapel<Integer> s2=new Stapel<Integer>(4);
   /*Stapel<Vorgang> s3 =
               new Stapel<Vorgang>(9);*/

    for (int i=0; i<3; i++) {
      Double d = i * 2.5; s1.push(d);
    }

    while ( ! s1.isEmpty() ) {
      System.out.print(s1.pop()+" ");
    }
    // jetzt Syntaxfehler
    // String n="AB"; s1.push(n);
    /* . . . */
    // keine expliziten Freigaben nötig
  }
}
```

(Poly/Stack/DemoTemp.java)

Die Implementierung der Schablonenmethoden erfolgt in C++ häufig direkt in der Header-Datei oder durch Einbinden der Quellcode-Datei per `#include`-Anweisung in die Header-Datei. Erfolgt die Definition der C++-Methoden nicht direkt in der Klasse, muss vor jeder Schablonenmethode wieder der Ausdruck `template <typename T>` stehen, um anzuzeigen, dass hier die Implementierung einer Schablonenmethode mit dem Platzhalter `T` erfolgt.

```
template <typename T>
  void Stapel<T>::init(int si) { data = new T [si]; size = si; top = 0; }
template <typename T>
  void Stapel<T>::push(T x) { data[top++]=x; }
template <typename T>
  T Stapel<T>::pop() {return data[--top];}
/* ... */
```

6.3.2 Implementierung der generischen Datentypen

Auf den ersten Blick mögen die C++- und Java-Implementierungen für generische Datentypen noch fast gleich aussehen. Es gibt aber gravierende Unterschiede im Detail.

Grundsätzlich gibt es zwei Realisierungsmöglichkeiten von generischen Datentypen:

- Heterogene Variante: Für jeden `Stapel`-Typ (etwa `String`, `Vorgang`, `Netz`) wird individueller Code erzeugt, d.h. Code für drei Klassen.
- Homogene Übersetzung: Für jede parametrisierte Klasse `Stapel<T>` wird nur eine gemeinsame Klasse erzeugt, die statt des generischen Typs einen speziellen Typ, z.B. `Object`, einsetzt. Für einen konkreten Typ werden Typanpassungen (type cast) in die Anweisungen eingebaut.

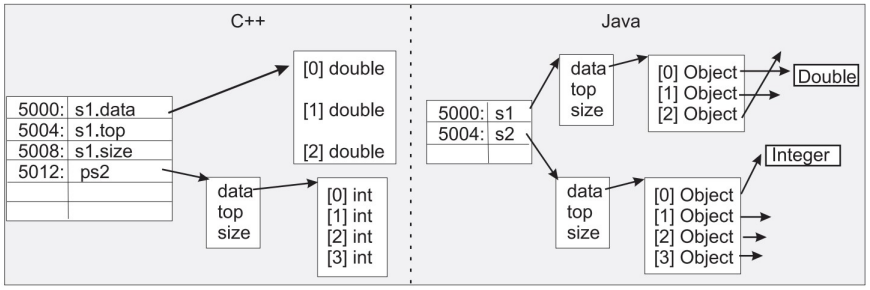

Bild 6.2: Speicherbelegung einer generischen Klasse auf dem Stack und Heap

Java nutzt die homogene Übersetzung, C++ die heterogene. Für unser Beispielprogramm aus Listing 6.18 wird dieser Sachverhalt im Bild 6.2 veranschaulicht.

Der Grund für die unterschiedlichen Ansätze liegt wieder in der Tatsache, dass in C++ Objekte sowohl auf dem Programm-Stack (`s1`) als auch auf dem Programm-Heap (`ps2`) angelegt werden können. Der Speicherplatz, den der `Stapel` belegt, hängt in C++ vom Typ des Schablonenparameters `T` ab. Sowohl der Aufruf `new Stapel<T*>(4)` als auch `new Stapel<T>(4)` erzeugen Objekte vom Typ `Stapel` mit den drei Attributen `data`, `top` und `size`. Allerdings wird im Fall von `<T*>` Speicherplatz für Referenzen auf `T`-Objekte erzeugt, während im Fall `<T>` direkt Speicherplatz für die `T`-Werte angelegt wird. Der Aufruf der Methode `push` überschreibt dann im ersten Fall die Zeiger, im zweiten Fall die Werte direkt. Je nach der Größe von `T` belegen `Stapel<T>` und `Stapel<T*>` unterschiedlich viel Speicher.

Da in Java für den generischen Datentyp `T` immer nur Referenzen zum Einsatz kommen, beschränken sich die Unterschiede zwischen zwei Stapeln von unterschiedlichem Typ `T`, etwa `Stapel<Integer>` und `Stapel<Double>`, nur auf den Typ der Referenzen: Das Array `data` enthält einmal `Integer`- und einmal `Double`-Referenzen, aber doch in beiden Fällen eben nur Referenzen. Aus Sicht des Speicherplatzbedarfs ist es gleichgültig, ob wir Referenzen auf `Integer`, `Double` oder auf ein beliebiges anderes Objekt speichern. Da sich die verschiedenen Stapel-Klassen also kaum unterscheiden, ist es wenig sinnvoll, hier jedes Mal eine neue Klasse zu erzeugen.

Als wichtiger Vorteil der generischen Datentypen verbleibt (in Java nur) ein Zugewinn an Typsicherheit. Bei einem generischen Stapel `s` für beliebige Objekte (Java: `Object`) oder Referenzen (C++: `void*`) würden wir `s.pop()` auf den erwarteten Typ `T` bzw. `T*` zur Laufzeit casten müssen. Dieser Type-Cast führt zu einem Laufzeit-Fehler, wenn wir an irgendeiner Stelle zuvor versehentlich eine Variable eines anderen Typs auf den Stapel geschoben haben. Durch die Verwendung der typsicheren Variante `Stapel<T>` kann uns der Compiler noch vor der Programmausführung auf diese unsachgemäße Verwendung hinweisen. Wir gehen hierauf noch näher im Zusammenhang mit dem dynamischen Binden im Abschn. 7.3.6 ein. Dort ist die Prüfung erst zur Laufzeit möglich.

6.4 Ausnahmebehandlung, 2. Teil

Im Abschn. **??** haben wir Ausnahmen als eine fortgeschrittene Programmiertechnik zur Fehlerbehandlung in Anwendungen kennengelernt. Diese Thematik werden wir nun insbesondere im Zusammenhang mit Klassen und der Freigabe von Instanzen auf dem Programm-Stack und -Heap vertiefen (Abschn. 6.4.1).

Stack Unwinding: Beim *normalen* Verlassen einer Funktion, z.B. über eine `return`-Anweisung, werden automatisch die Variablen freigegeben, die auf dem Stack verwaltet werden (alle lokalen Variablen). Das Gleiche geschieht auch beim Verlassen einer Funktion über eine `throw`-Anweisung, d.h. **der Programm-Stack wird abgebaut** (engl.: *stack unwinding*). Handelt es sich bei den Variablen um Instanzen von Klassen (nur in C++ möglich, weil in Java nur Referenzen auf die Objekte im Stack stehen), wird deren Destruktor automatisch aufgerufen.

In der Funktion `f` wird im folgenden Codeausschnitt ein Objekt `a` der Klasse `A` vereinbart. Sowohl beim *normalen* Ausstieg über `return` als auch beim *Notausstieg* über `throw` wird der zugehörige Destruktor aufgerufen, bzw. der Java-Garbage Collector gibt den Speicher wieder frei.

```
double f(double z) {
  A a;
  . . .
  if (z < 0.0) throw X(z);
  . . .
  return sqrt(z);
}

void call_f() {
  while (1){
    try { ... y = g(x); ... }
    catch (X& x) { . . . }
  }
}
```

```
double f(double z) {
  A a = new A();
  . . .
  if (z < 0.0) throw X(z);
  . . .
  return sqrt(z);
}

void call_f() {
  while (1){
    try { ... y = g(x); ... }
    catch (X x) { . . . }
  }
}
```

In anderen Fällen sind wir oft selbst für die Freigabe von Ressourcen, die wir nur innerhalb der Funktion benötigen, verantwortlich. Üblicherweise befinden sich solche *Aufräumarbeiten* am Ende der Funktion. Wird die Funktion über eine `throw`-Anweisung frühzeitig verlassen, wird der Aufräumcode nie ausgeführt, wodurch Ressourcenlecks entstehen können. Um die Problematik von Lecks und inkonsistenten Zuständen kümmern wir uns in Abschn. 6.4.2 und 6.4.3.

6.4.1 Ausnahmen weiterleiten

Fehler oder Ausnahme? Wann werfen wir eine Ausnahme? Zumindest eines ist sicher: Ausnahmen zur Laufzeit sind ein teures Instrument. Wenn wir mit `a[i]` auf ein Array-Element mit ungültigem Index `i` zugreifen, wird in Java eine Ausnahme geworfen. Das Erzeugen und Abfangen der Ausnahme ist etwa tausend Mal teurer als die explizite Kontrolle durch eine Anweisung der Art `if (i<a.length) a[i]=...`. Wenn Fehler selten sind, dann schlägt der hohe Aufwand kaum zu Buche. Für vermeidbare Situationen sollten Ausnahmemechanismen jedoch nicht missbraucht werden.

Lohnt es sich, auf alle möglichen Ausnahmen zu reagieren? Vielfach nicht, denn wie sollte eine geeignete Reaktion auf eine Bereichsüberschreitung bei `a[i]` aussehen? Hier handelt es sich offenbar um eine Situation, die der Programmierer nicht berücksichtigt oder falsch umgesetzt hat. Wir können die Ausführung nicht mehr retten, die Ausnahme allenfalls nutzen, um einen Fehlerreport anzustoßen. Anders sieht es aus, wenn z.B. Datei-Ein/Ausgabe im Spiel ist. Dann kann die vom Benutzer angegebene Datei nicht gefunden werden, die Festplatte voll sein etc. Im Gegensatz zur Situation vorher liegt kein Programmierfehler (ohne Chance auf Korrektur) vor, sondern die Situation kann durch Umbenennung oder Löschen von Dateien gerettet werden. Hier ist eine echte Ausnahmebehandlung möglich. Java unterscheidet infolgedessen zwei Arten von Ausnahmen. Ausnahmen ohne Chance auf Fortsetzung des Programms (z.B. bei Programmierfehlern) werden *RuntimeException* genannt; sie können nicht behandelt werden und müssen daher auch nicht über `catch`-Klauseln gefangen werden. *Normale* Ausnahmen (*Exception*) benutzt man hingegen als Instrument, um den Entwickler auf vorhersehbare Fehlersituationen hinzuweisen. Ihre Behandlung erzwingt Java, indem der Compiler das Fehlen einer entsprechenden

Java

catch-Klausel bemängelt. Der Java-Entwickler hat dann zwei Möglichkeiten: Entweder er entschließt sich für die Behandlung der Ausnahme und fügt einen catch-Block ein, oder er *leitet die Ausnahme weiter* an den Aufrufer, damit er sich darum kümmert. Diese Weiterleitung muss im Funktionskopf angezeigt werden, damit der Aufrufer darauf hingewiesen wird, welche Probleme er (und weitere Aufrufer) behandeln müssen (siehe Funktion h im folgenden Listing 6.19).

C++/Java 6.19: Fehlerbehandlung, Aufräumen des Programm-Stacks

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

class A {
  public:
    A() {cout << "+A "; }
    ~A() {cout << "-A "; }
};

double f(double z) {
  A a;
  if (z < 0.0) throw double(z);
  return sqrt(z);
}
double g(double z) {
  A a;
  return f(z);
}
double h(double z) {
  A a;
  try { g(z); }
  catch (double z) {
     if (z<0.0) throw "negativ";
     else throw "???";
     }
  return f(z);
}

int main() {
   try {
      double y = h(25);
      cout << endl << "h(25) = "<<y<<endl;

      y = h(-25);
      cout << endl << "h(-25) = "<<y<<endl;
      }
   catch (char* string) {
      cout << endl << string << endl;
      }
   cout << "Programmende" << endl;return 0;
}
```

(Poly/exception/e1.cpp)

```java
class A {
  public A() {System.out.println("+A ");}
};

public class E1 {
  public static double f(double z)
  {
  A a = new A();
  if (z < 0.0) throw new DThrow(z);
  return Math.sqrt(z);
  }
public static double g(double z)
  {
  A a = new A();
  return f(z);
  }
public static double h(double z)
              throws StrThrow {
  A a = new A();
  try { g(z); }
  catch (DThrow dt) {
     if (dt.val<0.0)
     throw new StrThrow("negativ");
     else throw new StrThrow("???");
     }
  return f(z);
  }

public static void main(String args[])
  {
   try {
      double y = h(25);
      System.out.println("h(25) = " + y);
      y = h(-25);
      System.out.println("h(-25)=" +y+"\n");
      }
   catch (StrThrow str) {
      System.out.println("\n" + str.val);
      }
  System.out.println("Programmende");
  }
}
```

(Poly/exception/E1.java)

Das Listing 6.19 demonstriert das Aufräumen des Programm-Stacks und das Überspringen von Ebenen an einem Beispiel. Die Java-Ausnahmeobjektklassen DThrow

und `StrThrow` sind als *RuntimeException* bzw. *Exception* entsprechend dem folgenden Listing implementiert:

```
class DThrow extends RuntimeException{
  public double val;
  public DThrow(double v) {val = v;}
  };
```

```
class StrThrow extends Exception{
  public String val;
  public StrThrow(String v) {val = v;}
  };
```

In den drei Funktionen `f`, `g` und `h` wird jeweils ein Objekt der Klasse `A` angelegt, dessen C++-Konstruktor bzw. -Destruktor die Ausgabe `+A` bzw. `-A` erzeugen. Da es sich in Java bei `DThrow` um eine *RuntimeException* handelt, muss die Tatsache, dass die Funktionen `f` und `g` eine solche Ausnahme werfen, nicht explizit im Funktionskopf angegeben werden. In der Funktion `h` wird diese Ausnahme gefangen und als eine Ausnahme des Typs `StrThrow` neu geworfen. Da `StrThrow` eine normale *Exception* darstellt, die in `h` aber nicht behandelt wird, muss die Funktion im Kopf den Zusatz `throws StrThrow` tragen. Im Hauptprogramm `main` würde der Compiler das Fehlen des `catch (StrThrow str)`-Blocks bemängeln.

Die Ausgabe des Programms demonstriert im Fall von C++, dass auch bei der Eingabe eines fehlerhaften Wertes, die dann zur Erzeugung eines Fehlerobjekts beim Aufruf in der Funktion `f` führt, in jeder der drei Funktionen der Destruktor aufgerufen wird:

```
+A +A +A -A -A +A -A -A
h(25) = 5
+A +A +A -A -A -A
negativ
Programmende
```

Im entsprechenden Java-Programm werden keine Destruktoren aufgerufen, sondern es ist Aufgabe des Garbage Collectors, den nicht mehr benötigten Speicher auf dem Heap irgendwann freizugeben.

Die Ausgaben sind entsprechend:

```
+A +A +A +A
h(25) = 5
+A +A +A
negativ
Programmende
```

Bild 6.3 stellt den Kontrollfluss des C++-Programms aus Listing 6.19 grafisch dar, und zwar im linken Teil für den Fall, dass kein Ausnahmeobjekt erzeugt wird, im rechten Teil für den Fall, dass die Funktion `f` mit einem negativen Argument aufgerufen wird und dass sie infolgedessen ein Ausnahmeobjekt generiert. In Java kann für die Beispielanwendung in Listing 6.19 nicht eindeutig angegeben werden, wie viele Instanzen von `A` jeweils vorliegen, weil nicht bekannt ist, wann der Garbage Collector jeweils den nicht mehr benötigten Speicher auf dem Heap freigibt.

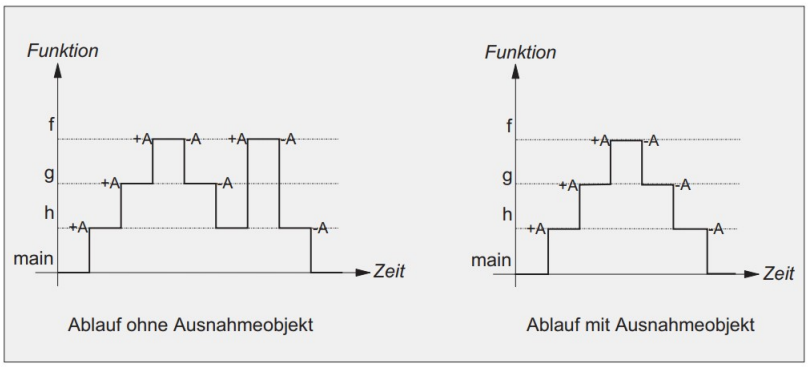

Bild 6.3: Ablauf ohne und mit Erzeugen eines Ausnahmeobjektes

6.4.2 Verhinderung von Ressourcen-Lecks

Ressourcen wie Speicher, Datei-Handler, Datenbank- oder Netzwerk-Verbindungen sind nur begrenzt verfügbar und damit nach Gebrauch (für andere Anwendungen) wieder freizugeben (für Speicheranforderungen siehe Abschn. 4.1). Eine solche Situation sehen wir auf der linken Seite im Listing 6.20.

Java 6.20: Betriebsmittelverwaltung mit und ohne `finally`

```
int f(...) {
  try {
    Ressource A beanspruchen
    Ressource B beanspruchen
    Operation C
    Ressource B freigeben
    Ressource A freigeben
  }
  catch (AusnahmeSituationX e) {
    throw new OperationGescheitert("X");
  }
  catch (AusnahmeSituationY e) {
    return 0;//bei Fehler Y ist 0 das Ergebnis
  }
  Weitere Operationen D
  return ergebnis;
}
```

(ADT/finally.sht)

```
int f(...) {
  try {
    Ressource A beanspruchen
    Ressource B beanspruchen
    Operation C
  }
  catch (AusnahmeSituationX e) {
    throw new OperationGescheitert("X");
  }
  catch (AusnahmeSituationY e) {
    return 0;// 0 muss dann Ergebnis sein
  }
  finally {
    Ressource B freigeben
    Ressource A freigeben
  }
  Weitere Operationen D
  return ergebnis;
}
```

Ressourcen A und B werden nach Gebrauch durch Operation C wieder freigegeben. Da in Operation C zwei verschiedene Ausnahmesituationen X und Y auftreten können, ist der Quelltext mit `try/catch` umschlossen. Dabei ergibt sich folgende Problematik: Die Ausführung des `try`-Blocks wird im Ausnahmefall abgebrochen, d.h. der Programmfluss erreicht den Code zur Ressourcen-Freigabe gar nicht erst. Um sicherzustellen, dass die Ressourcen A und B stets freigegeben werden, muss

die Freigabe drei Mal codiert werden: einmal wie im linken Teil in Listing 6.20 für den fehlerfreien Durchlauf sowie einmal für jede Ausnahmesituation (Kopie der beiden Freigaben am Anfang jedes `catch`-Blocks einfügen). Da diese Code-Duplikation etwas lästig und fehleranfällig ist, bietet Java hier das Schlüsselwort **finally**, das hinter dem Ende des letzten `catch`-Blocks angefügt werden kann. Die Ausführung dieses Blocks erfolgt garantiert, d.h. im Fehlerfall wie im Erfolgsfall und auch beim Weiterleiten einer Ausnahme (`throw new OperationGescheitert(..)`) oder beim Verlassen des Blocks durch `return`, `break` oder `continue`.

