Einführung in die Programmierung für Studierende der Naturwissenschaften

Skript zur Vorlesung von

Christian Palus

Abteilung für Angewandte Mathematik
Fakultät für Mathematik und Physik
Albert-Ludwigs-Universität Freiburg

Sommersemester 2019

Version: 22. Juli 2019

Anmerkung

Dieses Skript ist eine überarbeitete Version eines Skripts, welches im Rahmen einer Vorlesung an der Universität Freiburg im Sommersemester 2018 von Prof. Dr. Sören Bartels geschrieben wurde.

Inhaltsverzeichnis

KAPITEL 1

Erste Schritte in C++

Wir betrachten typische Aspekte eines Computerprogramms am Beispiel der Programmiersprache C++. Die Grundkonzepte gelten jedoch für die meisten Programmiersprachen.

Was ist eigentlich eine Programmiersprache? Eine Programmiersprache ist eine formale Sprache, die der Formulierung von Algorithmen dient, welche von einem Computer ausgeführt werden können. Um auch für Menschen verständlich zu sein, sind die meisten Programmiersprachen an der englischen Sprache angelehnt. Dies trifft auch auf C++ zu.

Ein C++-Programm wird in mehreren Arbeitsschritten erstellt:

(1) Der *Quelltext* des Programms wird mit einem Editor (z. B. Kate, gedit, Emacs, vim, Atom) erstellt und abgespeichert, etwa in einer Datei mit dem Namen `hello.cc`.
(2) Anschließend wird der Quelltext mit einem *Compiler* in ein ausführbares Programm *übersetzt*. Der Befehl

```
g++ -o hello hello.cc
```

erzeugt zum Beispiel durch Aufruf des C++-Compilers der Gnu Compiler Collection (GCC) im aktuellen Verzeichnis eine ausführbare Datei mit dem Namen `hello`.
(3) Das Programm kann nun durch Eingabe von `./hello` ausgeführt werden.

Hinweis: Mit der Tastenkombination `<Strg>+<C>` kann ein laufendes Programm in der Konsole abgebrochen werden.

Das Vokabular einer Programmiersprache besteht aus ihren *Schlüsselwörtern*. Das sind diejenigen Wörter, die durch die Definition der Sprache eine festgelegte Bedeutung haben und vom Compiler erkannt werden.

1.1. Das erste Programm

Um ein intuitives Verständnis vom Aufbau eines C++-Programms zu entwickeln, steigen wir mithilfe einiger Beispiele direkt in die Programmierung mit C++ ein. Die vorkommenden Anweisungen werden in späteren Lektionen genauer erläutert. Die Datei `hello.cc` enthalte den folgenden Quelltext (wobei die Zeilennummern nur der Orientierung dienen und nicht zum eigentlichen Quellcode gehören):

```
1 #include<iostream>
2 main()
3 {
4   std::cout << "Hello World!\n";
5 }
```

Wird dieses Programm kompiliert und ausgeführt, so wird der Quelltext Zeile für Zeile abgearbeitet. Mit der Präprozessoranweisung **#include<...>** in Zeile 1 wird eine zusätzliche Datei in das Programm eingebunden. In diesem Fall die Bibliothek <iostream> (input/output stream), welche Methoden zur Texteingabe und -ausgabe bereitstellt. Das eigentliche Hauptprogramm beginnt in Zeile 2 mit dem Schlüsselwort *main()* und wird von den geschweiften Klammern in den Zeilen 3 und 5 umfasst. Jedes C++-Programm hat eine solche `main`-Funktion. In Zeile 4 steht die einzige Anweisung dieses kurzen Programms: `std::cout` steht hier für die Standardausgabe auf der Konsole und diese Zeile bewirkt, dass alles, was nach dem *Ausgabeoperator* << steht, auf dem Bildschirm ausgegeben wird. Unser Programm schreibt die in Anführungszeichen stehende Zeichenkette, das heißt den Text *Hello World!* gefolgt von einem Zeilenumbruch (`"\n"`), in die Konsole.

Wichtig: Jede Anweisung in einem C++-Programm muss mit einem Semikolon abgeschlossen werden.

1.2. Ein komplexeres Beispiel

Als nächstes betrachten wir ein Programm, welches eine vom Benutzer eingegebene Zahl einliest, deren Quadratwurzel berechnet und diese ausgibt. Dazu sei der folgende Quelltext in einer Datei abgespeichert:

```
1  #include<iostream>
2  #include<cmath>
3
4  // Programm zur Berechnung von Quadratwurzeln
5  int main()
6  {
7    float x,y;
8
9    std::cout << "Berechnung der Quadratwurzel\n";
10   std::cout << "Gib eine Zahl ein: ";
11
12   std::cin >> x;
13
14   if ( x >= 0 ) {
15     y = sqrt(x);
16     std::cout << "Die Wurzel aus " << x << " ist " << y << std::endl;
17   }
18   else {
19     std::cout << "FEHLER: Eingegebene Zahl ist negativ!\n";
20   }
21
22   return 0;
23 }
```

In den Zeilen 1 und 2 werden wieder Hilfsbibliotheken eingebunden, <cmath> stellt dabei verschiedene mathematische Funktionen bereit. Die dritte Zeile ist leer und wird daher vom

Compiler ignoriert. Bei Zeile 4 handelt es sich um einen *Kommentar*, welcher der Erklärung des Quellcodes dient und vom Compiler ebenfalls ignoriert wird. Alles was in einer Zeile hinter einem Doppel-Schrägstrich (*//*) steht, ist für den Compiler unsichtbar. Das Hauptprogramm beginnt in Zeile 5. Das Schlüsselwort **`int`** kennzeichnet hier, dass die `main`-Funktion einen ganzzahligen (von engl. *integer*) Rückgabewert hat. Dies hat hier keine weitere Bedeutung, ist jedoch eine Konvention, die von manchen Compilern erwartet wird. Genauso verhält es sich mit der Rückgabe des Werts 0 in Zeile 22. In Zeile 7 werden zwei Variablen `x` und `y` deklariert. Damit teilen wir dem Compiler mit, dass wir im Speicher des Computers Platz für die Darstellung von zwei Gleitkommazahlen benötigen. Was das genau bedeutet, wird im nächsten Abschnitt klar werden. Die Textausgaben in den Zeilen 9 und 10 kennen wir bereits aus unserem ersten Beispiel. Neu ist hingegen das Einlesen eines Werts in Zeile 12 als Gegenstück zur Ausgabe: `std::cin` bezeichnet die Standardeingabe über die Konsole und der *Eingabeoperator* >> bewirkt, dass ein vom Benutzer eingegebener Wert im Speicherplatz der Variable `x` gespeichert wird. Eine solche Eingabe wird immer mit `<Enter>` abgeschlossen.

Der Rest ist einigermaßen selbsterklärend und die Syntax werden wir in den folgenden Lektionen genauer erläutern. In Zeile 14 wird mit dem Schlüsselwort **`if`** geprüft, ob die eingegebene Zahl größer oder gleich 0 ist. Falls das der Fall ist, wird der von geschweiften Klammern umschlossene darauffolgende Block ausgeführt. Falls nicht, so wird der Block hinter dem Schlüsselwort **`else`** ausgeführt. In Zeile 15 wird die Variable `y` mit dem Wert der Quadratwurzel aus `x` belegt. In Zeile 16 wird das Ergebnis der Rechnung auf dem Bildschirm ausgegeben. Dabei wird der Operator << mehrfach verwendet, um die gewünschte Ausgabe aus mehreren Einzelbausteinen zusammenzusetzen. Die Ausgabe von `std::endl` ist dabei gleichbedeutend mit `"\n"` und bewirkt einen Zeilenumbruch (end-of-line).