Die STL-Klasse `auto_ptr`

C++

In C++ gibt es das Schlüsselwort `finally` oder etwas Vergleichbares nicht. Man kann sich jedoch bei der Speicherfreigabe mit intelligenten Zeigern behelfen.

Im C++-Standardbibliotheks-Modul `memory` gibt es eine Schablonenklasse `auto_ptr`, die im folgenden Beispiel einen *intelligenten* Zeiger (engl. *smart pointer*) auf ein `ifstream`-Objekt erzeugt, das beim Verlassen der Funktion durch Aufruf seines Destruktors wieder an den Programm-Heap zurückgegeben wird. Entsprechende Zeiger werden auch **Auto-Zeiger** oder **auto pointer** genannt.

```
#include<memory>
using namespace std;
 . . .
 void f(...) {
  auto_ptr<ifstream> input = new ifstream("xxxx");

  . . .
  if (...) throw Error();
  . . .
 }
```

Auto-Zeiger sind selbst definierte Zeiger, die wie „normale“ Zeiger verwendet werden können, aber mit zusätzlichen Fähigkeiten ausgestattet sind. Einen Ausschnitt aus einer möglichen Implementierung aus der C++-Standardbibliothek mit einer kleinen Anwendung zeigt das folgende Listing:

```
template<class T> class auto_ptr {
   T* ptr;
  public:
   auto_ptr (T* p=0) : ptr(p) {}
   ~ auto_ptr ()  { delete ptr; }
};
void f() {
 auto_ptr <Vorgang> pv = new Vorgang; // Aufruf des Konstruktors
 . . .
}                                     // Aufruf des Destruktors
```

In diesem Beispiel, das nur das Prinzip zeigt, wird eine Instanz `px` vom Typ `auto_ptr` erzeugt und über den Konstruktor so initialisiert, dass sein interner Zeiger `ptr` auf einen `Vorgang` zeigt. Das Besondere daran ist, dass beim Verlassen des Gültigkeitsbereiches von `px` automatisch sein Destruktor (d.h. der Destruktor der Klasse `auto_ptr`)

aufgerufen und über dessen `delete` der Speicherbereich im Heap freigegeben wird. Damit ist es nicht mehr erforderlich, selbst verwalteten Heap-Speicher explizit mit `delete` wieder freizugeben, sondern der intelligente Zeiger enthält einen kleinen *Garbage Collector*.

Um mit solchen Auto-Zeigern ähnlich wie mit normalen Zeigern arbeiten zu können, stellt die STL-Implementierung von `auto_ptr` den Kopierkonstruktor, den Zuweisungsoperator, den Dereferenzierungsoperator und den für den Komponentenzugriff geeigneten Operator (->) zur Verfügung. Listing 6.21 zeigt eine Anwendung der intelligenten Zeiger.

C++ 6.21: Anwendung der Klasse `auto_ptr`

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

struct X {
  int x;
  X() { cout << "+A "; }
  ~X() { cout << "-A "; }
};
```

```
void f() {
  X *pa      = new X;
  auto_ptr<X> pb(new X);
  auto_ptr<X> pc;

  pa->x = 123; cout << pa->x << " ";
  pb->x = 321; cout << pb->x << " ";
  pc = pb;  cout << pc->x << " ";
}
int main() { f(); return 0;}
```

(Poly/AutoPtr/demo1.cpp)

An der Ausgabe des Programms

```
+A +A 123 321 321 -A
```

erkennt man, dass der Destruktor des an den intelligenten Zeiger `pb` gebundenen Objektes aufgerufen wird, der Destruktor des an den normalen Zeiger `pa` gebundenen Objektes hingegen nicht.

Die Implementierung von Kopierkonstruktor und Zuweisungsoperator erfolgt so, dass jeweils der *Zielzeiger* an das betreffende Objekt gebunden wird und der *Quellenzeiger* den Wert `NULL` erhält. Damit geht das Objekt in den Besitz des neuen intelligenten Zeigers über, denn es darf nur einen Besitzer haben, der dann auch für die Freigabe des Objektes zuständig ist.

Leider verursacht dieses Verhalten des Kopierkonstruktors eine unangenehme Nebenwirkung:

```
void f1(auto_ptr<X> p) {}
void f2(auto_ptr<X>& p) {}
. . .
auto_ptr<X> pa(new X);
auto_ptr<X> pb(new X);
f1(pa);
f2(pb);
cout << pa->x;       // Fehler: pa == NULL !!!
cout << pb->x;       // korrekt
```

Wenn ein Auto-Zeiger per Wert an eine Funktion übergeben wird, erfolgt die Übergabe durch den Aufruf des Kopierkonstruktors, und das bewirkt, dass der übergebene Zeiger vom Objekt getrennt (d.h. auf den Wert `NULL` gesetzt) wird! Daraus folgt ganz allgemein:

- `auto_ptr`-Objekte sollten per Referenz übergeben werden! **Tipp**
- Entsprechend werden `auto_ptr`-Objekte als Werte aus Funktionen zurückgegeben.

Sind andere Betriebsmittel als Heap-Speicherbereiche zu verwalten, kann entsprechend verfahren werden: Es wird eine separate Klasse für die Betriebsmittelverwaltung erstellt. Der Konstruktor der Klasse belegt das Betriebsmittel, und der Destruktor gibt es frei. Der Destruktor wird vor jedem Verlassen der Funktion aufgerufen, unabhängig davon, ob aufgrund einer normalen `return`-Anweisung oder aufgrund einer Ausnahme. Wir haben diese Technik bereits bei der Implementierung der *Log-Trace*-Funktionalität (Abschn. 6.2.5) angewendet.

Eine Alternative zur Klasse `auto_ptr` sind referenzzählende, intelligente Zeiger, d.h. Zeiger, die Buch darüber führen, wie viele Objekte auf eine bestimmte Ressource verweisen und die automatisch die Ressource löschen, wenn kein Objekt mehr auf die Ressource verweist. In der in TR1 (Technical Report 1) beschriebenen Erweiterung von C++ stellt die Klasse `tr1::shared_ptr` diese Funktionalität zur Verfügung.

```
{
tr1::shared_ptr<X> p1(new X); // Objekt X1
tr1::shared_ptr<X> p2;
p2 = p1; // 2 Zeiger zeigen auf X1
tr1::shared_ptr<X> p3(new X); // Objekt X3
p3 = p2; // X3 wird wieder freigegeben; 3 Referenzen auf X1
}
// X1 kann nun auch freigegeben werden.
```

Im Unterschied zum eingebauten Garbage Collector unterstützt die Klasse `tr1::shared_ptr` aber keine zyklischen Strukturen. Bei referenzzählenden, intelligenten Zeigern ist es somit möglich, dass die Referenzzähler einiger Objekte noch größer Null sind, obwohl die Ressourcen von außen schon nicht mehr erreichbar sind. Hier bietet TR1 mit der Klasse `tr1::weak_ptr` nun ebenfalls Unterstützung an, siehe z.B. [39] für weitere Details.

6.4.3 Ausnahmesicherer Code

Wir haben bereits gesehen, dass nicht in jedem Fall ein Fehler sinnvoll behandelt werden kann. Wenn dem Erzeugen einer Ausnahme doch nur der Programmabbruch folgt, dann helfen Ausnahmen nicht unbedingt weiter. Die Alternative sind *robuste Programme*, die fehlerfrei arbeiten, solange sie laufen, und beim Auftreten eines Fehlers mit einem entsprechenden Hinweis abbrechen.

Beim Entwurf von Bibliotheksmodulen oder anderen wiederverwendbaren Software-Bausteinen ist die spätere Verwendung breit gefächert. Hier kann es sinnvoll sein, alle Fehlermöglichkeiten durch die Generierung von Fehlerobjekten abzudecken, denn das Erzeugen von Ausnahmeobjekten ist viel einfacher als eine detaillierte Ausnahmebehandlung.

Bei der Verwendung von Modulen, die mit der Erzeugung von Ausnahmeobjekten reichlich ausgestattet sind, kann im einfachsten Fall auf eine Behandlung verzichtet werden; dann wird das Programm im Fehlerfalle beendet.

Die nächste Ebene kann eine Fehlerdiagnose auf der obersten Programmebene sein, damit lässt sich in etwa das Gleiche erreichen wie bei der Verwendung von `assert`-Makros (siehe Abschn. 5.1.1).

Die von C++ und Java bereitgestellten Sprachunterstützungen für die Ausnahmebehandlung sind kompakt und universell einsetzbar, beinhalten aber zumindest in C++ kein konsistentes Anwendungskonzept. Während in Java alle Bibliotheken konsistent Ausnahmen nutzen, verwenden einige zum C++-Standard gehörende Bibliotheken das Konzept der Ausnahmebehandlung sehr intensiv (Beispiel: Strings), andere wieder gar nicht oder wenig (Beispiel: Standard Template Library).

Es sollten zumindest die folgenden Regeln beim Werfen und Weiterleiten von Ausnahmeobjekten beachtet werden:

Tipp

- Ressourcen-Lecks müssen vermieden werden.
- Datenstrukturen dürfen durch eine Ausnahme nicht in einem inkonsistenten Zustand zurückgelassen werden.

Mit Möglichkeiten zur Verhinderung von Ressourcen-Lecks haben wir uns insbesondere für C++ im Abschn. 6.4.2 ausführlich befasst. Die Verhinderung eines inkonsistenten Zustands soll das folgende Beispiel illustrieren.

```
class Netz {
  Vorgang* ersterVorgang;
  string nameErster;
  /* ...*/
  };

void Netz::neuerErster(string n) {
  delete ersterVorgang;
  nameErster = n;
  ersterVorgang = new Vorgang(n,...);
  /* ...*/
  }
```

```
class Netz {
  Vorgang ersterVorgang;
  String nameErster;
  /* ...*/


void neuerErster(String n) {
  nameErster = n;
  ersterVorgang = new Vorgang(n,...);
  /* ...*/
 }
};
```

Wenn in dem Beispiel beim Aufruf von `new` oder im anschließend aufgerufenen Konstruktor eine Ausnahme geworfen wird, liegt das `Netz` in einem inkonsistenten Zustand vor. In C++ verweist das Attribut `nameErster` auf einen nicht mehr existierenden `Vorgang`. Außerdem wurde der Name bereits gesetzt, obwohl der Zeiger auf einen `Vorgang` noch gar nicht erfolgreich angepasst wurde.

Ausnahmesichere Funktionen sollten daher genau eine der folgenden Garantien bieten:

- In der Funktion wird direkt oder indirekt keine Ausnahme geworfen (sog. *nothrow-Garantie).*
- Beim Erzeugen oder Weitergeben eines Ausnahmeobjektes muss dafür gesorgt werden, das betroffene Objekt in dem Zustand zu verlassen, der beim Einstieg in die Funktion (Ausgangszustand) vorlag (sog. *starke Garantie).*
- Wenn trotz Werfens einer Ausnahme die *starke Garantie* nicht möglich oder nicht gewollt ist, sollte das Objekt zumindest in einem konsistenten Zustand verlassen

werden, d.h. in einem Zustand, der einem definierten Wert des Objektes entspricht (sog. *Basisgarantie)*.

In unserem Beispiel können wir durch einfaches Umstellen der Anweisungen sogar die starke Garantie erfüllen:

```
void Netz::neuerErster(string n) {
  Vorgang* tmp = new Vorgang(n,...);
  Vorgang* alt = ersterVorgang;
  ersterVorgang = tmp;
  nameErster = n;
  delete alt;
  /* ...*/
  }
```

```
void neuerErster(String n) {
  Vorgang tmp = new Vorgang(n,...);

  ersterVorgang = tmp;
  nameErster = n;
  // Garbage Collector räumt auf
  /* ...*/
 }
```

Allgemein spricht man hier von der *Kopiere-Und-Tausche-Strategie* (engl. *copy-and-swap*). Es wird in der Funktion, die ausfallsicher implementiert werden soll, zunächst eine Kopie der zu verändernden Objekte erstellt. Die Kopien werden dann manipuliert. Erst wenn das erfolgreich war, werden die Kopien gegen die Originale ausgetauscht (swap). Allerdings ist die Strategie nicht unbedingt sehr effizient, da immer Kopien erstellt werden müssen, die im Erfolgsfall überflüssig sind.

Aber auch die *copy-and-swap*-Stategie kann im allgemeinen Fall keine Ausnahmesicherheit garantieren, wie das folgende Beispiel zeigt:

```
void funktion() {
    ...          // Kopie des lokalen Zustands erstellen
   f1();
   f2();
    ...          // swap durchführen
}
```

Hier bestimmt wieder einmal der langsamste Waggon die Geschwindigkeit des gesamten Zuges. Wenn `f1` oder `f2` keine starke Garantie für Ausnahmesicherheit bieten können, muss `funktion` vor Ausführung von `f1` zunächst eine Kopie aller Variablen erstellen, die in `f1` direkt oder indirekt verändert werden können (entsprechend für `f2`). Dies ist im allgemeinen Fall kaum möglich, da dann `funktion` alle aufgerufenen Funktionen nachbilden müsste. Aber selbst wenn sowohl `f1` als auch `f2` eine starke Garantie bieten sollten, ist damit immer noch nichts gewonnen. Nehmen wir an, `f1` wird ohne Ausnahme ausgeführt und bei Ausführung von `f2` wird eine Ausnahme geworfen. Dann ist der Zustand durch Ausführung von `f1` nun nicht mehr der gleiche wie zu Beginn der Funktion, selbst dann nicht, wenn `f2` überhaupt keine Änderungen durchgeführt hätte.

Daher wird man sich häufig damit begnügen müssen, dass Funktionen nur die Basisgarantie bieten können, d.h. sie hinterlassen die Objekte zumindest in konsistenten Zuständen. Welche nun genau vorliegen, muss der Aufrufer selbst bestimmen. Ein inkonsistenter Zustand eines `Stapel`-Objektes liegt z.B. vor, wenn in der Funktion `push` der Zähler erhöht, aber kein entsprechendes neues Element abgelegt wurde.

6.5 Zusammenfassung

Wir haben in diesem Kapitel gesehen, wie wir das Konzept der abstrakten Datentypen umsetzen können – und zwar *von Hand* in einer prozeduralen Sprache (C) und in einer objektorientierten Sprache (C++ oder Java). Beim Übergang in die objektorientierten Sprachen hat unsere prozedurale Version folgende Änderungen erfahren:

- Statt `struct Vorgang` benutzen wir nun `class Vorgang` und können die Sichtbarkeit von Methoden und Attributen in drei Stufen steuern.
- Weil wir einen abstrakten Datentyp anstreben, sind nur Methoden der Schnittstelle öffentlich, alle Attribute sind privat.
- Das erste Argument unserer Schnittstellenfunktionen wird jeweils vom Compiler *eingefügt* und taucht in der Deklaration nicht mehr auf. Im Aufruf der Methoden wandert dieses erste Argument der Funktion *vor* die Methode. Das Argument ist aber weiterhin unter dem Namen `this` vorhanden.
- An die Stelle der `erzeuge...()`-Funktionen treten die Konstruktoren.
- Globale Variablen und Funktionen werden als statische Klassenattribute und -methoden umgesetzt. In unserer Netzplanung ist die Konstante `MAX` ein Beispiel für eine globale Variable, die sich als statisches Attribut der Netz-Klasse zuordnen lässt.

Die ersten Errungenschaften von Klassen haben bereits Auswirkungen auf unser Wissen zu Ausnahmen gehabt. Da Klassen als abstrakte Datentypen für die Konsistenz der verwalteten Daten zuständig sind, müssen wir bei Ausnahmen besondere Sorgfalt walten lassen, um die Objekte nicht unbedacht für jede weitere Verwendung unbrauchbar zu hinterlassen. Dabei sollte eine von drei Arten der Ausfallsicherheit vorliegen: Entweder es wird überhaupt kein Ausnahmeobjekt geworfen (nothrow-Garantie), oder die Objekte werden so hinterlassen, als ob die Methode gar nicht aufgerufen worden wäre (starke Garantie), oder die Objekte werden zumindest in einem konsistenten Zustand hinterlassen (Basis-Garantie).

Kapitel 7

Vielgestaltigkeit (Polymorphie)

Nach der Einführung abstrakter Datentypen in Form von Klassen im letzten Kapitel wird uns in diesem Kapitel die Möglichkeit beschäftigen, dieselbe Schnittstelle auf unterschiedliche Weise zu implementieren. Am Ende wird es uns gelingen, unterschiedliche Implementierungen parallel zu benutzen, ohne dass der Anwender von dieser Vielfalt (Vielgestaltigkeit) etwas mitbekommt. Wie schon im letzten Kapitel werden wir uns zunächst damit beschäftigen, wie wir dieses Ziel mit prozeduralen Sprachmitteln erreichen können. Damit haben wir gleichzeitig ein Modell für die Abläufe in objektorientierten Sprachen, wodurch deren Verständnis erleichtert wird.

7.1 Statische Bindung

Wir haben im letzten Kapitel den Nutzen von Schnittstellen erkannt und als einen Vorteil ihrer Verwendung die erleichterte Austauschbarkeit von verschiedenen Implementierungen der gleichen Schnittstelle erkannt. Ist es denn so wahrscheinlich, dass wir mehrere Implementierungsvarianten vorliegen haben, die wir hinter derselben Schnittstelle *verstecken* können? Oder bezieht sich dieses Szenario eher auf die Wartung und Pflege von Software-Systemen?

Ein Paradebeispiel für die mehrfache Implementierung derselben Konzepte bietet die Welt der Container-Datentypen. Unter Container-Datentypen versteht man Datenstrukturen, die mehrere andere Daten (gleichen Typs) aufnehmen. Diese werden oftmals als Array, verkettete Liste, binärer Suchbaum, AVL-Baum etc. ausgelegt. Jede dieser Datenstrukturen kann dann für den Einsatz in einer konkreten Anwendung noch speziell optimiert werden. Hätte jede dieser Varianten eine andere Schnittstelle, so hätte die Entscheidung des Entwicklers für eine Realisierung große Tragweite für das ganze Projekt, eine nachträgliche Änderung wäre aufwändig. Kapseln wir die unterschiedlichen Datenstrukturen wie Bit-Menge, Liste, Suchbaum, Hash-Tabelle usw. aber unter einer einheitlichen Schnittstelle, etwa dem Konzept der Menge im mathematischen Sinn, kann der Entwickler mit irgendeiner Realisierung des abstrak-

ten Datentyps beginnen. Ändern sich die Anforderungen der Applikation, lässt sich dank vereinheitlichter Schnittstelle die verwendete Datenstruktur austauschen.

Wie geht das praktisch? In einer Header-Datei wird (getrennt von der Realisierung) nur die Schnittstelle hinterlegt, etwa wie im folgenden Listing die Schnittstelle für das Laden und Speichern von Netzplänen unter einer Nummer. In verschiedenen Source-Dateien wird dieselbe Schnittstelle mehrfach implementiert. Nehmen wir an, wir haben eine operative Variante (Quellcode-Datei `loadDB`), in der die Netze aus einer Datenbank gelesen werden, sowie für Debugging-Zwecke eine Test-Variante (Quellcode-Datei `loadFile`), in der die Netze aus einer Datei gelesen werden. Beide Implementierungen beziehen sich auf dieselbe Header-Datei. Alle Varianten werden getrennt compiliert, und erst beim Linken wird entschieden, welche Objekt-Datei an die verbleibenden Dateien gebunden wird. Ein Aufruf von `make appDB` linkt die Objektdateien für den Datenbank-Zugriff, ein Aufruf von `make appFile` linkt die Objektdateien für den Datei-Zugriff.