Zum Abschluss noch ein paar Hinweise: Programme in C++ können theoretisch komplett formatfrei geschrieben werden, das heißt es spielt für den Compiler keine Rolle, ob der Quelltext eine bestimmte Zeilenstruktur hat. Das Programm aus dem obigen Beispiel könnte man also auch in eine einzige lange Zeile schreiben, solange man die entsprechenden Klammern und Semikolons richtig setzt. Für Programmierer die so etwas tun, ist jedoch ein besonders unangenehmer Platz in der Hölle reserviert. Daher – und natürlich um generell die Lesbarkeit von Programmen zu erhöhen – sollte man sich gleich zu Beginn einen gut lesbaren Programmierstil angewöhnen und gewisse Konventionen beachten. Dazu gehört zum Beispiel, dass man sinnvolle Namen für Variablen und Funktionen wählt, geschweifte Klammern nach einheitlichen Regeln setzt und zusammengehörende Teile des Quelltexts um die gleiche Anzahl von Leerzeichen einrückt.

Gerade bei komplizierteren Programmen sind Kommentare eine gute Möglichkeit, um die Funktionsweise eines Programms zu erklären. Ergänzend zu dem einzeiligen Kommentar aus dem Beispielprogramm sind in C++ auch mehrzeilige Kommentare möglich. Sie werden durch */** eingeleitet und mit **/* beendet:

```
/* Ich bin ein
mehrzeiliger Kommentar */
```

Die Verwendung von Kommentaren ist auch bei der Fehlersuche in Programmen (dem *Debuggen*) hilfreich. Da sie vom Compiler ignoriert werden, können selektiv ganze Programmteile *auskommentiert* werden, um den Fehlerbereich so einzuschränken.

Kommentare zur Erläuterung der Programmfunktion sollten mit Bedacht eingesetzt werden. Sinnvolle Bezeichner und strukturierter Code sind häufig hilfreicher als eine Vielzahl langer Kommentare. Kurze und präzise Hinweise schaden jedoch selten. Eine gute Faustregel ist es, Quelltexte so zu schreiben und kommentieren, als ob es sich bei der Person, die diese lesen muss, um einen gewaltbereiten Psychopathen handelt, der den eigenen Wohnort kennt.

1.3. Aufbau eines C++-Programms

Unsere einleitenden Beispiele lassen den grundlegenden Aufbau eines C++-Programms bereits erahnen. In den ersten Zeilen stehen in der Regel *Präprozessoranweisungen*, mit denen notwendige Hilfsbibliotheken eingebunden werden. Eine Präprozessoranweisung ist dabei eine Zeile, die mit einer Raute (**#**) beginnt und heißt so, weil sie vom Präprozessor noch vor der eigentlichen Übersetzung des Programms bearbeitet wird. Solche Anweisungen müssen deshalb auch nicht wie alle übrigen Anweisungen mit einem Semikolon abgeschlossen werden. Der Präprozessor kann auch komplexere Aufgaben als das Einbinden von Hilfsbibliotheken ausführen, wir beschränken uns jedoch auf den bereits bekannten **`#include`**-Befehl.

Der Hauptteil eines jeden C++-Programms steht innerhalb der `main`-Funktion, das heißt innerhalb der geschweiften Klammen, die auf das Schlüsselwort `main()` folgen. Ganz allgemein bezeichnet man eine Folge von Anweisungen die von geschweiften Klammen umgeben sind als *Block*. In den folgenden Abschnitten werden wir einige Beispiele von möglichen Anweisungen kennenlernen. Um dieses Skript übersichtlicher zu gestalten, werden wir uns bei der Darstellung häufig auf die Anweisungen selbst beschränken und gehen vorerst immer davon aus, dass diese innerhalb der `main`-Funktion eines Programms stehen. Wie bereits erwähnt, ist es eine gängige Konvention, dass die `main`-Funktion den Rückgabetyp **`int`** hat und bei normaler Ausführung den Wert 0 zurückgibt. Die genaue Bedeutung von Rückgabetypen und -werten werden wir in einem späteren Kapitel erläutern. Zusammengefasst hat ein C++-Programm die folgende Struktur:

```
1 #include <...>
2 int main()
3 {
4   // Anweisungen
5   return 0;
6 }
```

1.4. Variablen

Eine Variable ist eine Stelle im Arbeitsspeicher des Computers, an der Werte abgelegt und später wieder abgerufen werden können. Der Zugriff auf den Inhalt dieses Speicherbereichs erfolgt über den entsprechenden Bezeichner der Variable.

Typen. In C++ gibt es unterschiedliche Typen von Variablen, die unterschiedlich viel Speicherplatz benötigen. Die Größe des jeweils reservierten Speicherbereichs ist maschinenabhängig. Auf neueren Systemen werden für eine Variable vom Typ **`int`** in der Regel 32 Bit und für eine Variable vom Typ **`long`** 64 Bit Speicherplatz bereitgestellt und die in der folgenden Tabelle angegebenen Werte erhöhen sich dementsprechend.

Die wichtigsten Typen in C++		
Typ	Werte	Speicher
`void`	–	0 Bit
`bool`	$\{0, 1\}$	1 Bit
`int`	mind. $\{-32768, \ldots, 32767\}$	mind. 16 Bit
`long`	mind. $\{-2147483648, \ldots, 2147483647\}$	mind. 32 Bit
`float`	$[\pm 3.402 \cdot 10^{38}]$	32 Bit
`double`	$[\pm 1.797 \cdot 10^{308}]$	64 Bit
`char`	`'a','b','c','0','1',...`	systemabh.

Tabelle 1.1. Variablentypen in C++.

Die wichtigsten Typen sind in Tabelle 1.1 aufgelistet. Der Typ `bool` dient dem Speichern von Wahrheitswerten logischer oder *boolescher* Ausdrücke. Alternativ zu den Werten `0` und `1` können dabei auch die Schlüsselwörter `false` und `true` verwendet werden. Variablen vom Typ `int` oder `long` können nur ganzzahlige Wert speichern, solche vom Typ `float` oder `double` sind zum Speichern von Gleitkommazahlen vorgesehen. Der Typ `char` dient dem Speichern von einzelnen Zeichen und ist für Rechnungen nicht geeignet.

Bevor eine Variable in einem Programm benutzt werden kann, muss man sie *deklarieren*, damit der entsprechende Speicherplatz reserviert wird. Dies geschieht durch Angabe des Typs und eines Bezeichners für die Variable, wobei auch mehrere Variablen gleichen Typs gleichzeitig deklariert werden können.

```
int i;                  // Deklaration einer Integervariable
float x;                // Deklaration einer Floatvariable
int summe, differenz;   // Deklaration von zwei Variablen
```

Hierbei muss man beachten, dass eine Variable immer nur innerhalb des Blocks, in dem sie deklariert wurde, verwendet werden kann. Man spricht vom sogenannten *Scope* oder Gültigkeitsbereich.

Hinweis: Variablennamen dürfen in C++ aus Buchstaben, Ziffern und Unterstrichen bestehen, wobei das erste Zeichen keine Ziffer sein darf. Schlüsselwörter (z. B. `int`, `main`, `if`) dürfen logischerweise nicht als Bezeichner für Variablen verwendet werden. Es wird zwischen Groß- und Kleinschreibung unterschieden, das heißt, dass beispielsweise `Hausnr` und `HausNr` zwei unterschiedliche Variablen bezeichnen können.

Zuweisungen. Die Zuweisung von Werten erfolgt mit dem Zuweisungsoperator = und bewirkt, dass die links des Operators stehende Variable mit dem Wert des rechts davon stehenden *Ausdrucks* belegt wird. Was wir unter einem Ausdruck genau verstehen, wird in Kürze klar werden. Für den Moment begnügen wir uns noch mit konstanten Werten. Wenn man möchte, kann man Variablen direkt bei der Deklaration initialisieren, das heißt ihnen einen bestimmten Wert zuweisen.

```
int j = 3;              // Deklaration und Initialisierung
char buchstabe = 'b';   // Deklaration und Initialisierung
j = 4;                  // Aenderung des Werts auf 4
```

Achtung: Es hängt vom jeweiligen Compiler ab, was passiert, wenn man mit einer deklarierten Variable arbeitet, der nie ein Wert zugewiesen wurde. Dies muss nicht unbedingt zu einer Warnung oder Fehlermeldung des Compilers führen.

Konstanten. Konstanten sind spezielle Variablen deren Werte sich im Verlauf des Programms nicht ändern dürfen. Sie werden mit dem Schlüsselwort `const` definiert und müssen direkt initialisiert werden.

```
const float pi = 3.141592;      // Die Kreiszahl Pi
```

Versucht man den Wert einer Konstanten später zu ändern, erhält man beim Übersetzen des Programms eine Fehlermeldung.

Klassischer Fehler: Man muss darauf achten, passende Variablentypen zu wählen. Wenn man einer Variable den Wert eines Ausdrucks zuweist, dessen Typ nicht zu dem der Variable passt, so findet – falls möglich – eine implizite Typkonvertierung statt.

```
int i;    // in i koennen nur ganze Zahlenwerte gespeichert werden
i = 2.7;  // hier findet eine Typkonvertierung statt
```

Im obigen Fall entählt die Variable `i` beispielsweise anschließend den Wert 2.

1.5. Felder

Es ist oft hilfreich, mehrere Variablen gleichen Typs zusammenzufassen, etwa wenn man mit Vektoren oder Matrizen arbeitet. Die einfachste Möglichkeit dazu ist die Verwendung von *Feldern* (engl. *Arrays*). Ein Feld wird deklariert wie eine Variable, wobei die Größe – das heißt die Anzahl der Einträge – in eckigen Klammern angehängt wird, zum Beispiel:

```
float vektor[3];        // Feld aus 3 Gleitkommazahlen
int matrix[3][2];   // 3x2-Matrix aus ganzen Zahlen
```

Der Zugriff und die Zuweisung von Werten funktionieren wie bei normalen Variablen, der Index des jeweiligen Eintrags steht in eckigen Klammern.

```
vektor[0] = 2.1;
vektor[1] = 0.0;
vektor[2] = 5.7;
matrix[2][1] = 2;
```

Dabei muss man beachten, dass Indizes in C++ immer mit 0 beginnen. Im Beispiel besitzt der Vektor beispielsweise Einträge mit den Indizes 0, 1 und 2. Wie Variablen können auch Felder direkt bei der Deklaration initialisiert werden. Dazu werden die entsprechenden Werte, durch Kommas getrennt, in geschweiften Klammern als sogenannte *Initialisierungsliste* angegeben. Die Angabe der Feldgröße kann in diesem Fall weggelassen werden, wobei bei mehrdimensionalen Feldern lediglich die Größe in der ersten Dimension fehlen darf:

```
int vektor[] = {2.1, 0.0, 5.7};
float matrix[][2] = {{1,0}, {4,2}, {0,2}};
```

Klassischer Fehler: Versucht man im obigen Beispiel auf den nicht definierten Eintrag `vektor[3]` zuzugreifen, führt das in der Regel nicht zu einer Fehlermeldung des Compilers, sondern zu einem unerwünschten Programmverhalten. Diese Art von Programmierfehlern ist besonders schwer zu finden.

1.6. Ein- und Ausgabe über `std::cin` und `std::cout`

Die Ein- und Ausgabe von Werten haben wir bereits in den einleitenden Beispielen kennengelernt. Hierfür muss die Bibliothek `<iostream>` vor der `main`-Funktion mit einer Präprozessoranweisung eingebunden werden: `#include <iostream>`.

Möchte man einer Variable einen vom Benutzer in der Konsole einzugebenden Wert zuweisen, so geschieht dies über den *Input Stream* `std::cin` mithilfe des *Eingabeoperators* >>, der manchmal auch als Extraktionsoperator bezeichnet wird.

```
int j;
std::cin >> j;
```

Die Benutzereingabe in der Konsole wird dabei wie alle dortigen Eingaben mit `<Enter>` bestätigt. Die verwendete Variable muss vorher deklariert werden und zum Typ der Eingabe passen, da sonst eine Typkonvertierung stattfindet.

Die Ausgabe von Text und Variablenwerten in der Konsole geschieht über den *Output Stream* mit dem *Ausgabeoperator* <<. Vorher festgelegte Texte werden dabei als Zeichenkette, das heißt umschlossen von doppelten Anführungszeichen ("), hinter den Operator geschrieben. Die Werte von Variablen werden ausgegeben, indem man den Bezeichner der Variable (ohne Anführungszeichen) hinter den Ausgabeoperator schreibt. Wie wir bereits gesehen haben, können dabei mehrere Ausgaben aneinander gehängt werden. Das Zeichen `"\n"` innerhalb einer Zeichenkette bewirkt genauso wie das Schlüsselwort `std::endl` einen Zeilenumbruch.

```
1 #include <iostream>
2 int main()
3 {
4   int beine = 8;
5   float pi = 3.14, e = 2.71;
6   std::cout << "Spinnen haben " << beine << " Beine\n";
7   std::cout << "pi=" << pi << ", e=" << e << std::endl;
8   return 0;
9 }
```

Dieser Code führt beispielsweise zu der Ausgabe von

```
Spinnen haben 8 Beine
pi=3.14, e=2.71
```

in der Konsole.

1.7. Operatoren und Ausdrücke

Ein *Ausdruck* ist eine zulässige Formel aus Konstanten, Variablen, Operatoren und/oder Funktionsaufrufen, deren Auswertung bei gültiger Belegung der Variablen einen Wert eines bestimmten Typs liefert. Abstrakt können Ausdrücke induktiv definiert werden:

- Konstanten und Variablen sind Ausdrücke.
- Zulässige Verknüpfungen eines oder mehrerer Ausdrücke mit Operatoren und/oder Funktionen liefern wieder einen Ausdruck.

Bei den Funktionen kann es sich dabei um eigene Funktionen handeln (betrachten wir später), oder um bereits vorhandene, wie etwa die mathematischen Funktionen aus der Bibliothek `<cmath>`, wozu unter anderem

`cos`, `sin`, `tan`, `exp`, `log`, `sqrt` und `pow`

gehören.