```
#include "load.h"

int main() {
  Netz *n = erzeugeNetz(0,100);

  /* Laden eines Netzes */
  load(n,0);
  /* Verarbeitung ... */
  plane(n);
  /* Speichern eines Netzes */
  save(n,0);

  return 0;
}
```

```
/**
 * @brief Laden/Speichern von Netzen
 */

#ifndef LOAD_HEADER
#define LOAD_HEADER

#include "Netz.h"

void load(Netz *n,int id);
void save(Netz *n,int id);

#endif
```

Listing 7.1: Makefile

```
NetzObj=../v4-adt/Netzplanung.o
loadDB.o : load.h loadDB.c
	cc -c loadDB.c -o loadDB.o

loadFile.o : load.h loadFile.c
	cc -c loadFile.c -o loadFile.o

main.o : main.c
	cc -c main.c -o main.o
```

```
appDB : main.o $(NetzObj) loadDB.o
	cc main.o $(NetzObj) \
		loadDB.o -o appDB

appFile : main.o $(NetzObj) loadFile.o
	cc main.o $(NetzObj) \
		loadFile.o -o appFile

clean :
	rm *.o *~ appDB appFile
```

(netzplanung/statbind/Makefile.in)

Man spricht in diesem Beispiel von *statischer Bindung*, weil zur Compile-Zeit bestimmt wird, welche Implementierung benutzt wird. Eine gleichzeitige Verwendung *beider* Realisierungen im gleichen Programm an verschiedenen Stellen ist bei dieser Vorgehensweise aber nicht möglich (weil ein Namenskonflikt auftritt). Noch weniger ist es möglich, die gleiche Stelle, d.h. eine bestimmte Funktion im selben Programmdurchlauf, mal mit der einen und mal mit der anderen Realisierung aufzurufen. In

diesem Fall müsste *dynamisch* entschieden werden, welche Funktion jeweils aufzurufen ist. Diesen Zustand zu erreichen, ist das Ziel des folgenden Abschnitts.

7.2 Dynamische Bindung

In Anwendungen kann es durchaus vorkommen, dass in verschiedenen Kontexten verschiedene Implementierungen desselben Datentyps oder Algorithmus vorteilhaft sind. So könnte es bei unserer Netzplanung einen Transportvorgang geben, der durch die Anzahl der Teile, die einfache Transportdauer und die maximale Anzahl von Teilen pro Transport charakterisiert wird. Abhängig von diesen Größen ergibt sich die Dauer des Vorgangs. An anderer Stelle haben wir einen Produktionsvorgang, bei dem ein bestimmtes Produkt in einer bestimmten Menge gefertigt wird. Hier ergibt sich die Vorgangsdauer aus Umrüstzeiten und dem Produkt aus Anzahl und Produktionsdauer für ein Teil. Beides sind Vorgänge im Sinne unseres abstrakten Datentyps `Vorgang`. Dennoch benötigen wir unterschiedliche Implementierungen für Transport- und Produktionsvorgänge, weil andere Attribute gespeichert werden müssen und sich die Vorgangsdauer unterschiedlich errechnet.

Für die Netzplanung ist es völlig unerheblich, um welche Art von Vorgang es sich handelt, es reicht zu wissen, dass es sich um Vorgänge handelt und für sie eine Dauer vorliegt (die über `getDauer` abzufragen ist). Dies ist ein Beispiel, bei dem wir beide Vorgänge gleichzeitig in unserer Anwendung benutzen wollen. Mit statischer Bindung können wir hier nichts erreichen. Wir wollen wieder unsere prozedurale Version der Netzplanung um diesen Fall erweitern, um zu verstehen, wie eine Lösung aussehen kann – und damit auch zu verstehen, wie dynamische Bindung in objektorientierten Sprachen funktioniert. Zunächst gehen wir von zwei Strukturen für die beiden Vorgänge und den verschiedenen `getDauer`-Implementierungen aus (vgl. Listing 7.2).

C/C++ 7.2: Überladen von `getDauer`

```
typedef struct {
  int menge; // Anzahl
  int maxperweg; // Anzahl pro Transport
  double wegeinmal; // Dauer fuer einen Weg
} TransportVorgang;

double getDauer(TransportVorgang* tv) {
  return (int)
    ((tv->menge-0.5)/tv->maxperweg + 1)
    * tv->wegeinmal;
}
```

```
typedef struct {
  double ruestzeit;
  double dauerprostueck;
  int menge; // Anzahl zu produz. Teile
} ProduktionsVorgang;


double getDauer(ProduktionsVorgang* pv) {
  return pv->ruestzeit
     + pv->menge * pv->dauerprostueck;
}
```

(netzplanung/v5-polymorphie/dynBindLoes1.h)

Wir können eine dritte Struktur anlegen, für die ebenfalls eine `getDauer`-Funktion existiert. In dieser Struktur speichern wir eine Referenz auf einen Transport- und einen Produktionsvorgang, wobei eine der beiden Referenzen immer `NULL` sein soll. Die Methode `getDauer` können wir dann in folgender Weise implementieren:

```
typedef struct {
  Transportvorgang* transportVorgang;
  Produktionsvorgang* produktionsVorgang;
} Vorgang;
```

```
double getDauer(Vorgang *v) {
  if (v->transportVorgang != NULL)
    return transportVorgang->getDauer();
  else if (v->produktionsVorgang != NULL)
    return produktionsVorgang->getDauer();
  else return 0; // kein Vorgang
}
```

Wenn es noch eine Reihe weiterer Vorgangsarten gibt, wollen wir nicht für jeden Vorgangstyp einen eigenen Zeiger speichern (ineffizient). Stattdessen könnten wir den Typ (bisher: Produktions- oder Transportvorgang) numerisch codiert speichern (`int vorgangstyp`) und einen `void*`-Zeiger über eine `switch`-Anweisung auf den „richtigen“ Typ casten:

```
typedef struct {
  void* vorgang;
  enum { TYP_TRANSPORT,
       TYP_PRODUKTION } vorgangstyp;
} Vorgang;
```

```
double getDauer(Vorgang *v) {
 switch (vorgangstyp) {
 case TYP_TRANSPORT : return getDauer(
              (TransportVorgang*)vorgang);
 case TYP_PRODUKTION: return getDauer(
             (ProduktionsVorgang*)vorgang);
  default           : return 0;
 }
}
```

Das Attribut `vorgangstyp` übernimmt hier eine Schlüsselfunktion bei der dynamischen Entscheidung zur Laufzeit, um welche Art von Vorgang es sich gerade handelt und welche Funktion aufgerufen werden soll. Damit erhalten wir genau die gewünschte Funktionalität, allerdings zu einem hohen Preis: Bei einer Änderung, zum Beispiel bei einem neuen Vorgangstyp, müssen wir *alle* diese Fallunterscheidungen manuell aufspüren und korrekt erweitern – andernfalls wird die dynamische Weiterleitung nicht überall korrekt ausgeführt. Aus Sicht der Wartbarkeit ist es hinderlich, dass die Verantwortung für die korrekte Weiterleitung der `getDauer`-Anfrage an die richtige Funktion *außerhalb* der einzelnen Vorgangs-Datenstrukturen liegt. Idealerweise erfordert das Einbeziehen einer weiteren Vorgangsart nur die Bereitstellung der neuen Implementierung eines bekannten Datentyps, aber keinerlei Änderungen an bereits existierenden Implementierungen. Dies ist auch als *Open Closed Principle (OCP)* bekannt [37]: Ein Modul bzw. eine Klasse sollte offen für Erweiterungen, aber geschlossen für Modifikationen sein.

7.2.1 Polymorphie selbst gemacht

Wie können wir dynamisch die richtige Funktion aufrufen und gleichzeitig die Verantwortung für den richtigen Funktionsaufruf bei den einzelnen Vorgängen belassen? Wir zeigen zunächst wieder das Prinzip mit prozeduralen Mitteln und werden sehen, dass uns der Compiler bei der objektorientierten Lösung einen Großteil der Arbeit abnimmt. Das Prinzip bleibt aber das gleiche. Wir erweitern zunächst unsere prozedurale Lösung: In unserem `struct Vorgang` speichern wir für jede Funktion (z.B. `getDauer`) einen Funktionszeiger (siehe Abschn. 4.4), der – abhängig davon, um wel-

che Implementierung es sich handelt – auf die richtige `getDauer`-Funktion zeigt. Wird die Schnittstellenfunktion `getDauer` angefordert, rufen wir diejenige Funktion auf, auf die unser Funktionszeiger zeigt. Die erste Deklaration eines solchen Funktionszeigers `fgetDauer` sieht vielleicht wie folgt aus:

```
double (*fgetDauer)(Vorgang*);
```

Dabei bemerken wir, dass der `getDauer`-Funktion des Transportvorgangs ein `TransportVorgang` und der des Produktionsvorgangs ein `ProduktionsVorgang` übergeben werden muss, sonst können diese die Vorgangsdauer nicht korrekt bestimmen. Wir haben daher in `Vorgang` einen Zeiger auf die erweiterten Daten des jeweiligen Transport-/Produktionsvorgangs vorgesehen. Wir müssen aber einen `void*`-Zeiger verwenden, weil die tatsächlich referenzierten Datenstrukturen je nach Vorgangsart unterschiedlich sind. Die `getDauer`-Funktionen können aus dem `Vorgang`-Argument dann über das Attribut `daten` auf die vorgangsspezifischen Details zugreifen.

Insgesamt brauchen wir für jede Funktion einen Funktionszeiger (hier nur `fgetDauer`) und den Zeiger auf die Datenrepräsentation (hier: `daten`). Unsere `Vorgang`-Datenstruktur degeneriert hierbei zu einer Art Hülle oder Anker für spätere Vorgangsergänzungen. Die Schnittstellenfunktionen für diese Basis-Datenstruktur können wir schon angeben: Sie ruft einfach die referenzierte `fgetDauer`-Funktion auf.

```
struct Vorgang {
  void* daten;          /* tatsächliche Daten dieses Vorgangs */
  double (*fgetDauer)(Vorgang*); /* Zeiger auf getDauer-Funktion */
}
double getDauer(Vorgang *v) {
  return (*fgetDauer)(v);
}
```

Dieser *allgemeine* Teil der Implementierung des abstrakten Datentyps enthält praktisch keine Funktionalität, d.h. ihre Erstellung könnte gut automatisiert werden. Für jede öffentliche Schnittstellenfunktion kommen ein Funktionszeiger und eine Funktion in der beschriebenen Art und Weise hinzu. Wie aber kommen wir nun zu den Varianten Transport- und Produktionsvorgang?

Wir gehen von der Anfangslösung aus Listing 7.2 aus und verändern nur das Argument der `getDauer`-Funktion so, dass es der Vorgangs-Schnittstelle entspricht, d.h. einen `Vorgang` und keinen `Transportvorgang` erwartet (Listing 7.3). Es kommt nur die Dereferenzierung des Attributs `daten` hinzu, ansonsten hat sich nichts geändert.

C/C++ 7.3: Ergänzung für Transportvorgänge (netzplanung/v5-polymorphie/Netzplanung.cpp)

```
/** ergaenzende Informationen zur struct Vorgangsdaten */
typedef struct TransportVorgangsDaten {
  int menge,maxperweg;
  double wegeinmal;
} TransportVorgang;
/** Berechnung der Dauer bei TransportVorgang */
double getDauerTV(Vorgang* v) {
  TransportVorgang* tv = (TransportVorgang*)v->daten;
  return (int) ((tv->menge-0.5)/tv->maxperweg + 1) * tv->wegeinmal;
}
```

Es bleibt nur noch zu klären, wie wir einen neuen Transportvorgang anlegen können und was dafür zu tun ist. Unsere alte Funktion `erzeugeVorgang` ersetzen wir – wie in Listing 7.4 gezeigt – durch eine Funktion `erzeugeTransportVorgang`, in der wir (a) Speicher für die benötigten Daten allokieren (`TransportVorgang`), (b) den Zeiger für die Datenrepräsentation (`daten`) korrekt setzen und (c) alle Funktionszeiger auf die mit der Datenrepräsentation korrespondierenden Funktionen setzen. Im Gegensatz zur Funktion `erzeugeVorgang` benötigen wir zum Anlegen eines Transportvorgangs weitere Argumente.

C/C++ 7.4: Anlegen eines Transportvorgangs (netzplanung/v5-polymorphie/Netzplanung.cpp)

```
/** Erzeugung eines Transportvorgangs, wobei a die Anzahl der zu transportierenden Teile ist,
  m die maximale Anzahl von Teilen, die auf einmal transportiert werden kann,
  e die Dauer für den einmaligen Transport */
Vorgang* erzeugeTransportVorgang(int a,double e,int m) {
  Vorgang *v = erzeugeVorgang(); /* ein TransportVorgang ist ein (erweiterter) Vorgang */
   /* Erweiterung anlegen */
  TransportVorgang *tv = new TransportVorgang();
  tv->menge = a; tv->wegeinmal = e; tv->maxperweg = m; v->daten = tv;
  v->fgetDauer = &getDauerTV; /* Setzen des Fkt-Zeigers */
  return v;
}
```

Resümieren wir abschließend, welche Dinge wir zu tun haben, um als weitere Vorgangsart einen Produktionsvorgang einzuführen. Es sind zwei Dinge: Erstens müssen wir die nötigen Daten zusammenfassen und die Schnittstellenfunktionen an diese Daten anpassen, wie in Listing 7.3 geschehen. Zweitens brauchen wir eine Erzeugerfunktion wie in Listing 7.4. In beiden Schritten ist kein Eingriff in bereits bestehende Funktionen oder Datenstrukturen von `Vorgang` oder `Transportvorgang` erforderlich. Damit haben wir unser Ziel erreicht: Wir können verschiedene Objekte mit `Vorgang*` referenzieren, die wir über `erzeugeTransportVorgang` oder über `erzeugeProduktionsvorgang` anlegen. Wenn wir dann zum Beispiel `getDauer(v)` aufrufen, wird je nachdem, welche Erzeugerfunktion wir vorher aufgerufen hatten, anschließend automatisch die korrespondierende `getDauer`-Funktion aufgerufen. Wir können weitere Vorgangsarten hinzufügen, ohne irgendwelche Änderungen am bestehenden Code vornehmen zu müssen. Das Problem der *dynamischen Bindung*, d.h. der Auswahl der richtigen Funktion zur Laufzeit, wurde gelöst!

Eine Schnittstelle (abstrakter Datentyp) kann mehrere, verschiedene konkrete Datenstrukturen abstrahieren. Wird die Entscheidung, welche Funktion konkret die Schnittstellenfunktion implementiert, zur Compile-Zeit festgelegt, spricht man von **statischer Bindung**. Erfolgt diese Entscheidung zur Laufzeit, so ist eine parallele Nutzung verschiedener Implementierungen möglich, und man spricht von **dynamischer Bindung**. Dabei obliegt die Verantwortung für die Auswahl der richtigen Funktion dem Ersteller des Datentyps und nicht dem Nutzer. In diesem Fall kann eine Variable (von einem bestimmten (abstrakten) Datentyp) in unterschiedlicher Gestalt vorkommen. Man spricht von Vielgestaltigkeit bzw. **Polymorphie**.

Wenn Klassen, zum Beispiel `Transportvorgang` und `Produktionsvorgang`, ein Interface implementieren, zum Beispiel `Vorgang`, dann spricht man auch von einer „**ist ein**“-Beziehung (ein `Transportvorgang` **ist ein** `Vorgang`).

7.2.2 Automatische Polymorphie

Weil die im vorangegangenen Abschnitt vorgestellte Vorgehensweise so nützlich ist, wird sie von objektorientierten Programmiersprachen in komfortabler Weise bereitgestellt. Damit entfallen die Deklaration der jeweiligen Funktionszeiger, die Deklaration des `daten`-Zeigers und die Implementierung der primitiven Schnittstellenfunktionen, die einfach die Funktion aufrufen, auf die der entsprechende Funktionszeiger zeigt. Das Prinzip bleibt aber das gleiche.

Der `Vorgang` mit den Funktionszeigern und den Umleitungs-Schnittstellenfunktionen wird durch eine Klasse oder in Java (wahlweise) durch ein Interface ersetzt, das nichts weiter als die Funktionsdeklarationen der Schnittstellenfunktionen enthält, aber keine Implementierung. Die administrativen Datenstrukturen (Funktionszeiger) und Umleitungsfunktionen werden praktisch vom Compiler erzeugt.

```
class Vorgang {
public:
  virtual double getDauer() const = 0;
};
```

```
interface Vorgang {
  double getDauer();
}
```

Die konkreten (Transport- und Produktions-)Vorgänge werden anschließend in eigenen Klassen implementiert. Die Tatsache, dass es sich um Klassen handelt, die die Vorgangs-Schnittstelle einhalten, wird durch `class Transportvorgang : public Vorgang` (C++) bzw. `class Transportvorgang implements Vorgang` (Java) angezeigt. Gibt es in `Transportvorgang` dann Funktionen mit identischen Rückgabewerten, Namen und Argumenten wie die Schnittstellenfunktion, so ordnet der Compiler diese Funktionen den Schnittstellenfunktionen zu (in unserem Modell entspricht das der Übernahme der Funktionszeiger). Das Beispiel in Listing 7.3 und 7.4 lässt sich somit in objektorientierten Sprachen viel einfacher umsetzen:

```
class Transportvorgang :
            public Vorgang {
  /* ...*/
  virtual double getDauer() const;
};
```

```
class Transportvorgang
            implements Vorgang {
  /* ...*/
  double getDauer() { return ...; }
}
```

Dabei ist für öffentliche Methoden der Schnittstelle, die dynamisch gebunden werden sollen (für die also in unserer prozeduralen Variante ein Zeiger angelegt werden soll), in C++ das Schlüsselwort **virtual** voranzustellen. In Java wird immer *für alle* Methoden die Möglichkeit zur dynamischen Bindung vorgesehen.

Eine Klasse heißt **abstrakte Klasse**, wenn die Implementierung einzelner oder aller *virtuellen* Methoden fehlt, was durch den Zusatz „$= \mathbf{0}$“ hinter der Funktionsdeklaration (C++) bzw. den Zusatz `abstract` vor der Funktionsdeklaration (Java) kenntlich gemacht wird. Eine abstrakte Klasse ist *unvollständig*, weil die abstrakten Methoden

in ihr nicht aufgerufen werden können, es gibt für sie noch keine zugeordnete Implementierung (Funktionszeiger ist noch NULL). Daher wird die Instanziierung (d.h. der Aufruf des Konstruktors) einer abstrakten Klasse auch vom Compiler mit einer Fehlermeldung quittiert. Eine Klasse, deren Methoden ausschließlich abstrakt sind, entspricht also einem Interface in Java, die beiden folgenden Java-Deklarationen sind daher nahezu gleichwertig.