Operatoren. Wir betrachten nun einige Arten von häufig verwendeten Operatoren in C++. In späteren Lektionen werden noch mehr hinzukommen.

Arithmetische Operatoren: Die Zeichen

`+`, `-`, `*` und `/`

stehen in C++ für die klassischen Grundrechenarten und können beispielsweise auf die Datentypen **`int`** und **`float`** angewendet werden. Wenn man nicht genau weiß, was man tut, sollte man immer nur Variablen gleichen Typs mit einem Operator verknüpfen.

Achtung: Wendet man den Divisionsoperator `/` auf ganzzahlige Typen an, so wird auch ein ganzzahliger Typ zurückgegeben und daher der Nachkommaanteil abgeschnitten. Zum Beispiel liefert `7/4` den Wert `1` wohingegen `7.0/4.0` den Wert `1.75` liefert.

Der *Modulo-Operator* `%` kann auf zwei ganzzahlige Typen angewendet werden, wobei der Ausdruck `k%l` den Rest liefert, der bei der Division von `k` durch `l` entsteht.

Kommen mehrere Operatoren in einem Ausdruck vor, so lässt sich die Reihenfolge ihrer Auswertung wie bei einer mathematischen Formel durch runde Klammern, also `(` und `)`, beeinflussen. Ohne die Verwendung von Klammern gilt die folgende Bindungsstärke:

(1) `+`, `-` (unär/als Vorzeichen) – bindet am stärksten,
(2) `*`, `/`, `%`,
(3) `+`, `-` (binär/als Rechenoperation) – bindet am schwächsten.

Inkrement-/Dekrementoperatoren: Die Operatoren `++` und `--` erhöhen bzw. verringern den Wert des Operanden jeweils um 1. Beispielsweise hat

```
x++;
```

genau den gleichen Effekt wie

```
x = x+1;
```

oder auch

```
x += 1;
```

Der Operator += kann dabei auch mit einem anderen Inkrement als 1 verwendet werden, x+=3 erhöht etwa den Wert von x um 3. Analog zu += lassen sich die Operatorn −=, *= und /= verwenden.

Hinweis: Die Operatoren ++ und −− können sowohl vor (prefix) als auch hinter (postfix) dem Operanden stehen. Bei Voranstellung wird der entsprechende Wert vor der Auswertung des Ausdrucks erhöht bzw. verringert, im anderen Fall danach. Dazu ein Beispiel:

```
int n = 2;
int x, y;
x = n++; // x erhaelt den Wert 2 und n den Wert 3
y = ++n; // y erhaelt den Wert 4 und n den Wert 4
```

Vergleichsoperatoren: Die Operatoren

==, !=, <, >, <= und >=

entsprechen genau den mathematischen Relationen »gleich«, »ungleich«, »kleiner«, »größer«, »kleiner-gleich« sowie »größer-gleich«, und geben dementsprechend einen logischen Wert, also **true** oder **false** zurück. Die Vergleichsoperatoren binden schwächer als die arithmetischen Operatoren.

Logische Operatoren: Die logischen Verknüpfungen UND und ODER sowie die Negation sind durch die Operatoren

&&, || und !

gegeben. Hierbei gilt die Regel, dass ! stärker bindet als &&, was wiederum stärker bindet als ||. Falls etwa die Variablen a, b und c jeweils den Wert 2 haben und die Variable d den Wert 0, so hat der Ausdruck

```
(a==b || b==c) && c==d
```

den Wert **false**, der Ausdruck

```
a==b || b==c && c==d
```

hingegen den Wert true.

Bindungsstärke aller bisherigen Operatoren. . Wir listen noch einmal alle bisherigen Operatoren auf, geordnet nach ihrer Bindungsstärke.

(1) () (Klammerung),
(2) ++/−− (postfix), [] (Zugriff auf Array-Element),
(3) +/− (Vorzeichen), !, ++/−− (prefix),
(4) *, /, %,
(5) +, −,
(6) <, >, <=, >=,
(7) ==, !=,

(8) `&&`,
(9) `||`,
(10) `=`, `+=`, `-=`, `*=`, `/=`.

Ohne Klammern werden Operatoren gleicher Bindungsstärke von links nach rechts ausgewertet. Um Fehler zu vermeiden, sollte man im Zweifel lieber eine Klammer zu viel als zu wenig setzen.

1.8. Kontrollstrukturen

Kontrollstrukturen ermöglichen uns die Steuerung des Programmablaufs durch Fallunterscheidungen und Wiederholungen.

Die `if-else`-Verzweigung. Die wichtigste Möglichkeit zur Fallunterscheidung ist die **`if-else`**-*Verzweigung*, die wir bereits aus unserem einleitenden Beispiel kennen. Sie erlaubt uns in Abhängigkeit einer Bedingung unterschiedliche Anweisungen auszuführen. Die Syntax ist wie folgt:

```
if (bedingung) {
  anweisungen1;
}
else {
  anweisungen2;
}
```

Hat der Ausdruck `bedingung` den logischen Wert **`true`**, so wird der darauffolgende Anweisungsblock ausgeführt, hat er den logischen Wert **`false`**, so wird der Anweisungsblock nach dem Schlüsselwort **`else`** ausgeführt. Das Schlüsselwort **`else`** mit dem dazugehörenden Anweisungsblock ist dabei optional. Falls einer der Blöcke aus lediglich einer einzigen Anweisung besteht, so können die geschweiften Klammern weggelassen werden. Das folgende Beispiel zeigt, dass mehrere solcher Verzweigungen ineinander geschachtelt werden können.

```
 1 #include<iostream>
 2 int main()
 3 {
 4   int zahl;
 5   std::cin >> zahl ;
 6   if ( zahl > 0 ) {
 7     std::cout << "Die Zahl ist positiv\n";
 8   }
 9   else if ( zahl < 0 ) {
10     std::cout << "Die Zahl ist negativ\n";
11   }
12   else {
13     std::cout << "Die Zahl ist Null\n";
14   }
15   return 0;
16 }
```

Als Bedingung in der `if`-Abfrage kann theoretisch jeder Ausdruck stehen. Ein Ausdruck gilt dabei als `false`, falls sein numerischer Wert gleich 0 ist, anderenfalls gilt er als `true`. In der Regel verwendet man jedoch Vergleiche bzw. logische Verknüpfungen davon als Bedingung.

Klassischer Fehler: Ein häufiger Anfängerfehler ist die Verwendung des Zuweisungsoperators = anstatt des Vergleichsoperators == in der Bedingung einer `if`-Abfrage. Das Konstrukt `if(a=b)` sorgt dafür, dass der bedingte Block immer dann ausgeführt wird, wenn `b` nicht 0 ist. Außerdem wird `a` dabei der Wert von `b` zugewiesen, das heißt nach der Abfrage hat die eigentlich richtige Bedingung `a==b` in jedem Fall den Wert `true`.

Die `while`-Schleife. Die Verwendung von `while`-*Schleifen* ist die einfachste Möglichkeit, um Wiederholungen zu realisieren. Diese Schleifen sind nach dem Schlüsselwort `while` benannt, durch welches sie eingeleitet werden. Die Syntax ist folgendermaßen:

```
while (bedingung) {
  anweisungen;
}
```

Verwendet man eine solche Schleife in einem Programm, so wird der Anweisungsblock der Schleife (der sogenannte *Schleifenkörper*) wiederholt, solange die Bedingung `bedingung` erfüllt ist, das heißt den Wahrheitswert `true` hat.