```
public abstract class Vorgang {
  public abstract double getDauer();
}
```

```
public interface Vorgang {
  public double getDauer();
}
```

Während in Java konsequent alle Methoden polymorph behandelt werden, ist das in der sehr auf Effizienz bedachten Sprache C++ nicht automatisch der Fall. Immerhin wird durch den Einsatz von Polymorphie ein Funktionsaufruf nicht direkt ausgeführt, sondern er läuft über eine Zwischenstation (vgl. `getDauer`-Funktion auf Seite 173). In C++ hat man sich dafür entschieden, diesen Mehraufwand nicht generell zu leisten, sondern nur dann, wenn der Entwickler es wünscht. Die Motivation, in Java grundsätzlich alle Methoden dynamisch zu binden, liegt darin begründet, dass man als Entwickler oft noch nicht mit Sicherheit entscheiden kann, für welche Methoden man morgen einmal polymorphes Verhalten benötigt, daher wird konsequent jede (nicht private) Methode polymorph behandelt.

Durch das dynamische Binden müssen wir nun zwischen dem *statischen Typ* und dem *dynamischen Typ* einer Variablen unterscheiden.

```
Vorgang* vv = new Vorgang();
Vorgang* vt = new TransportVorgang(...);
vv = vt; /* 3 */
```

```
Vorgang vv = new Vorgang();
Vorgang vt = new TransportVorgang(...);
vv = vt; /* 3 */
```

Der statische Typ einer Referenz wird bei deren Deklaration festgelegt. Der statische Typ von `vv` und `vt` ist (für alle Zeiten) `Vorgang` oder ganz korrekt *Zeiger auf Vorgang* in C++. Der dynamische Typ einer Variablen kann sich ändern, es ist der Typ des referenzierten Objekts. Zunächst ist der dynamische Typ von `vv` `Vorgang`, und in Zeile `3` ist er dann `TransportVorgang`.

Bei einer polymorphen Methode ist es der dynamische Typ einer Variablen, der entscheidet, welche Methode aufgerufen wird. Fehlt in C++ das Schlüsselwort `virtual`, so ist die Methode nicht polymorph und der statische Typ entscheidet, aus welcher Klasse die Methode aufgerufen wird. Da der statische Typ bereits zur Compile-Zeit feststeht, kann hier schon beim Übersetzen entschieden werden, welche Methode aufzurufen ist, was zu der oben angesprochenen kleinen Laufzeitverbesserung für nicht virtuelle Methoden führt. Das folgende Listing veranschaulicht dies an einem C++-Beispiel. Es ist jeweils die Ausgabe als Kommentar angegeben.

```
class A {
 public:
   virtual void f() const {cout << "fA ";}
   void g() const {cout << "gA ";}
   void h() const {f();}
};
class B: public A {
 public:
   virtual void f() const {cout << "fB ";}
   void g() const {cout << "gB ";}
};
```

```
int main() {
   // statischer gleich dynamischer Typ
 A a; a.f(); a.g(); a.h(); //fA gA fA
   // statischer gleich dynamischer Typ
 B b; b.f(); b.g(); b.h(); //fB gB fB

   // sta. Typ A, dyn. Typ B--> fB gA fB
 A* pa=&b; pa->f(); pa->g(); pa->h();
   // sta. Typ B, dyn. Typ B--> fB gB fB
 B* pb=&b; pb->f(); pb->g(); pb->h();
}
```

Mit den Operatoren `instanceof` in Java bzw. `dynamic_cast` in C++ können wir zur Laufzeit bestimmen, welcher dynamische Typ bei einer Instanz vorliegt; siehe Abschn. 7.3.4.

Nachdem wir neben einem `Transportvorgang` auch einen `Produktionsvorgang` in der beschriebenen Weise angelegt haben, können wir in einem Array von Vorgängen in beliebiger Reihenfolge mal Transport- und mal Produktionsvorgänge einfügen, wie das folgende Java-Beispiel zeigt:

```
Vorgang vv[] = new Vorgang[2];
vv[0] = new TransportVorgang(...);
vv[1] = new ProduktionsVorgang(...);
double gesamtdauer = 0;
for (int i=0;i<2;++i) gesamtdauer += vv[i].getDauer();
```

Beim Abarbeiten der Liste zur Bestimmung der Gesamtdauer aller Vorgänge wird nun dank der Polymorphie die jeweils richtige `getDauer`-Funktion aufgerufen.

Es soll an dieser Stelle betont werden, dass das Beispiel aus Abschn. 7.2.1 nur *ein Modell* dafür ist, wie man Polymorphie sauber erreichen kann, die in der Praxis vom Compiler durchgeführte Umsetzung ist implementierungsabhängig. So ist es beispielsweise nicht unbedingt sinnvoll, für jedes Objekt immer wieder so viele Zeiger anzulegen, wie es Funktionen gibt. Diese Liste oder Tabelle von Funktionszeigern kann sich nicht beliebig zur Laufzeit ändern, sondern es gibt nur so viele verschiedene Belegungen, wie es auch verschiedene Implementierungen des Datentyps gibt. Eine effizientere Umsetzung besteht daher darin, für jede neue Klasse eine Tabelle für alle (virtuellen/polymorphen) Funktionszeiger anzulegen und eine Instanz mit nur einem Zeiger auf diese Tabelle mitführen zu lassen (**virtual method table**).

Immer wenn Variablen in Wertesemantik vorliegen, ist der statische gleich dem dynamischen Typ, und Funktionsaufrufe können statisch gebunden werden. Wenn verschiedene Realisierungen des Datentyps unterschiedlich groß sein können und wir im Sinne eines polymorphen Datentyps nicht wissen, welche Gestalt ein Vorgang gerade annimmt, ist das Anlegen einer polymorphen Variablen in Wertesemantik schlicht unmöglich. Eine Vorbedingung für das Auftreten von dynamischer Bindung ist also, dass wir es mit Zeigern oder Referenzen zu tun haben. Wieder hat der C++-Entwickler die Wahl, in Java werden alle Methodenaufrufe dynamisch gebunden.

In unserem Beispiel heißt das: Schreibt man in C++ `vector<TransportVorgang>`, so drückt man damit aus, dass man Wert-Instanzen von (ausschließlich) Transport-

vorgängen speichern möchte (bestehend aus Anzahl, Weglänge und Einheitsdauer). Der Typ ist damit eindeutig festgelegt, und Polymorphie ist unnötig (und unmöglich). Schreibt man hingegen `vector<TransportVorgang*>`, so drückt man damit aus, dass man Referenzen auf Transportvorgänge speichern will, wobei es dann durchaus möglich ist, dass es mehrere verschiedene Transportvorgänge geben kann, von denen automatisch die richtige Funktion aufgerufen wird. Diese Überlegungen braucht ein Java-Entwickler nicht anzustellen; semantisch entspricht bei ihm `Vector<Vorgang>` immer der C++-Variante `vector<Vorgang*>`.

7.2.3 Polymorphie ganz konkret

Anfänger neigen dazu, den Zauber der Polymorphie so zu verstehen, dass wie durch ein Wunder immer die richtige Funktion gefunden und aufgerufen wird. Obwohl durch unsere Modellvorstellung klar sein sollte, wann genau Polymorphie greift und wann nicht, verdeutlichen wir uns diesen Sachverhalt anhand einiger Beispiele nochmals.

Folgende Beispiele sind in Java formuliert (für entsprechenden C++-Code nehmen wir an, dass alle Methoden virtuell sind und alle Argumente call-by-reference übergeben werden).

1. Welche der drei Funktionen in `Netzplanung` wird aufgerufen, wenn wir `f` mit einem `Transportvorgang` aufrufen?

```
class Netzplanung {
  void addNachfolger(Vorgang v) {...}

  void addNachfolger(TransportVorgang v) {...}
  void addNachfolger(ProduktionsVorgang v) {...}

  void f(Vorgang v) { addNachfolger(v); }
}
```

2. Welche Methode `addNachfolger` wird aufgerufen, wenn die Methode `f` mit einem `Produktionsvorgang` als Argument aufgerufen wird?

```
abstract class Vorgang {
  abstract void addNachfolger(Vorgang v);
}

class TransportVorgang extends Vorgang {
  void addNachfolger(TransportVorgang v) {...}
}

class ProduktionsVorgang extends Vorgang {
  void addNachfolger(ProduktionsVorgang v) {...}
}

class Demo {
  void f(Vorgang x) {
    x.addNachfolger(new TransportVorgang(..)); }
}
```

3. Welche Funktion `addNachfolger` wird aufgerufen, wenn die Funktion f mit einem `Produktionsvorgang` als Argument aufgerufen wird?

```
abstract class Vorgang {
  abstract void addNachfolger(Vorgang v);
}

class TransportVorgang extends Vorgang {
  void addNachfolger(Vorgang v) {...}
  void addNachfolger(TransportVorgang v) {...}
}

class ProduktionsVorgang extends Vorgang {
  void addNachfolger(Vorgang v) {...}
  void addNachfolger(ProduktionsVorgang v) {...}
  void addNachfolger(TransportVorgang v) {...}
}

class Demo {
  void f(Vorgang x) {
    x.addNachfolger(new TransportVorgang(..)); }
}
```

4. Abschließend das gleiche Beispiel noch einmal als C++-Code. Welche Funktion `addNachfolger` wird aufgerufen, wenn die Funktion f mit einem `Produktionsvorgang` als Argument aufgerufen wird?

```
class Vorgang {
  virtual void addNachfolger(Vorgang v) = 0;
}

class TransportVorgang : public Vorgang {
  virtual void addNachfolger(Vorgang v) {...}
  virtual void addNachfolger(TransportVorgang v) {...}
}

class ProduktionsVorgang : public Vorgang {
  virtual void addNachfolger(Vorgang v) {...}
  virtual void addNachfolger(ProduktionsVorgang v) {...}
  virtual void addNachfolger(TransportVorgang v) {...}
}

class Demo {
  void f(Vorgang x) {
    TransportVorgang tv(...);
    x.addNachfolger(tv); }
}
```

Antworten:

1. Nach unserer Modellvorstellung wird für jede Funktion der Klasse `Netzplanung` ein eigener Funktionszeiger bereitgestellt. Würde die Netzplanung mehrfach implementiert, so fänden wir durch Verfolgung der Funktionszeiger jeweils die richtige Implementierung. Hier haben wir es zwar offenbar mit mehreren verschiedenen Vorgängen zu tun, aber (da nichts anderes erwähnt ist) nur mit einer Netzplanung. Der Aufruf von `f` ist somit überhaupt kein Thema für Polymorphie. Anhand des Typs des Arguments (Vorgang) wird die erste `addNachfolger`-Funktion als passend herausgefunden und aufgerufen.
2. Wieder liegt keine Polymorphie vor: Voraussetzung für Polymorphie ist, dass dieselbe Funktion durch verschiedene Implementierungen mehrfach vorliegt. Das ist nicht der Fall, wir haben hier unterschiedliche Signaturen und damit unterschiedliche Methoden vorliegen: einmal kann man einen `Vorgang` hinzufügen, einmal einen `Transportvorgang` und einmal einen `Produktionsvorgang`. Keine Signatur taucht zwei Mal auf, und es wird somit kein Funktionszeiger überschrieben. Weil das Argument `x` den Typ `Vorgang` hat, ergibt sich der Aufruf der `addNachfolger`-Methode dieser Klasse als einzige Möglichkeit.
3. Im dritten Beispiel liegt nun endlich der Fall vor, dass die Funktion `addNachfolger` aus der Klasse `Vorgang` sowohl von `Produktions-` als auch von `Transportvorgang` implementiert wird und damit die Funktionszeiger entsprechend gesetzt werden. Wenn das Argument `x` ein `Produktionsvorgang` ist, dann wird entsprechend die Methode `addNachfolger(Vorgang v)` aus der Klasse `Produktionsvorgang` aufgerufen. Wie schon im vorigen Fall erklärt, wird nicht die Methode `addNachfolger(TransportVorgang v)` aufgerufen, weil es diese in der Schnittstelle von `Vorgang` gar nicht gibt (nur eine Methode mit einem `Vorgang` als Argument).
4. Die Fragestellung besagt zwar, *dass es sich um dasselbe Beispiel in C++ handelt*, doch ist das nicht ganz richtig: Der Code ist zwar fast identisch, aber in C++ werden in dem Beispiel alle Argumente als Werteparameter übergeben (sonst hätte die Deklaration `addNachfolger(Vorgang* v)` oder `addNachfolger(Vorgang& v)` heißen müssen. Der Code scheitert schon bei der Compilierung, weil die Klasse `Vorgang` in allen unseren Beispielen abstrakt war und es daher überhaupt nicht möglich ist, eine Wertinstanz von `Vorgang` anzulegen. Selbst wenn dem nicht so wäre und beim Aufruf von `addNachfolger(...)` eine Kopie des `Transportvorgangs` für den Übergabeparameter angelegt werden könnte, handelt es sich bei `x` danach um einen `Vorgang`, und damit wird die `addNachfolger`-Methode aus der Klasse `Vorgang` aufgerufen.

Die **dynamische Bindung** kann nur dort erfolgen, wo über den Punktoperator (oder Pfeiloperator) eine Methode aufgerufen wird, die mit identischem Prototyp mehrfach in verschiedenen Implementierungen realisiert wurde (Überschreiben des Funktionszeigers).

7.3 Vererbung

Wir haben uns in diesem Kapitel bisher auf die Methode `getDauer` der Vorgangs-Schnittstelle beschränkt, weil sie sich bei Transport- und Produktionsvorgängen unterscheidet. Die vollständige Vorgangs-Schnittstelle aus Kap. 6 enthält noch einige andere Methoden, wie etwa die Abfrage des frühesten Startpunktes eines Vorgangs mit `getFruehAnf`. Diese müssen wir noch anpassen, wenn unsere Netzplanung am Ende dieses Kapitels wieder lauffähig sein soll.

Für diese und andere Methoden hatten wir in Kap. 6 einige Attribute bei jedem Vorgang vorgesehen, u.a. `fruehanf` für den frühesten Anfangs- und `spaetend` für den spätesten Endzeitpunkt des Vorgangs. Diese Attribute wurden (über `set`-Methoden) von der Netzplanung gesetzt, sodass die Realisierung von `getFruehAnf` nur aus `return fruehanf` bestand.

Da sowohl Transport- als auch Produktionsvorgänge die komplette Vorgangs-Schnittstelle implementieren müssen, brauchen beide Klassen die entsprechenden Attribute und Zugriffsfunktionen. Allerdings unterscheidet sich die Funktionalität in diesem Punkt in beiden Fällen überhaupt nicht, sondern ist absolut identisch. Es ist natürlich – dank Copy & Paste – nicht schwer, diese Attribute und Zugriffsfunktionen zwei Mal zu implementieren, aber das Kopieren von Quelltext ist als kritisch zu bewerten: In dem Codeabschnitt, der durch Copy & Paste dupliziert wurde, kann sich ein Fehler befinden. Wenn dieser Fehler an einer Stelle im Code auffällt und dann behoben wird, gibt es keine Möglichkeit, festzustellen, wohin dieses Code-Fragment überall kopiert worden ist. Das wäre aber wichtig, um in allen Kopien den Fehler ebenfalls zu korrigieren. Im Resultat wird das Debugging dann so oft durchgeführt, wie wir das Fragment vorher kopiert haben. Copy & Paste kann sich damit zu einem teuren Luxus entwickeln.

7.3.1 Code in mehreren Klassen gemeinsam nutzen

Eine Lösung, die ohne Copy & Paste auskommt, präsentieren wir zunächst wieder an unserer prozeduralen Lösung: Wenn wir im `Vorgang` an der Stelle, an der wir die gesamten Funktionszeiger speichern, die von `Produktions-` und `Transportvorgang` *gemeinsam* benötigten Elemente hinterlegen, können wir für eine Vorgangsreferenz `v` direkt auf diese Attribute zugreifen, z.B. `v->fruehanf`, aber auch auf die Ergänzungen des Transportvorgangs über `(TransportVorgang*) v->daten`:

C/C++ 7.5: Gemeinsam genutzte Attribute (netzplanung/v5-polymorphie/Netzplanung.cpp)

```
struct Vorgang {
  void* daten;
  double fruehanf,spaetend;
  int id;

  double (*fgetDauer)(Vorgang *);
  double (*fgetFruehAnf)(Vorgang *);
  double (*fgetSpaetEnd)(Vorgang *);
  void (*fsetFruehAnf)(Vorgang *,double);
```

```
  void (*fsetSpaetEnd)(Vorgang *,double);
  int (*fgetId)(Vorgang *);
  void (*fsetId)(Vorgang *,int);
};
```

Mehr noch, wir können auch eine Funktion zur Abfrage des frühesten Anfangszeitpunktes bereits für `Vorgang` implementieren (denn alle dafür benötigten Attribute sind in `Vorgang` bereits vorhanden) und den Funktionszeiger `fgetFruehAnf` für alle Vorgangstypen auf diese Funktion setzen. Damit ist diese Schnittstellenfunktion bereits implementiert und muss z.B. nicht mehr durch `Transportvorgang` implementiert werden.

```
double getFruehAnfV(Vorgang* v) {
  return v->fruehanf;
}
```

Listing 7.6 zeigt die Erzeugerfunktion für `Vorgang`, dort weisen wir diese gemeinsam genutzte Implementierung den entsprechenden Funktionszeigern zu. Wir lassen in unserem Beispiel nur die Funktion `getDauer` aus (NULL-Zeiger), weil wir hierfür keine gemeinsam genutzte Implementierung haben.

C/C++ 7.6: Erzeugerfunktion für `Vorgang` (netzplanung/v5-polymorphie/Netzplanung.cpp)

```
Vorgang* erzeugeVorgang() {
  Vorgang *v = new Vorgang();
  v->daten = NULL; v->id = -1;
  v->fgetDauer=NULL; // keine Funktion bisher
  v->fgetFruehAnf=&getFruehAnfV; v->fsetFruehAnf=&setFruehAnfV;
  v->fgetSpaetEnd=&getSpaetEndV; v->fsetSpaetEnd=&setSpaetEndV;
  v->fgetId=&getIdV; v->fsetId=&setIdV;
  return v;
}
```

Die Erzeugerfunktion für einen `Transportvorgang` muss zunächst eine Instanz von `Vorgang` anlegen und kann dann gezielt Funktionszeiger *überschreiben*, d.h. ausgewählte Implementierungen durch neue ersetzen, oder sie eben beibehalten, wenn sich gegenüber der Default-Implementierung nichts geändert hat. Diese Übernahme von Implementierungen für verschiedene Realisierungen von Vorgängen wird auch **Vererbung** genannt. Sie erspart das explizite Kopieren von Quellcode. Im Fall von `TransportVorgang` überschreiben wir nur die `getDauer`-Funktion, die aber vorher *abstrakt* war und der somit noch keine Implementierung zugewiesen worden ist.

C/C++ 7.7: Erzeugerfunktion für TransportVorgang (netzplanung/v5-polymorphie/Netzplanung.cpp)

```
Vorgang* erzeugeTransportVorgang(int a,double e,int m) {
  Vorgang *v = erzeugeVorgang(); /* ein TransportVorgang ist ein (erweiterter) Vorgang */
   /* Erweiterung anlegen */
  TransportVorgang *tv = new TransportVorgang();
  tv->menge = a; tv->wegeinmal = e; tv->maxperweg = m; v->daten = tv;
  v->fgetDauer = &getDauerTV; /* Setzen des Fkt-Zeigers */
  return v;
}
```

Unter **Vererbung** versteht man die Übernahme von Attributen und Methoden von der einen Realisierung in eine andere Realisierung des gleichen (abstrakten) Datentyps. Vererbung ermöglicht die Vermeidung von Code-Duplikation und kann damit die Wartung von Software erleichtern.

Von objektorientierten Sprachen wird das Vererbungskonzept durch geeignete Sprachkonstrukte gut unterstützt. Fahren wir nach `class B` mit `: public A` (C++) bzw. `extends A` (Java) fort, so *erbt* `B` alle Attribute und Methoden von `A`. Unsere prozedurale Version in der objektorientierten Variante sieht dann wie folgt aus (etwas zusammengefasst):

```
class Vorgang {
protected:
  double fruehAnf,spaetEnd;
public:
  virtual double getDauer() = 0;
  virtual double getFruehAnf() {
    return fruehanf; }
  virtual double getSpaetEnd() {
    return spaetend; }
}

class Transportvorgang
  : public Vorgang {
protected:
  int menge,maxperweg;
  double wegeinmal;
public:
  virtual double getDauer() { return ...; }
}
```

```
abstract class Vorgang {

  protected double fruehAnf,spaetEnd;

  public abstract double getDauer();
  public double getFruehAnf() {
    return fruehanf; }
  public double getSpaetEnd() {
    return spaetend; }
}

class Transportvorgang
   extends Vorgang {

  protected int menge,maxperweg;
  protected double wegeinmal;

  public double getDauer() { return ...; }
}
```

In jeder Methode von `TransportVorgang` können wir *sowohl* auf die Attribute dieser Klasse als *auch* auf die Attribute der so genannten **Vaterklasse** `Vorgang` zugreifen, weil die Klasse `TransportVorgang` die Deklarationen und die Methodenimplementierungen geerbt hat, sofern die Attribute durch den Sichtbarkeitsmodifikator **private** nicht geschützt sind. Auf öffentliche Attribute kann ohnehin jede andere Klasse zugreifen, der Zugriff auf geschützte Attribute und Methoden (**protected**) ist bei abgeleiteten Klassen uneingeschränkt möglich. Andere gebräuchliche Namen für Vaterklasse sind auch **Oberklasse** oder **Basisklasse**.

Auch hier gilt, wie schon vorher bei der Polymorphie, dass unsere prozedurale Lösung nur ein Modell dafür ist, von dessen Implementierung eine objektorientierte Sprache abweichen wird. Bild 7.1 zeigt auf der linken Seite die Vorgehensweise in unserer prozeduralen Lösung: Die Basisklasse `Vorgang` hatte einige Attribute, die dann durch eine `void*`-Referenz um weitere Daten ergänzt werden konnte. Dieser Aufbau muss in unserer Lösung zur Laufzeit hergestellt werden, was aber nicht nötig wäre, denn schließlich weiß der Compiler schon zur Übersetzungszeit, welche Klassen von welchen anderen Klassen erben. Effizienter ist daher die übliche Umsetzung rechts in Bild 7.1, bei der jede abgeleitete Klasse aus den Daten der Vaterklasse besteht und sie ggf. um weitere Attribute ergänzt.

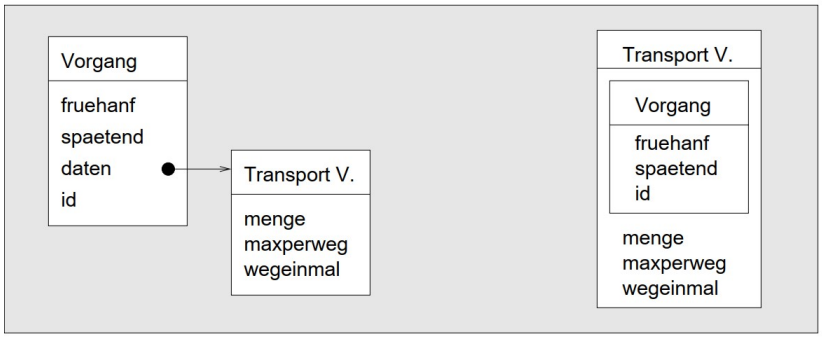

Bild 7.1: Gemeinsame Nutzung von Daten durch Vererbung; *links:* in unserem prozeduralen Modell; *rechts:* übliche Umsetzung in OO-Sprachen

7.3.2 Konstruktor-Verkettung

In Listing 7.8 sehen wir ein weiteres Beispiel für eine Vorgangsklasse. Dieses Mal hat ein Vorgang eine laufende Nummer und eine Dauer, wobei die laufende Nummer über den Konstruktor und die Dauer über eine `init`-Methode initialisiert wird, die der Konstruktor aufruft. Erstellen wir eine neue Klasse `TransportVorgang`, die von `Vorgang` abgeleitet ist (noch ohne jegliche Methoden), so scheitert der Versuch, ein Objekt von diesem Typ anzulegen. Keine Probleme beim Anlegen eines Transportvorgangs gibt es hingegen, wenn wir den Konstruktor der Klasse `Vorgang` *löschen*. Woran liegt das?

C++/Java 7.8: Vorgangsklasse

```
class Vorgang {
  int nummer;
  double dauer;
  virtual void init() { dauer=12.3; }
public:
  Vorgang(int n) { cout << "V ";
    init(); nummer=n; }
  virtual void print() { cout << nummer
    << ' ' << dauer << ' '; }
  ~Vorgang() {
    cout << "-V "; }
};
```

(Poly/poly.cpp)

```
class Vorgang {
  protected int nummer;
  protected double dauer;
  protected void init() { dauer=12.3; }

  public Vorgang(int n) { System.out.
    print("V "); init(); nummer=n; }
  public void print() { System.out.
    print(nummer+" "+dauer+" "); }
  public void finalize() { System.out.
    print("-V "); }
};
```

(Poly/Poly.java)

Wir haben auf Seite 139 den **Default-Konstruktor** kennengelernt, den der Compiler für uns automatisch erstellt. Wenn weder `Vorgang` noch `TransportVorgang` von uns einen Konstruktor erhalten, dann erstellt der Compiler für beide den Default-Konstruktor ohne Parameter. Dank deren Existenz verläuft der zweite Fall problemlos. Der Compiler unterbindet aber die automatische Erzeugung von Konstruktoren für eine Klasse, sobald wir einen (beliebigen) Konstruktor für diese Klasse selbst vorgeben. Dies ist im Fall von `Vorgang` geschehen. Somit wird für `Vorgang` kein ar-

gumentloser Default-Konstruktor mehr generiert – wohl aber für `TransportVorgang`. Dabei ergibt sich für den Compiler folgendes Problem: Da aufgrund der Vererbungsbeziehung ein `TransportVorgang` ein `Vorgang` ist, muss zunächst ein `Vorgang` angelegt werden, der dann zu einem `TransportVorgang` ergänzt wird (vgl. auch Bild 7.1). Der Default-Konstruktor von `TransportVorgang` muss einen Konstruktor von `Vorgang` aufrufen. Es gibt aber nur einen, und der erfordert ein Argument – woher soll der Compiler dieses Argument nehmen? Somit scheitert die automatische Erzeugung des Default-Konstruktors, und wir können kein `TransportVorgang`-Objekt anlegen.

Der Compiler kann uns mit einem Default-Konstruktor nun nicht mehr helfen, wir müssen den Konstruktor für `TransportVorgang` selbst implementieren; siehe dazu Listing 7.9. Dabei muss der Aufruf des Konstruktors der Oberklasse zwingend die erste Tätigkeit sein. In C++ erfolgt der Konstruktoraufruf der Oberklasse nach einem Doppelpunkt (`: Vorgang(n)`), in Java erfolgt er durch eine einleitende `super(n)`-Anweisung (dabei steht `super(n)` für den Aufruf des Konstruktors der Oberklasse, hier `Vorgang`, mit Argument `n`).

C++/Java 7.9: Konstruktor-Verkettung

```
class TransportVorgang : public Vorgang {
  vector<int> *barcodes;
  virtual void init() {
    barcodes = new vector<int>();
    barcodes->push_back(21); }
public:
  TransportVorgang(int n) : Vorgang(n) {
    cout << "TV "; init(); }
  virtual void print() { Vorgang::print();
    cout << barcodes->size() << ' '; }
  ~TransportVorgang() { delete barcodes;
    cout << "-TV "; }
};
```

(Poly/poly.cpp)

```
class TransportVorgang extends Vorgang {
  protected Vector barcodes;
  protected void init() {
    barcodes = new Vector();
    barcodes.add(new Integer(21)); }

  public TransportVorgang(int n) { super(n);
    System.out.print("TV "); init(); }
  public void print() { super.print();
    System.out.print(barcodes.size()+" "); }
  public void finalize() { System.out.
    print("-TV "); super.finalize(); }
};
```

(Poly/Poly.java)

> Bevor der Konstruktor einer abgeleiteten Klasse (Unterklasse) ausgeführt wird, wird immer zunächst der Konstruktor der Basisklasse (Basisklassen bei C++-Mehrfachvererbung) aufgerufen. Dieser muss explizit als erste Anweisung aufgerufen werden, wenn kein Default-Konstruktor vorhanden ist. Entsprechend wird in C++ zunächst der Destruktor der abgeleiteten Klasse aufgerufen und anschließend implizit der Destruktor der Basisklasse(-n).

Alles virtuell, oder was? In C++ muss jede Methode, von der wir polymorphes Verhalten erwarten, als virtuell deklariert sein – in Java ist jede Methode automatisch virtuell. Obwohl es sich in Listing 7.10 bei `v` um einen (Zeiger auf einen) Vorgang handelt, erwarten wir, dass diejenige `print`-Methode aufgerufen wird, von deren Typ das von `v` referenzierte Objekt ist. Im Listing haben wir `v` auf einen Transportvorgang zeigen lassen. Somit sollte die `print`-Methode von `TransportVorgang` aufgerufen werden.

C++/Java 7.10: Konstruktor-Verkettung

```cpp
int main() {
  Vorgang *v = new TransportVorgang(87);
  v->print();
  delete v;
  return 0;
}
```

(Poly/poly.cpp)

```java
public static void main(String[] args) {
  // System.runFinalizersOnExit(true);
  Vorgang v = new TransportVorgang(87);
  v.print();
  // Garbage Collector räumt alleine auf
}
```

(Poly/Poly.java)

Dass dem auch so ist, zeigt die Ausgabe des Programms – allerdings sind die Ausgaben des Programms in C++ und Java nicht identisch, und keine der Ausgaben entspricht vollständig unserer Erwartung.

erwartet:	`V TV 87 12.3 1 -TV -V`
C++:	`V TV 87 12.3 1 -V`
Java:	`V TV 87 0.0 1`

Welche Ausgabe erwarten wir? Der Aufruf des Konstruktors von `TransportVorgang` ruft als Erstes (noch bevor irgendeine Bildschirmausgabe erfolgen kann) den Konstruktor der Oberklasse auf. Zuerst sollte die Ausgabe `V` erscheinen, danach die Ausgabe `TV`. Da `v` einen `Transportvorgang` referenziert, sollte die `print`-Methode des `Transportvorgangs` aufgerufen werden: Von dort aus wird zunächst die `print`-Methode der Oberklasse `Vorgang` aufgerufen (C++: `Vorgang::print()`, Java: `super.print()`), bevor noch die Anzahl der Elemente im Vektor `barcodes` ergänzt wird. Wir erwarten die Ausgabe dreier Zahlen `87`, `12.3` und `1`. Nun erwarten wir, dass analog zum Aufruf der Konstruktoren zunächst der Destruktor von `TransportVorgang` und anschließend der von `Vorgang` aufgerufen wird,[1] damit jede Klasse die von ihr verwalteten Daten wieder freigeben kann (z.B. den Vektor mit den Barcodes).

In der Ausgabe des C++-Programms fehlt aber der Destruktor-Aufruf des Transportvorgangs: Er wird niemals aufgerufen, der für `barcodes` allokierte Speicher wird nicht wieder freigegeben. Die Erklärung für dieses Verhalten ist ganz einfach: Auch ein Destruktor ist eine Methode. Die Anweisung `delete v` in `main` ruft den Destruktor auf. Da dieser aber nicht als virtuell deklariert wurde, bestimmt der statische Typ von `v` (hier `Vorgang`) darüber, welcher Destruktor aufgerufen wird. Richtig wäre der Destruktor von `TransportVorgang` gewesen. Deklarieren wir auch den Destruktor als virtuell, erhalten wir die erwartete Ausgabe. Hierbei handelt es sich um eine *beliebte* Fehlerquelle in C++-Programmen, an deren Vermeidung der C++-Entwickler selbst denken muss: Destruktoren sollten **immer** als `virtual` deklariert werden.

Tipp

Der Java-Entwickler hat es in diesem Punkt wesentlich einfacher. Dieses Problem kann nicht entstehen, weil erstens stets alle Methoden virtuell sind und zweitens der Garbage Collector den Objektabbau übernimmt. Das ist auch der Grund, warum wir in der Java-Ausgabe *überhaupt* keine Destruktor-Tätigkeit beobachten können. Die `finalize`-Methoden führen in Java ein Schattendasein, und von ihrer Verwendung ist eher abzuraten; z.B. muss die Verkettung der `finalize`-Methoden „von Hand“

[1] Stellen Sie sich die Objekterzeugung wie einen Turmbau vor: die Basisklasse kommt als Sockel zuerst, danach wird die Verfeinerung aufgesetzt. Um den Turm wieder geordnet abzubauen, müssen Sie die oberen Teile zuerst entfernen.

erfolgen (`super.finalize()` am Ende von `TransportVorgang.finalize()`), und durch die auskommentierte Zeile zu Beginn von `main` muss dafür gesorgt werden, dass bei Programmende wenigstens alle `finalize`-Methoden auch aufgerufen werden. Dann erscheinen endlich die erwarteten Zeichen `-TV -V`.

Ein subtiler Unterschied zwischen C++ und Java: Damit entspricht die Java-Zeile aber noch nicht der erwarteten Ausgabe, weil statt der erwarteten Dauer `12.3` ein Wert von `0.0` ausgegeben wird. Dieses unerwartete Verhalten weist auf einen subtilen Unterschied beim Aufbau der Objekte in C++ und Java hin. In Java ist das Objekt, das wir durch `new TransportVorgang()` erzeugen, vom ersten Moment an vom Typ `TransportVorgang` – schon wenn wir zuerst den Konstruktor von `Vorgang` aufrufen. Der Aufruf von `init` erfolgt erwartungsgemäß polymorph und führt – da es sich um einen Transportvorgang handelt – zu `TransportVorgang.init()` statt (wie in C++) zu `Vorgang.init()`. Damit wird die `init`-Methode des `Vorgangs` nie aufgerufen, und `dauer` behält seinen Default-Wert. Als Richtlinie für Java-Entwickler sollte daher gelten, solche `init`-Methoden als `private` zu deklarieren, denn private Methoden können nicht überschrieben werden. Eine weitere Möglichkeit besteht darin, solche `init`-Aufrufe im Konstruktor gänzlich zu vermeiden. Java erlaubt es einem Konstruktor, mit `this(...);` einen anderen Konstruktor derselben Klasse zuvor aufzurufen (**Konstruktor-Verkettung**). Dann kann der Inhalt von `init` in einen (privaten) Default-Konstruktor verlagert werden:

Tipp