Beispiel:

```
1  #include<iostream>
2
3  int main()
4  {
5    int countdown=3;
6    while (countdown>0) {
7      std::cout << countdown << ", ";
8      countdown--;
9    }
10   std::cout << "Los!\n";
11   return 0;
12 }
```

In diesem Programm wird in jedem Schleifendurchlauf der Wert der Variable `countdown`, gefolgt von einem Komma und einem Leerzeichen, ausgegeben und anschließend um 1 verringert. Führt man das übersetzte Programm aus, so erhält man die Ausgabe

```
3, 2, 1, Los!
```

in der Konsole.

Da die Bedingung einer `while`-Schleife ganz oben steht und direkt vor dem ersten Durchlauf geprüft wird, spricht man auch von einer *kopfgesteuerten* Schleife. Eine leichte Abwandlung ist die *fußgesteuerte* `do-while`-*Schleife*. Ihre Syntax lautet:

```
do{
  anweisungen;
} while (bedingung);
```

Man beachte dabei das Semikolon in der letzten Zeile. Bei der Verwendung einer **do-while**-Schleife wird die Bedingung erst nach einem Durchlauf des Schleifenkörpers ausgewertet. Das bedeutet, dass die Anweisungen im Schleifenkörper in jedem Fall mindestens einmal ausgeführt werden – im Gegensatz zur normalen **while**-Schleife, deren Schleifenkörper übersprungen wird, falls die Bedingung gleich zu Beginn nicht erfüllt ist.

Klassischer Fehler: Selbstverständlich muss man darauf achten, dass die Abbruchbedingung irgendwann erfüllt wird. Sonst landet man in einer *Endlosschleife* und die Ausführung des Programms wird unendlich lange fortgesetzt – oder zumindest bis zum nächsten Stromausfall oder Abbruch durch den Benutzer.

Die `for`-Schleife. Wenn man a priori weiß, wie viele Wiederholungen eines Anweisungsblocks in einem Programm nötig sind, dann bietet sich die Verwendung einer **for**-*Schleife* an. Dies sind die in der Praxis wohl am häufigsten verwendeten Schleifen. Abstrakt folgt Ihre Verwendung dem Syntax:

```
for (initialisierung; bedingung; inkrement) {
  anweisungen;
}
```

Dabei ist `initialisierung` eine Anweisung, `bedingung` ein boolescher Ausdruck und `inkrement` wieder eine Anweisung. Tritt in einem Programm eine **for**-Schleife auf, so passiert Folgendes:

(1) Die Anweisung `initialisierung` wird ausgeführt.
(2) Der Ausdruck `bedingung` wird ausgewertet; Hat er den Wert **true**, so läuft die Schleife weiter, anderenfalls wird der Schleifenkörper übersprungen und es wird direkt mit (5) fortgefahren.
(3) Die Anweisungen im Schleifenkörper werden ausgeführt.
(4) Die Anweisung `inkrement` wird ausgeführt und die Schleife springt zurück zu (2).
(5) Die Schleife endet und die Programmausführung wird mit der nächsten Anweisung nach der Schleife fortgesetzt.

Zur Veranschaulichung betrachten wir noch einmal unser Beispielprogramm mit der **while**-Schleife. Dieses lässt wie folgt auch äquivalent mit einer **for**-Schleife realisieren:

```
1 #include<iostream>
2 int main()
3 {
4   for (int countdown=3; countdown>0; countdown--) {
5     std::cout << countdown << ", ";
6   }
7   std::cout << "Los!\n";
8   return 0;
9 }
```

Da in jeder **for**-Schleife ein initialisierter Wert bei jedem Schleifendurchlauf nach einer gegebenen Rechenvorschrift verändert wird, und damit sozusagen die Anzahl der Durchläufe gezählt wird, spricht man auch von einer *Zählschleife*.

Hinweis: Die Anweisung `inkrement` kann eine beliebige Anweisung sein. Man könnte beispielsweise durch `countdown = countdown - 3` beziehungsweise `countdown-=3` den Wert von `countdown` in jedem Schleifendurchlauf um 3 verringern. Wie bei den **while**-Schleifen muss man jedoch darauf achten, nicht in einer Endlosschleife zu landen.

Labels und die goto-Anweisung. Mithilfe der Anweisung **goto** kann man jederzeit zu einer Stelle des Programms springen, welche mit einem sogenannten *Label* markiert ist:

```
goto markierung;
...
// hier geht es dann weiter
markierung:
...
```

Ein Label kann an jeder beliebigen Stelle des Programms stehen und wird mit einem Doppelpunkt abgeschlossen. Um mit **goto** zu einem Label zu springen, müssen die **goto**-Anweisung und das Label innerhalb derselben Funktion (etwa der `main`-Funktion) stehen. Dabei ist es egal, ob das Label vor oder hinter der **goto**-Anweisung steht.

Achtung: Die Verwendung von Labels und **goto**-Anweisungen führt zu sehr unübersichtlichen »Spaghetti-Programmen« und sollte vermieden werden. Wir haben sie hier nur der Vollständigkeit halber erwähnt. Formal sind **goto**-Anweisungen niemals notwendig und können immer durch andere Kontrollstrukturen ersetzt werden.

Die switch-Verzweigung. Die Fallunterscheidung mit Hilfe der Schlüsselwörter **if** und **else** haben wir bereits kennengelernt. Möchte man für eine Programmverzweigung mehr als zwei Fälle unterscheiden, so bietet sich manchmal die Verwendung einer **switch**-Verzweigung anstatt geschachtelter **if-else**-Verzweigungen an. Die **switch**-Verzweigung ermöglicht eine Steuerung des Programmflusses in Abhängigkeit eines Ausdrucks, welcher verschiedene konkrete Werte annehmen kann. Der Einfachheit halber beschränken wir uns hier auf ganzzahlige Ausdrücke. Die Syntax der **switch**-Verzweigung besteht aus den Schlüsselwörtern **switch**, **case**, **default** und **break**, und wird am besten anhand eines Beispiels erklärt:

```
 1 #include <iostream>
 2 int main()
 3 {
 4   int zahl;
 5   std::cout << "Gib eine ganze Zahl ein: ";
 6   std::cin >> zahl;
 7   switch(zahl) {
 8     case 0:
 9       std::cout << "Die Zahl ist 0!\n";
10       break;
11     case 2:
```

```
12        std::cout << "Die Zahl ist 2!\n";
13        break;
14      case 4:
15        std::cout << "Die Zahl ist 4!\n";
16        break;
17      default:
18        std::cout << "Die Zahl ist weder 0 noch 2 noch 4!\n";
19        break;
20    }
21    return 0;
22  }
```

Je nachdem, ob `zahl` hier den Wert 0, 2 oder 4 hat, werden die Anweisungen nach dem jeweiligen **`case`**-Label ausgeführt und damit der entsprechende Text ausgegeben. Die Anweisungen nach dem optionalen **`default`**-Label werden ausgeführt, falls der tatsächliche Wert von `zahl` zu keinem der **`case`**-Labels passt. Lässt man das **`default`**-Label weg, so wird die **`switch`**-Verzweigung in diesem Fall ohne Wirkung verlassen. Dabei sind einige Dinge zu beachten:

- Mithilfe der **`case`**-Labels können beliebig viele Fälle unterschieden werden, jedoch umfasst jedes **`case`**-Label nur einen möglichen Wert des Bedingungsaudrucks der Verzweigung. (Im Beispiel ist dieser Ausdruck durch die Variable `zahl` gegeben.)
- Nach jedem **`case`**-Label können beliebig viele Anweisungen stehen. Am Ende einer solchen Folge von Anweisungen sollte in den meisten Fällen ein **`break`** stehen. Der Grund dafür ist der folgende Punkt.
- Die **`case`**-Labels funktionieren wie die Labels im Zusammenhang mit einer **`goto`**-Anweisung: Das Programm springt an die Stelle des entsprechenden **`case`**-Labels und wird danach linear fortgeführt. Inbesondere werden alle Anweisungen unter den darauffolgenden **`case`**-Labels ebenfalls ausgeführt, falls man die **`break`**-Anweisung weglässt.