```
private Vorgang() {
  // Initialisierung
}
public Vorgang(int n) {
  this(); // Aufruf von Vorgang() und damit der Initialisierung
  // n verarbeiten
}
public Vorgang(int n, double d) {
  this(n); // Aufruf von Vorgang(n) und damit der Initialisierung
  // nur noch d verarbeiten
}
```

7.3.3 UML-Klassendiagramme: *Vererbung und Polymorphie*

Abstrakte Klassen, abstrakte Methoden:

Vererbung (Typerweiterung, Subtyping, Spezialisierung):

Wiederholte Vererbung und Mehrfachvererbung: Wir haben gesehen, dass wir von der Basisklasse `Vorgang` *mehrfach* erben können, einmal erbt `TransportVorgang` und ein weiteres Mal `ProduktionsVorgang`. Diese Form der mehrfachen Vererbung – die nicht die festgelegte Bedeutung von Mehrfachvererbung hat (siehe unten) – entspricht der Eingangsmotivation des Kapitels: derselbe abstrakte Datentyp soll mehrfach implementiert werden. Gemeinsamkeiten der verschiedenen Realisierungen werden nur einmal implementiert und an abgeleitete Klassen weitergegeben.

Statt mehrmals von `Vorgang` abzuleiten, könnten wir aber auch weitere *Produktionsvorgänge* einführen wollen, die alle für sich spezielle Produktionsvorgänge sind, aber weitere Informationen speichern müssen. Dann haben wir – wie in Bild 7.2 dargestellt – mehrfache Vererbung in dem Sinne vorliegen, dass eine *lineare Kette* von Ableitungen entsteht, man nennt das auch *wiederholte Vererbung (repetetive inheritance)*. Dabei erbt eine Unterklasse von *allen* Oberklassen und ist zu allen Oberklassen typkompatibel. Diese Art der verketteten Vererbung ist ohne weiteres möglich, auch in unserer prozeduralen C-Variante (so können wir bspw. auch im `ProduktionsVorgang` neue Funktionszeiger einführen, die dann polymorphe Schnittstellenfunktionen für Produktionsvorgänge ermöglichen). Man spricht hier von *Vererbungshierarchien*.

Unter dem Begriff der *Mehrfachvererbung* versteht man jedoch eine andere Art der mehrfachen Ableitung, nämlich die gleichzeitige, direkte Ableitung einer Klasse von *mehreren* Oberklassen, wie sie im rechten Teil von Bild 7.2 dargestellt ist.

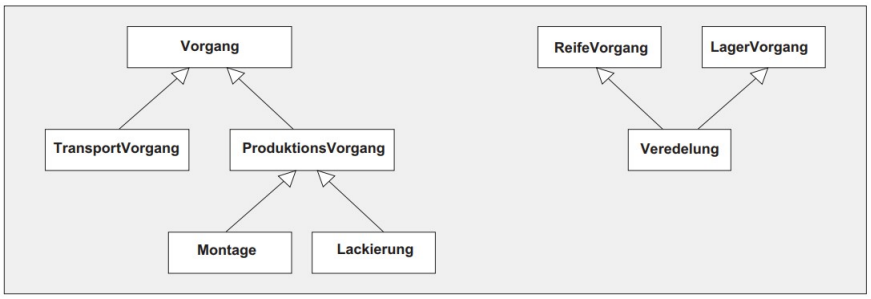

Bild 7.2: *links:* Vererbungshierarchie (wiederholte Vererbung), *rechts:* Ein Beispiel für Mehrfachvererbung

Bild 7.3 zeigt ein anderes Beispiel für *Mehrfachvererbung*: ein Amphibienfahrzeug, das sowohl ein Auto als auch ein Schiff ist, d.h., die Unterklasse erbt von zwei Oberklassen. Mehrfachvererbung gibt es nur in C++ und ist ein komplexes Konzept, gegen das es berechtige Vorbehalte gibt, die an diesem Beispiel deutlich werden:

Probleme entstehen speziell dann, wenn – wie hier – zwei Oberklassen (hier: `Auto` und `Schiff`) einer Klasse (hier: `Amphibienfahrzeug`) selbst wieder von einer gemeinsamen Klasse (hier: `Fahrzeug`) abgeleitet sind, sodass je nach Anwendung zu diskutieren ist, ob deren Eigenschaften jetzt doppelt in der Klasse (hier: `Amphibienfahrzeug`) vorhanden sind oder nicht; siehe C++-Kompendium [30, Abschn. 9.4] für weitere Details.

Völlig unproblematisch ist es hingegen, eine Klasse mehrere Schnittstellen implementieren zu lassen, d.h. in Java wäre von mehreren *Interfaces* bzw. in C++ von mehreren *rein* abstrakten Klassen abzuleiten (keine Daten, keine Implementierungen). Java stellt für diesen speziellen Fall das Schlüsselwort `interface` zur Verfügung. In diesen Fällen gibt es keine Attribute oder Methodenimplementierungen, die geerbt werden könnten, weshalb sich die angesprochene Problematik auch nicht ergeben kann.

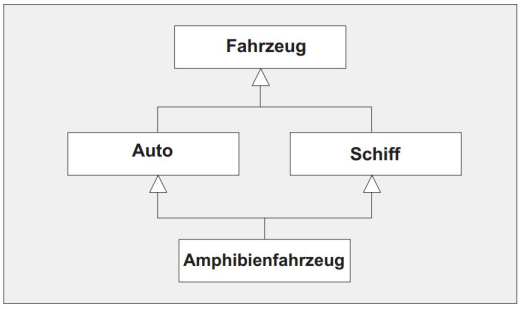

Bild 7.3: Mehrfachvererbung

7.3.4 Typkompatibilität und -konvertierung

Unter Typprüfung versteht man, dass für jede Funktionalität vorgegeben sein muss, von welchem Typ die beteiligten Operanden sind. Dann kann der Compiler schon zur Compile-Zeit überpüfen, ob die Argumente den geforderten Typen entsprechen. Die Erfahrung lehrt, dass bei Sprachen, die eine strenge Typprüfung vorweisen können, mehr Fehler bereits zur Compile-Zeit erkannt werden und sich nicht erst zur Laufzeit herausstellt, dass die Argumente für die Operation nicht passend sind.

Upcasts

Jede Klasse und jedes Interface, das wir einführen, führt einen neuen Typ ein, den wir dann in Argumenten von Funktionen verwenden können. Wenn eine Klasse mehrere Interfaces implementiert, dann steht sie in mehreren **ist ein**-Beziehungen und kann damit an verschiedenen Stellen unter einem anderen Gesicht (=Interface) auftreten. Wo es vorher nicht möglich war, eine Variable von einem anderen Typ zu übergeben als dem angegebenen, kann in objektorientierten Sprachen ein `TransportVorgang` auch überall dort verwendet werden, wo ein `Vorgang` gefragt ist. Aufgrund der **ist ein**-Beziehung kann der Compiler es erlauben, ohne Typecast einen `Transportvorgang` als einen `Vorgang` einzusetzen, ein `Transportvorgang` ist **aufwärtskompatibel** zu `Vorgang`. Der Begriff der *Aufwärts*kompatibilität kommt aus der Darstellung in einer Klassenhierarchie, bei der die **ist ein**-Beziehungspfeile meist von unten nach oben zeigen. Eine Typkonvertierung *aufwärts in der Hierarchie* (ein so genannter **upcast**) ist unproblematisch und wird automatisch durchgeführt.[2]

[2] Ein Nachtrag zu Bild 7.1: Wegen des fehlenden automatischen Upcasts war es in unserer prozeduralen Lösung erforderlich, den Weg auf der linken Seite im Bild zu gehen, weil auf der rechten Seite ein `Transportvorgang` einen `Vorgang` *hat*, aber keiner ist. Damit wäre eine Funktion `void f(Vorgang* v)` für einen `Transportvorgang` nicht aufrufbar, eine Typkonvertierung wäre erforderlich. Wie gesagt, in objektorientierten Sprachen sorgt dafür der Compiler.

Downcasts

Unter einem **Downcast** versteht man hingegen einen Typecast vom allgemeineren zum spezielleren Typ in der Vererbungshierarchie, etwa von `Vorgang` nach `Transportvorgang`. Solche Konvertierungen sind grundsätzlich als kritisch anzusehen, aber in manchen Fällen doch zwingend erforderlich. Wenn wir eine Referenz `v` auf einen `Vorgang` haben, dann gibt es keinen Hinweis darauf, ob es sich bei dem referenzierten Objekt um einen Transport- oder einen Produktionsvorgang handelt. Eine Typkonvertierung der Referenz `v` *verändert nicht* das durch `v` referenzierte Objekt, sondern nur den Typ der Referenz `v` und damit die Menge der Methoden, die wir für das Objekt aufrufen dürfen. Wie soll das System reagieren, wenn ein Downcast auf einen `Transportvorgang` durchgeführt wird, es sich aber tatsächlich um einen `Produktionsvorgang` handelt? Ein `Produktionsvorgang` lässt sich zwar als `Vorgang`, aber natürlich *nicht* als `Transportvorgang` interpretieren, daher ist dieser Downcast unzulässig und führt zu einem Fehler (Ausnahme in Java und undefiniertes Verhalten in C++).

Downcasts in objektorientierten Sprachen zu verbieten, würde aber zu sehr einschränken: Wir haben schon an einigen Stellen gesehen, dass wir in Schnittstellen sehr allgemeine Typen (in Java: `Object`) verwenden müssen, etwa in der Schnittstelle `Comparable` oder wenn wir sowohl eine Liste von Transport- als auch Produktionsvorgängen benötigen. Dann bietet es sich an, einmal eine Liste für Vorgänge zu implementieren und dann in die eine Instanz nur Transport- und in die andere nur Produktionsvorgänge einzufügen. Aus dem Kontext ist dann unmittelbar klar, dass die eine Liste nur Transport- und die andere nur Produktionsvorgänge enthält, ein Downcast ist ohne Laufzeitfehler möglich.

Allgemeine Typkonvertierungen

Konvertierungen sind aber nicht nur bei den (komplexen) strukturierten Typen möglich, sondern können bereits bei den Standardtypen wie `double` und `int` viele Probleme lösen, aber auch aufwerfen. Wir haben bereits in den vorangegangenen Kapiteln gesehen, dass der Compiler an einigen Stellen automatische Typkonvertierungen durchführt, aber die Erfahrung gebietet es, dass die Anweisung `int i = 4.5;` vom Compiler trotz automatischer Konvertierung mit einer Warnung, z.B.: `converting to 'int' from 'double'`, bedacht wird – die Warnung weist in diesem Beispiel auf einen wahrscheinlichen Programmierfehler hin, weil die Typen sich nicht entsprechen. Erst wenn wir die Typkonvertierung selbst vornehmen (im einfachsten Fall: `int i = (int) 4.5;`), sieht der Compiler von einer Warnung ab.

C++ und Java unterscheiden die impliziten und die expliziten Typkonvertierungen; siehe Anhang 1.6. Die expliziten Typkonvertierungen erfolgen durch Voranstellen des Zieltyps vor eine Variable bzw. vor den zu konvertierenden Ausdruck. Das folgende Code-Fragment zeigt zwei Beispiele:

```
double d = 14.3;
int i= (double) d; // bzw. double (d)

Vorgang* pv = new TransportVorgang;
((TransportVorgang*) pv)->getDauer();
```

```
double d = 14.3;
int i= (double) d;

Vorgang v = new TransportVorgang;
((TransportVorgang) v).getDauer();
```

Die beiden Beispiele unterscheiden sich in ihrer Semantik sehr. Im ersten Fall findet eine Konvertierung der Genauigkeit statt, im zweiten Fall ein Downcast. C++ bietet nun im Unterschied zu Java mit den Schlüsselwörtern `static_cast`, `const_cast`, `reinterpret_cast` und `dynamic_cast` die Möglichkeit, diese semantischen Unterschiede auch syntaktisch hervorzuheben.

Ihre Verwendung ist die bessere Alternative zu den unauffälligen Allzweckkonvertierungen der Form `(Zieltyp)Ausdruck`, da aus ihnen jeweils der Zweck der Konvertierung unmittelbar ersichtlich ist und sie auffälliger sind.

Der static_cast-Operator in C++

C++

Der Operator **static_cast** erlaubt neben der Konvertierung zwischen verwandten elementaren Datentypen, wie z.B. zwischen `int` und `double`, die Konvertierung eines *nackten Zeigers* vom Typ `void*` in einen typisierten Zeiger:

```
void* ptr = ...;
XYZ* zeiger = static_cast<XYZ*>(ptr);
```

Der const_cast-Operator in C++

C++

Der Operator **const_cast** erlaubt es, ein konstantes Objekt in ein nicht konstantes Objekt zu konvertieren, z.B.:

```
const int size = 100;
int* ptrSize = const_cast<int*>(size);
*ptrSize     = 150; // d.h. size erhält den Wert 150
```

Er ist der einzige C++-Konvertierungsoperator, der ein `const`-Attribut einer Variablen *wegcasten* kann. Auf den ersten Blick scheint seine Verwendung für einen guten Programmierer überhaupt keinen Sinn zu ergeben, doch das Gegenteil ist der Fall: Er erlaubt dem gewissenhaften Programmierer überhaupt erst, den Code nachlässiger Entwickler zu verwenden, wie das folgende Beispiel zeigt:

```
// Ausgabe des Vornamens und des Nachnamens
void printName(string& name, string& vname) {
  cout << vname << " " << name << ";";
}

struct Name{
  string vname;
  string nname;
};
// Ausgabe der Namen aller Vereinsmitglieder
void printVerein(const vector<Name>& verein) {
```

```
  for (vector<string>::size_type i=0; i < verein.size(); ++i) {
    printName(verein[i].nname, verein[i].vname);
  }
}
```

Dieser Code ist nicht übersetzbar, weil der konstante Parameter `verein` nicht an die Funktion `printName` übergeben werden kann, denn der Programmierer dieser Funktion hat vergessen, die beiden Parameter als konstante Parameter zu spezifizieren. Damit der gewissenhafte Entwickler von `printVerein` die Funktion `printName` überhaupt nutzen kann, muss er den `const_cast`-Operator verwenden.

```
void printVerein(const vector<Name>& verein) {
  for (vector<string>::size_type i=0; i < verein.size(); ++i) {
    printName(const_cast<string&>(verein[i].nname),
              const_cast<string&>(verein[i].vname));
  }
}
```

Der `dynamic_cast`-Operator in C++

C++

Der Operator **`dynamic_cast`** wird in polymorphen Strukturen zum *Downcast* von einem Basistyp in einen abgeleiteten Typ verwendet. Er überprüft als einziger, ob die Konvertierung auch erfolgreich ist. Im Fehlerfall wird in einen `NULL`-Zeiger konvertiert oder im Falle der Konvertierung von echten C++-Referenzen im Fehlerfall eine Ausnahme vom Typ `bad_cast` erzeugt.

In Java gibt es ganz entsprechend das Schlüsselwort `instanceof`. Mit `instanceof` kann zur Laufzeit festgestellt werden, ob ein definiertes Objekt vom Typ einer Klasse ist.

```
void f(Vorgang* v) {
  if (dynamic_cast
        <ProduktionsVorgang*>(v)) {
    ...
}
```

```
void f(Vorgang v) {
  if (v instanceof ProduktionsVorgang) {
    ...
}
```

Der `reinterpret_cast`-Operator in C++

Der Operator **`reinterpret_cast`** erlaubt es, einen Zeiger auf einen beliebigen Typ in einen Zeiger auf einen beliebigen anderen Typ zu konvertieren. Der aus C bekannte Allzweck-Konvertierungsoperator `(Zieltyp)Ausdruck` erlaubt das natürlich *leider* auch.
Beispiel:

```
typedef struct{char b1, b2, b3, b4} BYTES;
int   ix = 1111;
BYTES* pbx = reinterpret_cast<BYTES*>(&ix);
// Die Bytedarstellung von ix kann nun ausgegeben werden.
```

Gegenüberstellung von `reinterpret_cast`, `dynamic_cast` und `static_cast`

Der Operator `reinterpret_cast` ist der *gefährlichste* aller Konvertierungsoperatoren, da er die Konvertierung zwischen beliebigen nicht verwandten Zeigertypen ermöglicht! `reinterpret_cast` liefert immer den gleichen Wert zurück wie sein Argument, d.h. die übergebene Adresse, nur der Typ der Referenz/des Zeigers ändert sich. Für die beiden Operatoren `static_cast` und `dynamic_cast` gilt dies nicht unbedingt. Die Anwendung aller drei Operatoren kann im konkreten Fall syntaktisch richtig sein, allerdings zur Laufzeit durchaus zu unterschiedlichen Ergebnissen führen, wie das folgende Beispiel zeigt:

```
class ReifeVorgang{int i, j, k;};
class LagerVorgang{double d;};
class Veredelung : public ReifeVorgang, public LagerVorgang {double x;};

  Veredelung* pve = new Veredelung;
  LagerVorgang* plv2 = reinterpret_cast<LagerVorgang*>(pve);
  cout << "Org " << pve << " Cast " << plv2 << endl;

  LagerVorgang* plv1 = static_cast<LagerVorgang*>(pve);
  cout << "Org " << pve << " Cast " << plv1 << endl;

  LagerVorgang* plv3 = dynamic_cast<LagerVorgang*>(pve);
  cout << "Org " << pve << " Cast " << plv3 << endl;
```

Die Ausgabe könnte hier z.B. wie folgt sein.

```
Org 00322EB0 Cast 00322EB0 // reinterpret_cast--> beide gleich
Org 00322EB0 Cast 00322EC0 // static_cast --> unterschiedlich
Org 00322EB0 Cast 00322EC0 // dynamic_cast --> unterschiedlich
```

Der `dynamic_cast`-Operator wäre der einzige Operator, der zur Laufzeit herausfinden könnte, ob `pve` auch wirklich auf einen `LagerVorgang` zeigt. Wenn das nicht der Fall wäre, würde er als einziger den `NULL`-Zeiger liefern bzw. eine `bad_cast`-Ausnahme werfen.

7.3.5 Die Verwendung von Vererbung

Wir haben Vererbung hier so motiviert, dass bei Vorliegen einer **ist ein**-Beziehung eine Vermeidung von Code-Duplikation über Copy & Paste auf jeden Fall wünschenswert ist und dass Vererbung ein probates Mittel dafür darstellt. Wir haben *nicht* gesagt, dass jede Art von Code-Duplikation über Vererbung vermieden werden kann oder soll.

Ein Beispiel mag diesen Sachverhalt verdeutlichen.

Peter: Peter schreibt folgende Klasse:

```
class Quadrat {
  protected double kantenlaenge;
  protected double positionx,positiony;
```

```
  public void setBreite(double k) { kantenlaenge=k; }
  public double getBreite() { return kantenlaenge; }
  public void setHoehe(double k) { kantenlaenge=k; }
  public double getHoehe() { return kantenlaenge; }
  public double getFlaeche() { return kantenlaenge*kantenlaenge; }
  public double verschiebe(int x,int y) { positionx+=x; positiony+=y; }
}
```

Peter erhält ein *Rechteck* aus seinem `Quadrat`, indem er einfach eine weitere Kantenlänge hinzufügt. Etwa wie folgt:

```
class Rechteck extends Quadrat {
  protected double kantenlaenge2; // als Hoehe

  public void setHoehe(double k) { kantenlaenge2=k; }
  public double getHoehe() { return kantenlaenge2; }
  public double getFlaeche() { return kantenlaenge*kantenlaenge2; }
}
```

Die Klasse `Rechteck` erbt dann von `Quadrat` die Positionsattribute und die `verschiebe`-Funktion, die beim `Rechteck` genauso funktioniert wie beim `Quadrat`.

Paul: Paul geht genau umgekehrt vor, er beginnt mit einem `Rechteck`:

```
class Rechteck {
  protected double kantenlaenge,kantenlaenge2;
  protected double positionx,positiony;

  public void setBreite(double k) { kantenlaenge=k; }
  public double getBreite() { return kantenlaenge; }
  public void setHoehe(double k) { kantenlaenge2=k; }
  public double getHoehe() { return kantenlaenge2; }
  public double getFlaeche() { return kantenlaenge*kantenlaenge2; }
  public double verschiebe(int x,int y) { positionx+=x; positiony+=y; }
}
```

Ein Quadrat ist für Paul ein Spezialfall eines Rechtecks, daher argumentiert er, dass die zweite Kantenlänge immer gleich der ersten ist. Er leitet `Quadrat` von `Rechteck` ab:

```
class Quadrat extends Rechteck {
  public void setBreite(double k) { kantenlaenge2=kantenlaenge=k; }
  public void setHoehe(double k) { kantenlaenge2=kantenlaenge=k; }
}
```

Dabei erbt er den Großteil der Funktionalität von `Rechteck`, nur die Nebenbedingung muss er in den set-Methoden garantieren.

Peter und Paul streiten sich nun darum, wer es richtig gemacht hat. Paul behauptet, Peters Lösung wäre falsch, er müsse doch nur mal überlegen: Es ist eben *nicht* jedes Rechteck ein Quadrat! Wohl aber ist jedes Quadrat ein Rechteck! Daraufhin experimentiert Peter mit Pauls Lösung und konfrontiert ihn mit folgender Funktion: Sie soll die Fläche eines Rechtecks verdoppeln, indem jede Kantenlänge mit $\sqrt{2}$ multipliziert wird:

```
void f(Rechteck x) {
  x.setBreite(x.getBreite()*Math.sqrt(2));
  x.setHoehe(x.getHoehe()*Math.sqrt(2));
}
```

Diese Funktion funktioniert mit Pauls Rechteck wunderbar. Wenn Peter aber Pauls Quadrat (Quadrat ist ein Rechteck) in die Funktion einsetzt, dann liefert die Funktion falsche Ergebnisse. Eine anfängliche Kantenlänge von 2 wird nach der ersten Anweisung zu $2 \cdot \sqrt{2}$, und die zweite Anweisung führt zu einer Kantenlänge von 4 – und damit zu einer Vervierfachung der Fläche. Bei seiner Lösung, so Peter, würde die Funktion mit Rechtecken funktionieren, aber für Quadrate könne man sie nicht aufrufen, daher wäre seine Lösung besser. Paul kontert, dass bei Peters Lösung alle Funktionen mit Quadraten als Argument auch mit Rechtecken aufgerufen werden könnten: Man könne leicht Beispiele finden, bei denen die Funktion mit einem übergebenen Rechteck fehlschlägt.

Wir blenden die fruchtlose Diskussion an dieser Stelle aus und halten fest: Vielleicht ist der Einsatz der Vererbung nicht so einfach, wie im ersten Moment gedacht. Wenn wir bei einem so einfachen Beispiel schon so viel diskutieren, hilft uns die Vererbung dann eigentlich?

Die Antwort lautet: Weniger als allgemein angenommen. Das wichtigere Element der Objektorientierung ist die Polymorphie, nicht die Vererbung! Beim Einsatz von Vererbung wird auch immer eine **ist ein**-Beziehung etabliert. In der praktischen Anwendung wird aus Effizienzgründen auch ab und zu Vererbung eingesetzt, wenn eine solche **ist ein**-Beziehung eigentlich gar nicht vorliegt (siehe Peter). Dass die Variante von Paul auch nicht gut funktioniert, liegt an der Schnittstelle. Dabei ist zu beachten, dass die Schnittstelle nicht nur aus den Methodendeklarationen, sondern auch aus der Semantik/Spezifikation besteht. In der gemeinsamen Schnittstelle (`Rechteck`) gibt es Methoden, deren Semantik sich zwischen Rechteck und Quadrat stark unterscheidet: Bei `setBreite(double k)` wird einmal nur die Breite und einmal die Kantenlänge (Länge und Breite) auf `k` gesetzt, was semantisch nicht gleichwertig ist. Wenn wir die Semantik dieser Funktionen vorher eindeutig definiert hätten, hätten wir auch sagen können, ob beide Klassen die Schnittstelle erfüllen. Ohne Spezifikation nur auf die *selbsterklärenden Methodennamen* zu bauen, ist risikoreich und führt zu den oben geschilderten Problemen. Dabei hätte ein einfacher Test für das Interface von `Rechteck` genügt, um diese Problematik zu entlarven:

```
boolean test1(Rechteck r) {
  r.setBreite(2);
  r.setHoehe(2);
  if (r.getFlaeche()!=4) return false;
  r.setHoehe(4);
  if (r.getFlaeche()!=8) return false;
  r.setBreite(4);
  if (r.getFlaeche()!=16) return false;
  return true;
}
```

Dies ist nur ein sehr einfacher Test, aber auch einfache Tests sind viel besser als keine Tests. Insbesondere muss dieser Test mit *allen* Rechtecken funktionieren. Wenn Paul durch die Vererbung auch ausdrückt, dass ein Quadrat ein Rechteck ist, dann können wir diese Aussage prüfen, indem wir `test1` für ein Quadrat aufrufen – und der Test schlägt fehl; ein Quadrat erfüllt die Spezifikation der Rechteck-Schnittstelle nicht!

Wie sehen die Auswege aus? Erstens kann man Code-Duplikation auch anders vermeiden, bspw. durch eine *benutzt-* oder **hat ein**-Beziehung statt einer **ist ein**-Beziehung.