Die Anweisungen `break` und `continue`. Das Schlüsselwort **`break`** kennen wir bereits aus der **`switch`**-Verzweigung, wo es bewirkt, dass der Körper der Verzweigung verlassen und die Programmausführung mit den Anweisungen nach der Verzweigung fortgesetzt wird. Analog dazu kann man die Anweisung

```
break;
```

innerhalb einer **`for`**-, **`while`**-, oder **`do-while`**-Schleife verwenden, um diese an einer beliebigen Stelle abzubrechen, also mit der nächsten Anweisung nach der Schleife fortzufahren. Bei mehreren geschachtelten Schleifen bzw. **`switch`**-Anweisungen wird nur diejenige verlassen, welche die **`break`**-Anweisung unmittelbar umgibt, beispielsweise:

```
1  #include <iostream>
2
3  int main() {
4    for (int i=0; i<=9; i++) {
5      for (int j=0; j<=9; j++) {
```

```
 6        if (i==5 && j==6) {
 7          break;
 8        }
 9        std::cout << i << j << " ";
10      }
11      std::cout << std::endl;
12    }
13    return 0;
14  }
```

Dieses Programm, über dessen Sinnhaftigkeit man streiten kann, führt zur Ausgabe der Zahlen von 00 bis 99, wobei die Zahlen von 56 bis 59 ausgelassen werden.

Innerhalb von Schleifen hat man zusätzlich die Möglichkeit mittels der Anweisung

```
continue;
```

nur den aktuellen Schleifendurchlauf abzubrechen, das heißt an das Ende des Schleifenkörpers zu springen und mit dem nächsten Durchlauf zu beginnen.

1.9. Strukturierte Programmierung mit Funktionen

Ein wirklich nützliches C++-Programm hat in der Regel einen erheblich größeren Umfang als unsere bisherigen Beispiele. Um große Programme übersichtlicher gestalten zu können, bietet C++ die Möglichkeit, diese in kleine, selbständige Unterprogramme, welche als *Funktionen* bezeichnet werden, aufzuteilen. Einige Funktionen, etwa die mathematischen aus der Bibliothek `<cmath>`, haben wir bereits kennen gelernt. Auch die das Hauptprogramm selbst ist eine Funktion mit dem Namen `main`.

Ganz allgemein führt eine Funktion bestimmte Anweisungen aus und gibt am Ende einen Wert eines vorher festgelegten Typs zurück, den sogenannten *Rückgabewert*. Dabei können die Anweisungen von einem oder mehreren *Übergabeparametern* abhängen. Funktionen schreibt man in der Regel zur Erfüllung einer speziellen Aufgabe. Neben der Übersichtlichkeit erhöht ein solches Vorgehen auch die Wiederverwendbarkeit des Codes, da eine einmal geschriebene Funktion in einer Datei abgespeichert und dann von verschiedenen Programmen immer wieder verwendet werden kann.

Ähnlich wie Variablen müssen alle Funktionen vor ihrem ersten Aufruf deklariert werden, damit sie dem Compiler bekannt sind. Neben der Deklaration, die in jedem Fall vor der ersten Verwendung erfolgen muss, gehört zu einer Funktion auch ihre *Definition*. Deklaration und Definition können dabei zusammen oder getrennt erfolgen, analog der Deklaration und Initialisierung von Variablen.

Die Definition und Deklaration einer Funktion geschieht abstrakt nach der folgenden Syntax:

```
rtyp fname(typ1 par1, typ2 par2, ... , typN parN)
{
  anweisungen;
  return ret;
}
```

Dabei gelten für den Funktionsnamen `fname` die gleichen Einschränkungen wie für Variablennamen. Der Rückgabetyp `rtyp` kann genauso wie die Parametertypen `typ1,...,typN` beliebig gewählt werden. Der Rückgabewert wird durch das Schlüsselwort **`return`** gekennzeichnet. Sobald das Programm auf eine **`return`**-Anweisung trifft, wird der entsprechende Wert zurückgegeben und die Funktion verlassen (das heißt das Programm springt wieder an die Stelle des Funktionsaufrufs). Der Typ der Rückgabewerts `ret` sollte logischerweise dem Rückgabetyp entsprechen, anderenfalls findet eine Konvertierung statt. Die Übergabeparameter können in den Anweisungen der Funktionsdefinition wie Variablen der entsprechenden Typen verwendet werden. Die reine Deklaration einer Funktion erfolgt durch:

```
rtyp fname(typ1, typ2, ... , typN);
```

Man beachte das notwendige Semikolon. Die spätere Definition der Funktion erfolgt dann nach der gleichen Syntax wie oben. Eine Trennung von Definition und Deklaration macht vor allem dann Sinn, wenn man sehr viele Funktionen in einer einzigen Datei definiert und diese Funktionen teilweise voneinander abhängen. Dann ist die notwendige Einhaltung der Reihenfolge der Deklarationen bei gleichzeitiger Definition nur noch schwer zu überblicken. Außerdem hilft eine separate Deklaration dabei, den Überblick über alle vorhandenen Funktionen zu behalten. Das gilt natürlich nur, wenn die Funktionen sinnvoll benannt wurden und die Deklarationen, falls sinnvoll, durch Kommentare ergänzt werden.

Zur Veranschaulichung betrachten wir ein einfaches Beispielprogramm, welches das Minimum und das Maximum zweier Zahlen jeweils in einer eigenen Funktion bestimmt:

```
 1 #include <iostream>
 2
 3 // bestimme Maximum zweier Zahlen
 4 double maximum(double x, double y)
 5 {
 6   if (x > y)
 7     return x;
 8   else
 9     return y;
10 }
11
12 // bestimme Minimum zweier Zahlen
13 double minimum(double x, double y)
14 {
15   if (x < y)
16     return x;
17   else
18     return y;
19 }
20
21 /* Hauptfunktion */
22 int main()
23 {
24   double zahl1, zahl2;
```

```
25    std::cout << "Gib zwei Zahlen ein: ";
26    std::cin >> zahl1 >> zahl2;
27    double maxi = maximum(zahl1,zahl2);
28    double mini = minimum(zahl1,zahl2);
29    std::cout << "Das Maximum ist " << maxi << std::endl;
30    std::cout << "Das Minimum ist " << mini << std::endl;
31    return 0;
32  }
```

Die Deklaration und Definition der Funktionen `minimum` und `maximum` findet ab den Zeilen 4 bzw. 13 statt. In den Zeilen 27 und 28 werden die Funktionen jeweils aufgerufen und ihre Rückgabewerte in den Variablen `mini` bzw. `maxi` gespeichert. Anschließend werden diese Werte ausgegeben.

Wenn man in diesem Programm die Deklaration der Funktionen von deren Definition trennen möchte, dann kann man das etwa wie im folgenden Beispiel realisieren:

```
1   #include <iostream>
2
3   // bestimme Maximum zweier Zahlen
4   double maximum(double, double);
5
6   // bestimme Minimum zweier Zahlen
7   double minimum(double, double);
8
9   /* Hauptfunktion */
10  int main()
11  {
12    double zahl1, zahl2;
13    std::cout << "Gib zwei Zahlen ein: ";
14    std::cin >> zahl1 >> zahl2;
15    double maxi = maximum(zahl1,zahl2);
16    double mini = minimum(zahl1,zahl2);
17    std::cout << "Das Maximum ist " << maxi << std::endl;
18    std::cout << "Das Minimum ist " << mini << std::endl;
19    return 0;
20  }
21
22  // Definition der Funktion maximum
23  double maximum(double x, double y)
24  {
25    if (x > y)
26      return x;
27    else
28      return y;
29  }
30
```

```
31 // Definition der Funktion minimum
32 double minimum(double x, double y)
33 {
34   if (x < y)
35     return x;
36   else
37     return y;
38 }
```

Hier erfolgen die Deklarationen in den Zeilen 4 und 7, die dazugehörenden Definitionen wurden jedoch hinter das Hauptprogramm verschoben und finden ab Zeile 23 bzw. 32 statt.

Hinweis: Bei der Parameterübergabe an eine Funktion werden im Normalfall Kopien der entsprechenden Werte übergeben. Das bedeutet, dass etwa eine Variable im Hauptprogramm ihren Wert behält, wenn sie als Parameter an eine Funktion übergeben wird und dieser Parameter innerhalb der Funktion verändert wird. Dies wird durch das folgende Beispielprogramm verdeutlicht:

```
1  #include <iostream>
2
3  float mal_drei(float x)
4  {
5    x*=3;
6    std::cout << "In der Funktion: x=" << x << std::endl;
7    return x;
8  }
9
10 int main()
11 {
12   float x, y;
13   std::cout << "Gib eine Zahl ein: ";
14   std::cin >> x;
15   y = mal_drei(x);
16   std::cout << "Im Hauptprogramm: x=" << x;
17   std::cout << ", y=" << y << std::endl;
18   return 0;
19 }
```

Hier haben wir für die Variable `x` sogar den gleichen Bezeichner wie für den Funktionsparameter `x` gewählt. Aufgrund der unterschiedlichen Gültigkeitsbereiche führt dies jedoch zu keinem Konflikt.

KAPITEL 2

Theoretische Grundlagen - Von der Binärzahl zum Transistor

Das wissenschaftliche Fachgebiet, das sich mit der Verwendung von Computern zur Automatisierung verschiedenster Aufgaben befasst, ist die *Informatik*. Der Duden gibt die folgende Definition: Informatik ist die »Wissenschaft von den elektronischen Datenverarbeitungsanlagen und den Grundlagen ihrer Anwendung«. Sie entstand aus der Notwendigkeit der schnellen und automatisierten Durchführung von Berechnungen und hat ihren historischen Ursprung einerseits in der Mathematik, andererseits in den Ingenieurswissenschaften. Die Informatik lässt sich in die folgenden Teilgebiete untergliedern:

- *Technische Informatik:* Die technische Informatik ist zum Beispiel mit der Entwicklung von Prozessoren und Speichermedien und allgemeiner mit der sogenannten *Hardware* befasst.
- *Praktische Informatik:* Gegenstand der praktischen Informatik ist beispielsweise die Konzeption von Algorithmen und deren Umsetzung in Programmen, also die Entwicklung von *Software*.
- *Theoretische Informatik:* Die theoretische Informatik untersucht unter anderem, ob gewisse Problemstellungen berechen- beziehungsweise entscheidbar sind, und entwickelt Kalküle für die Verifizierung der Korrektheit von Programmen.

Diese Beschreibungen sind natürlich nur beispielhaft. Wir wollen in den folgenden Abschnitten die theoretischen und technischen Grundlagen der Programmierung sowie der Funktionsweise eines Computers genauer erläutern.

2.1. Programmiersprachen und Algorithmen

Da Prozessoren nur Folgen von Nullen und Einsen verarbeiten können, ist die direkte Programmierung eines Prozessors kaum praktikabel. Programmiersprachen erlauben die automatisierte Übersetzung von verständlichen Kommandos in Maschinenbefehle. Eine Folge von Kommandos einer Programmiersprache bezeichnet man als *Programm*. Damit ein Prorgamm übersetzt werden kann, muss es bestimmten *syntaktischen* und *semantischen* Bedingungen genügen. Als Syntax einer Programmiersprache bezeichnet man dabei die durch sie vorgegebene Rechtschreibung und Grammatik, während man unter Semantik die inhaltlich korrekte Verwendung von Kommandos versteht. Im Kontext der deutschen Sprache ist beispielsweise die Aussage »die Straße ist eckig« syntaktisch korrekt nicht jedoch semantisch, wohingegen die Aussage »die Straße ist breit« unabhängig vom Wahrheitsgehalt beiden Anforderungen genügt.

Programme sind Realisierungen von *Algorithmen*. Als Algorithmus bezeichnen wir dabei eine eindeutige Handlungsvorschrift zur Lösung eines Problems, welche aus einer endlichen Folge

von elementaren, wohldefinierten Anweisungen besteht. Wie detailliert die einzelnen Schritte angegeben werden, hängt sehr vom Kontext ab und kann stark variieren. Die pq-Formel

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

zur Lösung der quadratischen Gleichung $x^2 + px + q = 0$ kann für sich schon als Algorithmus angesehen werden, allerdings lässt sie sich in dieser Form in den meisten Programmiersprachen nicht direkt implementieren. Etwas konkreter wäre die folgende Form:

Algorithmus 2.1 (pq-Formel)**.** *Eingabe:* Reelle Zahlen p und q.

(1) Setze $r = p^2/4 - q$.
(2) Ist $r < 0$, so gib die Fehlermeldung »nicht lösbar« aus und stoppe.
(3) Berechne $s = \sqrt{r}$.
(4) Setze $x_1 = -p/2 + s$ und $x_2 = -p/2 - s$.

Ausgabe: Lösungen x_1 und x_2.

Auch dieser Algorithmus ist nicht unbedingt direkt umsetzbar, da etwa die Quadratwurzel auf eine geeignete Weise berechnet werden muss. Dies kann beispielsweise effizient mit dem *Heronschen Verfahren* geschehen, welches Approximationen von $\sqrt{r}$ mittels der Initialisierung $s_0 = 1$ und der Iterationsvorschrift $s_{k+1} = \frac{1}{2}(s_k + \frac{r}{s_k})$ berechnet. Diese Vorschrift wird so lange angewendet, bis ein Abbruchkriterium erfüllt ist.

Algorithmus 2.2 (Heronsches Verfahren)**.** *Eingabe:* Reelle Zahl $r \geq 0$ und Toleranz $\delta > 0$.