```
class Rechteck {
  private Quadrat quadrat;
  private double kantenlaenge2;
  // ...
  public double getFlaeche() { return quadrat.getKantenlaenge()*kantenlaenge2; }
  public double verschiebe(int x,int y) { quadrat.verschiebe(x,y); }
}
```

Wenngleich die Vorstellung, ein Rechteck bestünde aus einem Quadrat und einer weiteren Kantenlänge, eher zweifelhaft ist, so ist das in anderen Situationen nicht unbedingt der Fall und – viel wichtiger – wir erkaufen uns die vermiedene Code-Duplikation nicht durch eine ungewollte **ist ein**-Beziehung. Zweitens können wir die gemeinsame Schnittstelle auf ein Maß reduzieren, bei dem keine Konflikte mehr auftreten. Zum Beispiel reduzieren wir die Rechteck-Schnittstelle um alle setter-Methoden (und definieren damit ein Interface `GeomFigur`), dann kann man immer noch die Größe und Fläche abfragen, sogar den Schwerpunkt der geometrischen Figur verschieben. Das Setzen von Breite und Höhe (im Falle eines Rechtecks `Rechteck implements GeomFigur`) oder einer Kantenlänge (im Falle eines Quadrats `Quadrat implements GeomFigur`) sind nicht allgemein genug, sondern spezifisch für die speziellen geometrischen Figuren. Wenn nötig, kann man als weitere Zwischenstufe in der Hierarchie noch das Interface `Viereck` einfügen, von dem dann sowohl `Rechteck` als auch `Quadrat` abgeleitet werden.

> Vererbung ist in der Praxis oft durch die Vermeidung von Code-Duplikation motiviert. Leider lässt sich die Code-Duplikation von der Deklaration der **ist ein**-Beziehung nicht trennen. Ohne eine Spezifikation der Schnittstellensemantik ist es bereits bei einfachen Beispielen nicht trivial, sicherzustellen, ob alle Realisierungen konform zueinander sind. Tests können hier die (partielle) Überprüfung der Semantik übernehmen.

Das Konzept *Vererbung* wird von vielen Sprachen unterstützt. Es ist aber festzustellen, dass die Verwendung der Vererbung (zumindest in Java und C++) uneinheitlich ist. Es gibt hier unterschiedliche Standpunkte zu ihrer Anwendung, z.B.:

1) *Klassen* und *Vererbung* sind Sprachkonzepte, die so eingesetzt werden, wie es gerade praktisch ist (**Pragmatik**).
2) Eine *Klasse* ist ein Konzept zur Realisierung eines *abstrakten Datentyps*. *Vererbung* wird verwendet, um aus einer Realisierung des abstrakten Datentyps eine neue Realisierung abzuleiten, die zur ursprünglichen Klasse in einer **„ist ein (is a)“**-Beziehung steht (**Spezialisierung**).

3) Wie 2), doch mit der Zusatzbedingung, dass die abgeleitete Klasse eine Erweiterung der ursprünglichen Klasse (d.h. der Basisklasse) darstellt. Damit beinhaltet die abgeleitete Klasse die Basisklasse und kann an die Stelle der Basisklasse treten (**Typerweiterung, Subtyping**).

Gesichtspunkt 1) beinhaltet keine weiteren Entwurfsrichtlinien; er erlaubt, was die Sprache hergibt. Entwurfsrichtlinien (design rules) dienen aber dem Zweck, Programme durchschaubarer, sicherer und wartbarer zu machen, und darauf sollte gerade bei Verwendung des komplexen Konzeptes *Vererbung* nicht verzichtet werden!

Gesichtspunkt 3) kommt von N. Wirth, der aus dieser Sicht die Sprache Oberon als Nachfolgerin von Pascal und Modula entwickelt hat. Typerweiterung bedeutet, dass eine abgeleitete Klasse die Oberklasse als Untermenge enthält, und die Ableitung fügt etwas hinzu. Das Konzept Typerweiterung ist relativ einfach und klar, und es kann auch in Java und C++ als *eingegrenzte Sicht der Vererbung mit einer klaren Semantik* verwendet werden.

Gesichtspunkt 2) ist wesentlich umfassender, aber auch komplexer in der Anwendung. Er hält sich im Grunde nur an die Semantik, die durch die UML-Notation und die Umsetzung in eine objektorientierte Sprache vorgegeben ist. Wenn `B` von `A` abgeleitet ist, dann erlaubt es nun einmal die Sprache, `B` überall dort einzusetzen, wo ein `A` erwartet wird (automatischer Upcast, vgl. Abschn. 7.3.4).

In der Praxis scheint es, als wäre Gesichtspunkt 3) ein besserer Ratgeber für den Einsatz von Vererbung: Ein `Quadrat` stellt keine Erweiterung eines `Rechtecks` dar, denn es kommen keine Attribute hinzu, also sollte Vererbung hier nicht verwendet werden. Aber es kann auch subtilere Fälle geben: Ein `SchnellerVorgang` ist ein `Vorgang` – er fügt aber kein Attribut hinzu, also keine Vererbung. Ein `SchnellerTransportVorgang` ist ein `Vorgang`, der etwas hinzufügt (nämlich Attribute für den Transport), also wäre Vererbung angebracht? Nein, bei `SchnellerTransportVorgang` handeln wir uns dieselben Probleme ein wie bei `SchnellerVorgang`, es ist also nicht allein die Frage entscheidend, ob Attribute hinzukommen. Entscheidend ist, ob eine *Einschränkung* oder *Bedingung* an die Oberklasse gestellt wird, unabhängig davon, ob Attribute hinzukommen oder nicht. Solche Bedingungen wirken sich potentiell auf alle Methoden der Oberklasse aus, die von einer zusätzlichen Bedingung ja nichts wissen. Sie einfach unverändert zu übernehmen (zu erben) ist gefährlich, denn das Ergebnis der Methode hält sich nicht unbedingt an die neu gestellte Bedingung. Also sollte in solchen Fällen von Vererbung abgesehen werden.

> Bei Ableitungen von konkreten Klassen können Probleme auftreten. Deshalb sollte entweder nur von Interfaces (oder abstrakten Klassen) abgeleitet werden, oder es sollte sich um eine echte Typerweiterung einer konkreten Klasse handeln (ohne Einführung von Bedingungen).

7.3.6 Gegenüberstellung: Templates und Polymorphie

Eine Schablone parametrisiert die Deklaration eines Typs oder einer Funktion mit einem anderen Typ. Der Code, der die Schablone implementiert, ist identisch für alle

konkreten Werte der Parametertypen. Bei Verwendung von Polymorphie und dynamischer Bindung implementiert eine Klasse ein Interface. Code für verschiedene Implementierungen der abstrakten Klasse kann in einer Klassenhierarchie gemeinsam genutzt werden, und der Code, der die Schnittstellen der Interface-Klasse verwendet, ist unabhängig von der konkreten Implementierung des Interfaces. Das folgende Listing zeigt die Ähnlichkeit beider Ansätze: Alle Elemente des Arguments vom Typ `Stack` werden gelöscht, einmal jedoch ist `Stack` ein Template-Parameter und das andere Mal ein Interface. In beiden Fällen können unterschiedliche Realisierungen eines Stacks von der Funktion benutzt werden (für das rechte Listing setzen wir den Stack aus Abschn. 6.2.4 voraus). Daher werden sie beide als Polymorphismus bezeichnet. Um sie unterscheiden zu können, spricht man im Zusammenhang mit dem dynamischen Binden vom *Laufzeit-Polymorphismus* und im Zusammenhang mit Templates vom *Compilezeit-Polymorphismus*.

```
template <class Stack>
void clearStack(Stack s) {
  while (!s.isEmpty()) s.pop();
}
```

```
void clearStack(Stack s) {
  while (!s.isEmpty()) s.pop();
}
```

In rein objektorientierten Sprachen wie Smalltalk, Eiffel und Java ist **Laufzeit-Polymorphismus** die einzige Möglichkeit, um generische und heterogene Strukturen zu entwickeln (Generics fügen Java keinen Compilezeit-Polymorphismus hinzu). Eine typische generische Struktur ist in dem Zusammenhang ein Container (z.B. eine Liste, ein Array oder ein Stapel), der wahlweise mit verschiedenen Typen von Elementen gefüllt sein kann, wobei alle Elemente eines Containers aber jeweils den gleichen Typ haben: *Der Inhalt ist homogen.* Eine typische heterogene Struktur ist demgegenüber ein Container, der gleichzeitig mit Elementen verschiedenen Typs gefüllt sein kann: *Der Inhalt ist heterogen.*

Die klassische polymorphe Realisierung mit dynamischer Bindung zur Laufzeit verwendet eine abstrakte Basisklasse und eine umfangreiche Hierarchie davon abgeleiteter Klassen mit virtuellen Funktionen zur Definition von Elementtypen und von Containertypen; gefährliche *Downcasts* sind unvermeidbare Konstruktionselemente, die Typüberprüfung erfolgt zur Laufzeit.

Die alternative Realisierung auf der Basis von Templates kann vollständig auf die Verwendung von virtuellen Funktionen verzichten; es wird streng statisch typisierter Code generiert, d.h. die Typüberprüfung findet zur Compile-Zeit statt (siehe Abschn. 7.3.6); ein weiterer Vorteil sind die einfachere Benutzbarkeit und in C++ die bessere Laufzeiteffizienz. Die *Standard Template Library (STL)* ist eine inzwischen weithin bekannte Vertreterin dieses *generischen Ansatzes* (siehe auch C++-Kompendium [30, Kap. 11]).

Im Falle der heterogenen Struktur gibt es keine Alternative zum Laufzeit-Polymorphismus; im Falle der generischen Struktur bietet sich in C++ in vielen Fällen der Compile-Zeit-Polymorphismus in der Form typparametrisierter Klassen als Alternative an.

Man könnte nun meinen, dass sich die Java-Lösung mit Templates (Listing 6.17, Abschn. 6.3.1) nicht von einer Lösung mit Polymorphie, wie sie Listing 7.11 zeigt, unterscheidet.

Java 7.11: Polymorphe Klasse `PolyStapel` mit Anwendungsprogramm

```
public class PolyStapel {
  Object[] data;
  private int top, size;

  public Stapel(int s, Object x) {
    data = new Object[s];
    size = s; top = 0;
    }
  public void push(Object x) {
     data[top++] = x;}
  public Object pop() {
     return data[--top];}
  public boolean isEmpty () {
     return top == 0;}
}
```

(Poly/Stack/PolyStapel.java)

```
public class DemoTemp {
  public static void main(String[] args) {
    PolyStapel s1=new PolyStapel(3);

    for (int i=0; i<3; i++) {
       Double d = i * 2.5; s1.push(d);
     }

    while ( ! s1.isEmpty() ) {
      System.out.print(s1.pop()+" ");
    }
    // jetzt kein Syntaxfehler !!!
    String n="AB"; s1.push(n);
  }
}
```

(Poly/Stack/PolyDemoTemp.java)

Allerdings findet in der polymorphen Lösung keine Prüfung zur Compile-Zeit statt. Es gibt nicht einmal einen Laufzeitfehler. Der `Stapel` ist hier somit quasi *doppelt generisch*: Er kann zum einen verschiedene Objekttypen aufnehmen und dies sogar gleichzeitig. In C++ gibt es keine Basisklasse wie `Object`, von der sich alle anderen Klassen ableiten, sodass die polymorphe Lösung in C++ so nicht möglich wäre. Man könnte sich allerdings mit nicht-typisierten Zeigern (`void*`-Zeigern) behelfen.

Benötigen wir unsere Netzplanung lediglich zur Planung von `Transportvorgängen` oder zur Planung von `Produktionsvorgängen`, aber niemals zur Planung von beiden gleichzeitig, so wäre die Verwendung einer Schablonenklasse `Netz` eine gute Wahl.

```
template <typename V>
class Netz {
 private:
   /* ... */
   V* vorg[MAX]; // Knoten
 public:
   void fuegeHinzu(V *v);
   const V& getVorgang(int i) const;
   /* ... */
};
```

```
public class Netz<V> {
 /* ... */
 private V vorg[] =
          (V[]) new Object[MAX];
 public void fuegeHinzu(V v) { /* ...*/ }
 public V getVorgang(int i) { /* ...*/ }
 /* ... */
```

Benötigen wir dagegen sowohl `Transportvorgänge` als auch `Produktionsvorgänge` gleichzeitig, so müssen wir Laufzeit-Polymorphismus verwenden.

Bei Programmiersprachen wird häufig *Orthogonalität* oder auch *Disjunktivität* als wichtiges Qualitätskriterium angesehen, was bedeutet, dass es im Wesentlichen für jede Aufgabe ein passendes Sprachkonstrukt (nicht mehrere) geben sollte. Damit stellt sich hier die Frage: *Wann ist Compile-Zeit-Polymorphismus gegenüber Laufzeit-Polymorphismus zu bevorzugen und wann nicht?*

- Templates sind gegenüber Interfaces zu bevorzugen, wenn Laufzeiteffizienz ganz **Tipp**

entscheidend ist (z.B. in Echtzeitanwendung; in betrieblichen Anwendungen spielt der Unterschied keine Rolle).

- Interface-Klassen sind zu bevorzugen, wenn neue Varianten ohne erneute Übersetzung hinzugefügt werden sollen.

C++ In C++ sind Templates im Gegensatz zur Polymorphie auch noch einsetzbar, wenn keine gemeinsame Basisklasse existiert, oder dann, wenn auch Standardtypen (z.B. `int`, `double`) als Schablonenparameter eingesetzt werden sollen.

Einschränkung der Schablonenparameter

Im Abschn. 2.3 haben wir generische Funktionen kennengelernt. Auf Seite 52 wurde eine generische Variante für den *Bubblesort*-Algorithmus vorgestellt. Dieser Algorithmus ist allerdings nur auf Typen anwendbar, bei denen zwei Instanzen dieser Typen miteinander verglichen werden können.

In C++ geschieht dies durch den Vergleichsoperator, beim Compilieren merkt der C++-Compiler, ob ein solcher verfügbar ist. Ist das nicht der Fall, d.h. wird `bubbleSort` z.B. für ein Array vom Typ `Vorgang` aufgerufen, ergibt sich in C++ entweder sofort ein Syntaxfehler, oder spätestens der Linker erkennt ein Fehlen der beiden Operatoren.

In Java müssen die `compareTo`- und `equals`-Methoden existieren und sinnvoll überladen sein. Um auch in Java den Bubblesort generisch schreiben zu können, müssen wir diese Eigenschaft des Typparameters (Existenz von `compareTo`) bereits beim Schablonenparameter mit angeben:

```
public static <T extends Comparable>
void bubbleSort(int beg,int st, T[] f){
/* . . . */
}
```

Jetzt wird bereits der Compiler einen Fehler melden, wenn `bubbleSort` mit einem Typ aufgerufen wird, der nicht die Schnittstelle `Comparable` implementiert.

Die Einschränkung des Typs mit dem Schlüsselwort `extends` ist sowohl bei Funktions- als auch bei Klassendeklarationen möglich. Soll der konkrete Typ zu mehreren Typen passen, d.h. mehrere Schnittstellen implementieren, so lassen sich mit `&` noch weitere Oberklassen hinzunehmen, z.B. `public class X <T extends O1 & O2 & O3> { ... }`. Hierbei darf entsprechend dem Konzept der Mehrfachvererbung (Abschn. 7.3.3) nur einer der Parameter O_i eine Klasse sein, bei den restlichen Einschränkungen muss es sich um Interfaces handeln.

7.3.7 Übungen

C++ **Übung 7.1:**

Gegeben sind die folgenden Header-Dateien `Figur.h`, `Kreis.h` und `Ring.h`.

```
#include <fstream>
using namespace std;
extern ofstream datei;

class Figur {
  int dummy;
public:
  Figur() {dummy=0; datei << "+F";}
  ~Figur() {datei << "-F";}
  void Umfang() const {datei<<"F"<< 0;}
  virtual void Flaeche()const=0;
  };
```

(./Poly/KonstrDestrPoly/Figur.h)

```
#include "Figur.h"
class Kreis: public Figur {
  protected:
    enum {PI=3};
    int rad; // Außenkreis-Radius
  public:
    Kreis(): rad(4) {
      datei << "+K" << rad << " ";
      }
    Kreis(int r) {
      rad=r;
      datei << "+K" << rad << " ";
      }
    ~Kreis() {
      datei << "-K" << rad << " ";
      }
    void Umfang()const {
      datei << "K" << 2*PI*rad;
      };
    virtual void Flaeche()const {
      datei << "K" << PI*rad*rad;
      };
  };
```

(./Poly/KonstrDestrPoly/Kreis.h)

```
#include "Kreis.h"
// Kreis mit Loch
class Ring: public Kreis {
  int rRad; // Innenkreis-Radius
  public:
    Ring(int r) {
      rRad=r;
      datei << "+R" << rRad << " ";
      }
    Ring(int r, int rr):Kreis(r), rRad(rr) {
      datei << "+R" << rRad << " ";
      }
    ~Ring() {
      datei << "-R" << rRad << " ";
      }
    void Umfang()const {
      datei << "R" << 2*PI*(rad-rRad);
      }
    virtual void Flaeche() const {
      datei<<"R"<<PI*(rad*rad - rRad*rRad);
      }
  };
```

(./Poly/KonstrDestrPoly/Ring.h)

Das zugehörige Anwendungsprogramm ruft nacheinander verschiedene Funktionen auf. Geben Sie bei den folgenden Unteraufgaben an, was durch Aufruf der Funktionen `funk1`, `funk2`, `...` jeweils in die Datei `datei` ausgegeben wird.

```
int main() {
  datei << "\nAufgabe a" << endl;
  funk1(); datei << endl << endl;

  datei << "Aufgabe b" << endl;
  funk2(); datei << endl << endl;

  datei << "Aufgabe c" << endl;
  funk3(); datei << endl << endl;

  datei << "Aufgabe d" << endl;
  funk4(); datei << endl << endl;

  datei << "Aufgabe e" << endl;
  funk5(); datei << endl << endl;

  Kreis k(2);
  Ring ring(5,4);
  datei << "Aufgabe f" << endl;
  funk6(k, ring); datei << endl << endl;
  datei << "Aufgabe g" << endl;
  funk7(k, ring); datei << endl << endl;

  datei << "\nAufgabe h" << endl;
    {
    Ring* r = new Ring(ring);
    datei << "Vor Aufruf" << endl;
    funk8(*r);
    datei << "\nNach Aufruf" << endl ;
    delete r;
    }
    datei << endl << "Ende" << endl;

  datei << "\nAufgabe i" << endl;
  /* Warum geht Folgendes nicht!!!
  Ring r_feld[10];
  Figur f_feld[10];
  */
  }
```

(./Poly/KonstrDestrPoly/ConstrDestr.cxx)

a) Zeichnen Sie zunächst das UML-Klassendiagramm der drei Klassen. Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk1`?

```
void funk1() {
  Kreis k(13); Ring ri(14,2); datei << endl;
  }
```

b) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk2`?

```
void funk2() {
  Kreis k; Ring ri(11); datei << endl;
  }
```

c) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk3`?

```
void funk3() {
  Kreis* k1[5]; datei << " xxx "; k1[4]=NULL; datei << " yyy "; Kreis k2[2];
  }
```

d) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk4`?

```
void funk4() {
  // Heapspeicherverwendung
  Kreis* p = new Kreis(12); delete p;
  }
```

e) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk5`?

```
void funk5() {
  // Heapspeicherverwendung
  Figur* p = new Kreis(12); delete p;
  }
```

f) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk6`? Veranschaulichen Sie zunächst die Speicherbelegung im Stack- und im Heap-Programmspeicher zum Zeitpunkt `/* 1 */` unter Beachtung der beiden Hauptprogramm-Variablen `k` und `ring`, mit denen `funk6` aufgerufen wird.

```
void funk6(const Kreis& k6, const Ring& ring6) {
  k6.Umfang(); datei << " "; k6.Flaeche(); datei<< " "; /* 1 */
  ring6.Umfang(); datei << " "; ring6.Flaeche(); datei << endl;
  }
```

g) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk7`? Veranschaulichen Sie zunächst die Speicherbelegung im Stack- und im Heap-Programmspeicher zum Zeitpunkt `/* 2 */` unter Beachtung der beiden Hauptprogramm-Variablen `k` und `ring`, mit denen `funk7` aufgerufen wird.

```
void funk7(Kreis& k, Ring& ring) {
  Figur* vec[2]; vec[0]=&k; vec[1]=&ring; /* 2 */
  for (int i=0;i<2;i++) {
    vec[i]->Umfang(); datei << " ";
    vec[i]->Flaeche(); datei << " ";
  } }
```

Warum können die beiden Parameter hier im Unterschied zur Funktion `funk6` nicht als `const` übergeben werden?

h) Welche Ausgabe ergibt sich durch Ausführung der Funktion `funk8`, d.h. zwischen den Ausgaben `Aufgabe h` und `Ende` (Ausgaben erfolgen in der Funktion `funk8` **und** in `main`)?
Veranschaulichen Sie zunächst die Speicherbelegung im Stack- und im Heap-Programmspeicher zum Zeitpunkt `/* 3 */` unter Beachtung der Hauptprogramm-Variablen `ring` und `r`.

```
void funk8(Kreis f) {
  f.Umfang(); datei << " "; /* 3 */
  f.Flaeche(); datei << " ";
  }
```

i) Warum kann kein Array erzeugt werden, dessen Elemente Objekte der Klassen `Figur` bzw. `Ring` sind?

j) In zwei Funktionen wird Heap-Speicher angefordert. Wie viel Prozent dieses Heapspeichers werden wieder freigegeben? Begründen Sie Ihre *Rechnung*.

Übung 7.2: Java

Gegeben sind zwei Beispielprogramme. Geben Sie jeweils die Ausgabe an. Prüfen Sie sorgfältig, wann Polymorphie zum Einsatz kommt und wann nicht.

```
class A {
   int i = 2;
   A() { System.out.println("A"); }
   void f(A a) { System.out.println(i); }
}
class B extends A {
   int j = 3;
   B() { System.out.println("B"); }
   void f(A a) { System.out.println(j); }
   void f(B a) { System.out.println(2*j); }
}
class Main1 {
   public static void main(String[] args) {
      A a = new B();
      A b = new A();
      B c = new B();
      a.f(b); b.f(a); c.f(c);
}  }
```

```
interface A {
  public void f(A x);
  public void g(B x);
}
class B implements A {
  public void f(A x) { /* Ausgabe "1" */ }
  public void f(B x) { /* Ausgabe "2" */ }
  public void g(B x) { /* Ausgabe "3" */ }
  public void g(C x) { /* Ausgabe "4" */ }
}
class C extends B {
  public void f(A x) { /* Ausgabe "5" */ }
  public void f(C x) { /* Ausgabe "6" */ }
}
class Main3 {
 public static void main(String[] args) {
  B b=new B(); C c=new C(); A a=c;
  a.f(a); a.f(b); a.f(c); a.g(c); a.g(b);
  b.f(a); b.f(b); b.f(c); b.g(c); b.g(b);
  c.f(a); c.f(b); c.f(c); c.g(c); c.g(b);
 }
}
```

7.4 Zusammenfassung

Wir haben in diesem Kapitel die Austauschbarkeit von verschiedenen Implementierungen derselben Schnittstelle über statische und dynamische Bindung kennengelernt. Diese nützliche Technik verbirgt Komplexität vor dem Benutzer, da er nur die Schnittstelle kennen muss. Wir haben gesehen, wie sich *dynamische Bindung* auch mit prozeduralen Sprachmitteln erreichen lässt und wie wir sie in objektorientierten Sprachen nutzen:

- Bei statischer Bindung ist bereits zur Compile-Zeit bekannt, welche Implementierung benutzt wird. Bei dynamischer Bindung wird die Entscheidung zur Laufzeit von Fall zu Fall getroffen.
- Dynamische Bindung bedeutet nicht, dass immer *„die richtige Methode aufgerufen wird“*, sie erfolgt nur dort, wo über den Punkt- bzw. Pfeiloperator eine (virtuelle) Methode aufgerufen wurde, die mit identischem Prototyp in mehreren Implementierungen überladen wurde.
- In der Klassen- oder Schnittstellenhierarchie werden Upcasts sozusagen automatisch (implizit) vorgenommen (eine Unterklasse kann alles, was die Oberklasse auch kann), Downcasts sind potenziell fehleranfällig und müssen mit Sorgfalt behandelt werden.
- Die Verwendung von Vererbung ist meistens durch die Vermeidung von Code-Duplikation motiviert, man darf aber nicht außer Acht lassen, dass damit auch immer eine **ist ein**-Beziehung etabliert wird.
- Typkonvertierungen und ihr Zweck sollten nicht verschleiert werden. Insbesondere C++ stellt für jeden Zweck einen eigenen Konvertierungsoperator zur Verfügung.

Literaturverzeichnis

Eine kurze Übersicht über die folgenden in alphabetischer Reihenfolge angegebenen Quellen finden Sie am Ende dieses Literaturverzeichnisses.

[1] Gl. Altrogge: *Netzplantechnik*, Oldenbourg, 1996

[2] Heide Balzert: *Methoden der objektorientierten Systemanalyse*, 2. Auflage, Spektrum Akademischer Verlag, 1996

[3] Heide Balzert: *Lehrbuch der Objektmodellierung. Analyse und Entwurf*, 2. Auflage, Spektrum Akademischer Verlag, 2004

[4] Helmut Balzert: *Lehrbuch der Software-Technik – Software-Entwicklung*, Spektrum Akademischer Verlag, 1996

[5] Helmut Balzert: *Lehrbuch – Grundlagen der Informatik – Modulare, objektorientierte und generische Programmierung*, 2. Auflage, Spektrum Akademischer Verlag, 2005

[6] Kent Beck: *Extreme Programming Explained – Embrace Change*, Addison Wesley, 1999

[7] Kent Beck, Martin Fowler: *Extreme Programming planen*, Addison Wesley, 2001

[8] Oliver Böhm: *Aspektorientierte Programmierung mit AspectJ 5*, dpunkt.verlag, 2006

[9] Grady Booch, Ivar Jacobson und James Rumbaugh: *The Unified Software Development Process*, Addison-Wesley, 1999

[10] Frederick P. Brooks: *The Mythical Man-Month – 20th Anniversary Edition*, Addison Wesley, 1995

[11] William J. Brown, Raphael C. Malveau, und Hays McCormick: *Anti-patterns. Refactoring Software, Architecture and Projects in Crisis*, John Wiley & Sons, 1998

[12] Bernd Brügge, Allen H. Dutoit: *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java*, Prentice Hall, 2004

[13] Martin D. Carroll, Margaret A. Ellis: *Designing and Coding Reusable C++*, Addison-Wesley, 1995

[14] Martin D. Carroll, Margaret A. Ellis: *What functions should all classes provide?*; C++ Report Vol.6/No.9, SIGS, 1994

[15] Alistair Cockburn: *Use Cases effektiv erstellen*, MITP Verlag, 2003

[16] Cormen, Leiserson, Rivest, Stein: *Algorithmen – eine Einführung*, Oldenbourg, 2005

[17] J. Dunkel, A. Holitschke: *Softwarearchitektur für die Praxis*, Springer, 2003

[18] Chr. Ebert, C., T. Liedtke, E. Baisch: *Improving Reliability of Large Software Systems*, Annals of Software Engineering, 1999

[19] Jutta Eckstein: *Agile Softwareentwicklung im Großen – Ein Eintauchen in die Untiefen erfolgreicher Projekte*, dpunkt.verlag, 2004

[20] Stefan Edlich: *Ant: Kurz & gut*, O'Reilly, 2002

[21] Martin Fowler: *Analysemuster. Wiederverwendbare Objektmodelle*, Addison-Wesley, 1999

[22] Martin Fowler: *Refactoring – Improving the Design of Existing Code*, Addison Wesley, 2002

[23] Erich Gamma, Richard Helm, Ralph E. Johnson, John M. Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley, 1996

[24] J. Gosling, B. Joy, G. Steele, G. Bracha: The Java Language Specification, Sun Microsystems: Java 2 Platform, Standard Edition, Version 5.0, API Specification, `http://java.sun.com/docs/books/jls/`

[25] Güting, Dieker: *Datenstrukturen und Algorithmen*, Teubner, 2003

www [26] Hartmut Helmke et al.: *Guide to survive with C++ – Rules, recommendations and tips for the advanced C++ programmer*, Deutsches Zentrum für Luft- und Raumfahrt, DLR-IB 112-2002/07, 2002

[27] Knut Hildebrand, Jan-Armin Reepmeyer: *Repeat-Statement considered harmful? Ergebnisse einer empirischen Untersuchung*, Informatik Spektrum, Band 19, Heft 2, April 1996

[28] Steve Holzner: *Ant: The Definitive Guide*, O'Reilly, 2005

[29] C.S. Horstmann, G. Cornell: *Core Java – Band 1 und 2*, Markt & Technik, 2002

[30] Rolf Isernhagen, Hartmut Helmke: *Softwaretechnik in C und C++ – Das Kompendium – Modulare, objektorientierte und generische Programmierung*, 4. Auflage, Hanser, 2004

[31] ISO/IEC 9899: *Programming Language – C*, International Organization for Standardization, 1999

[32] ISO/IEC 14882: *Programming Language – C++*, International Organization for Standardization, 2. Ausgabe vom 15.10.2003

[33] Christoph Kecher: *UML 2.0 – Das umfassende Handbuch*, Galileo Computing, 2006

[34] Donald E. Knuth: *Literate Programming*, The Computer Journal, 27(2):97-111, 1984

[35] Martin Lippert, Stefan Roock, Henning Wolf: *Software entwickeln mit eXtreme Programming – Erfahrungen aus der Praxis*, dpunkt.verlag, 2002

[36] Robert C. Martin: *The Dependency Inversion Principle*, `http://www.objectmentor.com/` or `http://www.objectmentor.com/resources/articles/dip.pdf`, 1996

[37] Robert C. Martin: *Design Principles and Design Patterns*, `http://www.objectmentor.com/` 2000

[38] Robert Mecklenburg: *GNU make*, O'Reilly, 2005

[39] Scott Meyers: *Effective C++– 55 Specific Ways to Improve your Programs and Designs*, 3. Auflage, Addison Wesley, 2005

[40] Bernd Oestereich: *Analyse und Design mit UML 2.1* 8. Auflage, Oldenbourg, 2006

[41] Bernd Oestereich, Christian Weiss, Claudia Schröder, Tim Weilkiens, Alexander Lenhard: *Objektorientierte Geschäftsprozessmodellierung mit der UML*, dpunkt.verlag, 2003

[42] Andy Oram und Steve Talbott: *Managing Projects with make*, O'Reilly, 1991

[43] Pearl, Judea: *Heuristics – Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984

[44] Martin Reiser, Niklaus Wirth: *Programming in Oberon*, Addison-Wesley, 1992

[45] Roland Rüdiger: *Extensible Programming*, Technischer Bericht, Institut für Verteilte Systeme, FH Braunschweig/Wolfenbüttel, 1997, unter http://www.fh-wolfenbuettel.de/fb/i/organisation/personal/ruediger/

[46] Chris Rupp, Jürgen Hahn, Stefan Queins: *UML 2 glasklar. Praxiswissen für die UML-Modellierung und Zertifizierung*, 2. Auflage, Hanser, 2005

[47] Chris Rupp, Sophist Group: *Requirements-Engineering und -Management – Professionelle, iterative Anforderungsanalyse für die Praxis*, 3. Auflage, Hanser, 2004

[48] Gunter Saake, Kai-Uwe Sattler: *Algorithmen & Datenstrukturen – Eine Einführung mit Java*, dpunkt.verlag, 2002

[49] Wolfgang Samlowski: *Tiger im Tank – Esel am Steuer? – Projektmanagement sichert Entwicklungserfolg in kommerziellen Expertensystem-Anwendungen*, KI, 3/1991, S. 41–46

[50] A.-W. Scheer: *Wirtschaftsinformatik – Referenzmodelle für industrielle Geschäftsprozesse*, Springer, 1997

[51] G. Schmidt: *Informationsmanagement – Modelle, Methoden, Techniken*, Springer, 1999

[52] J. Schwarze: *Netzplantechnik*, Verlag Neue Wirtschafts-Briefe, 1994

[53] Sedgewick: *Algorithmen in C*, Addison Wesley, 1992

[54] Sedgewick: *Algorithmen in Java*, Pearson Studium, 2003

[55] Klaus Spanderen: *Auf kleiner Flamme – Numerische Aufgaben mit Java lösen*, iX, 12/2005, S.128–130

[56] Michael Stal: *Aus der Zukunft – Auf dem Weg zu besserer Software*, iX, 2/2006, S. 38–44

[57] Bjarne Stroustrup: *Die C++-Programmiersprache*, 4. Auflage, Addison Wesley, 2000

[58] Sun Microsystems: Java 2 Platform, Standard Edition, Version 5.0, API Specification, `http://java.sun.com/j2se/5.0/docs/api/`

[59] Christian Ullenboom: *Java ist auch eine Insel – Programmieren für die Java 2-Plattform in der Version 5*, 5. Auflage, Galileo Computing, 2005

[60] Reiner Ullrich: *Grundfinanzierte Projekte in der Luftfahrt – Leitfaden für Projektleiter*, Deutsches Zentrum für Luft- und Raumfahrt, Version 4.03, Oktober 2003

[61] Uwe Vigenschow, Björn Schneider: *Soft Skills für Softwareentwickler*, dpunkt.verlag, 2007

[62] Niklaus Wirth: *Algorithmen und Datenstrukturen*, Teubner, 1986

[63] Andreas Zeller: *Why Programs Fail – A Guide to Systematic Debugging*, dpunkt.verlag, 2005

[64] Zuser, Biffl, Grechenig, Köhle: *Software Engineering mit UML und dem Unified Process*, Pearson Studium, München, 2004

Weitere Informationen zu C und insbesondere zu C++ finden Sie in den folgenden Quellen: [26, 27, 30, 31, 32, 39, 57]

Weitere Informationen zu Java finden Sie in den folgenden Quellen: [24, 58, 59, 55]

Weiterführende Literatur zur Thematik der *Netzplanung* finden Sie in den folgenden Quellen: [1, 50, 51, 52]

Weiterführende Literatur zu den in Kap. 3 angesprochenen Tools finden Sie insbesondere zu *make* in [38, 42] sowie zu *ant* in [20, 28] und im Internet unter `ant.apache.org/manual`.

Weiterführende Literatur zu Algorithmen und Datenstrukturen finden Sie insbesondere in den folgenden Büchern: [16, 25, 43, 48, 53, 54, 44, 62]

Die Unified Modelling Language (UML) wird vertieft in den folgenden Büchern: [12, 33, 40, 41, 46, 64] und unter `www.uml.org`

Die Thematik der agilen Vorgehensmodelle wird vertieft in den folgenden Quellen: [6, 7, 22, 19, 35] und unter `www.agilealliance.org`. Weiterführende Informationen zu nicht agilen Vorgehensmodelle findet man in [9, 10, 18, 49, 56, 60, 64] und unter `www-306.ibm.com/software/awdtools/rup`.

Der Software-Entwurf wird vertieft in den folgenden Quellen: [12, 13, 14, 15, 23, 33, 36, 37, 40, 41, 45, 46]