(1) Setze $s_{\text{neu}} = 1$ und $s_{\text{alt}} = r$.
(2) Gilt $|s_{\text{neu}} - s_{\text{alt}}| < \delta$, so stoppe.
(3) Setze $s_{\text{alt}} = s_{\text{neu}}$ und anschließend $s_{\text{neu}} = \frac{1}{2}(s_{\text{alt}} + \frac{r}{s_{alt}})$.
(4) Gehe zu Schritt (2).

Ausgabe: Approximation s_{neu} von $s = \sqrt{r}$.

In diesem Algorithmus wird die Anweisung in Schritt (3) so lange wiederholt, bis eine Abbruchbedingung erfüllt ist. Alternativ könnte man zu einem gegebenen $N > 0$ die Iterationsvorschrift N-mal anwenden und s_N als Approximation der Quadratwurzel verwenden, allerdings ist a priori nicht klar, wie groß die Zahl N gewählt werden musst, um eine ausreichend gute Approximation zu erhalten.

2.2. Die Von-Neumann-Architektur

Um die Funktionsweise von Computern, die im Folgenden auch als Rechner bezeichnet werden, zu verstehen, ist es sinnvoll, sie auf ihre wesentlichen Bestandteile zu reduzieren. Das Modell des *Von-Neumann-Rechners* beschreibt einen Computer durch die Komponenten *Prozessor*, *Hauptspeicher* und *Ein- und Ausgabeeinheiten* sowie einen *Datenbus*, deren Zusammenhang in Abbildung 2.1 skizziert ist.

Das Herzstück eines Von-Neumann-Rechners ist der Prozessor, der auch als *Central Processing Unit (CPU)* bezeichnet wird und sämtliche Rechnungen durchführt. Im Hauptspeicher

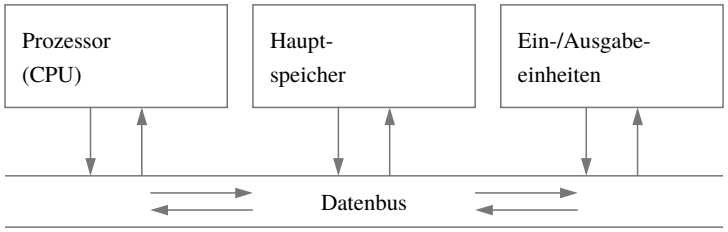

Abbildung 2.1. Aufbau des Von-Neumann-Rechners.

werden Programme und Daten abgelegt. Ein- und Ausgabeeinheiten wie Tastatur, Monitor, Drucker, Maus und weitere Speichermedien dienen der Interaktion mit dem Benutzer. Der Datenbus sorgt für den Transfer von Informationen zwischen den Komponenten.

2.3. Funktionsweise des Prozessors

Sämtliche Daten auf Computern liegen im Binärformat vor, das heißt sie werden in geeigneter Weise durch Folgen der Informationen 0 und 1 beziehungsweise durch entsprechende elektrische Signale realisiert. Der Prozessor eines Rechners besteht im Wesentlichen aus sogenannten *Transistoren*. Dies sind Halbleiter-Bauteile welche zum Steuern elektrischer Ströme und Spannungen dienen.

Eine einzelne 0/1-Information bezeichnet man als *Bit*. Auch das eigentliche Rechnen, also die Durchführung arithmetischer Operationen, erfolgt in geeigneter Weise mithilfe von Bits. Um dies zu realisieren, kombiniert man die folgenden Arbeitsschritte:

- Repräsentation von Dezimalzahlen im Dual- bzw. Binärsystem,
- Darstellung arithmetischer Operationen durch logische Ausdrücke,
- Technische Umsetzung mit Transistorschaltungen.

Die Vorgehensweise bei der Umsetzung dieser Schritte werden wir nun in den verbleibenden Abschnitten dieses Kapitels genauer erläutern.

2.4. Binärdarstellung natürlicher Zahlen

Jede natürliche Zahl inklusive Null lässt sich als Summe von Potenzen der Zahl 2 darstellen, das heißt für jedes $\ell \in \mathbb{N}_0$ existieren eine natürliche Zahl $k \in \mathbb{N}$ und Koeffizienten $b_0, b_1, \ldots, b_k \in \{0, 1\}$, so dass

$$\ell = b_k \cdot 2^k + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0.$$

Schreibt man die Zahlen $b_0, b_1, \ldots, b_k$ hintereinander, so erhält man die *Binärdarstellung* von ℓ, beispielsweise

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \equiv 1101$$

Insbesondere in der Programmierung und der technischen Informatik werden Binärdarstellungen häufig durch das Präfix $0\mathrm{b}$ oder das Suffix h gekennzeichnet, also beispielsweise

$$13 = 0\mathrm{b}1101 = 1101\mathrm{h}.$$

Das Dezimalsystem lässt sich übrigens genauso interpretieren, wenn anstatt der Basis 2 die Basis 10 verwendet wird. Jede Ziffer der Binärdarstellung bezeichnen wir als *Bit*. Fixieren wir die maximale Anzahl k zulässiger Bits, so können wir natürliche Zahlen im Bereich $0, 1, \ldots, 2^k - 1$ darstellen. Die Darstellung von Dezimalbrüchen mit einer vorgegebenen Anzahl m von Nachkommastellen lässt sich auf die Darstellung natürlicher Zahlen zurückführen, wenn wir alle solchen Dezimalzahlen mit 10^m multiplizieren. Zur Berücksichtigung negativer Zahlen können wir ein Bit zur Speicherung des Vorzeichens verwenden. Insgesamt lassen sich so arithmetische Operationen mit Dezimalzahlen auf das Rechnen mit Binärzahlen zurückführen.

2.5. Rechnen im Binärsystem

Die Addition zweier Binärzahlen erfolgt schriftlich mit Übertrag, im Beispiel $13 + 6$ ergibt sich:

$$\begin{array}{r} 1101 \\ +\;\;0110 \\ \hline \text{ü }1100 \\ \hline 10011 \end{array}$$

Die Subtraktion wird auf die Addition zurückgeführt. Dazu werden Minuend und Subtrahend durch Auffüllen mit Nullen auf gleiche Bitlänge gebracht und das additive Inverse des Subtrahenden wie folgt gebildet. Zunächst werden alle Bits invertiert, das heißt aus $8 \equiv 01000$ wird 10111, und anschließend wird 1 addiert, das heißt aus 10111 wird 11000, was wiederum der Dezimalzahl -8 in 5-Bit-Arithmetik entspricht. Diese Zahl wird zum Minuenden addiert. Für die Rechnung $14 - 8$ ergibt sich in Binärdarstellung also die Addition:

$$\begin{array}{r} 01110 \\ +\;\;11000 \\ \hline \text{ü }1000 \\ \hline \cancel{1}00110 \end{array}$$

Wir sehen dabei, dass das führende Bit bei fester Bit-Anzahl für das Vorzeichen einer Zahl steht. Zur Multiplikation einer Binärzahl mit einer Potenz der Zahl 2 werden durch Anhängen von Nullen die Bits um die Potenz nach links verschoben. Für die Rechnung $9 \cdot 4$ ergibt sich

$$1001 \cdot 2^2 = 100100$$

Mit Hilfe des Distributivgesetzes kann so die allgemeine Multiplikation von Binärzahlen auf die Addition zurückgeführt werden:

$$1001 \cdot 0101 = 1001 \cdot (2^2 + 2^0) = 100100 + 001001$$

Divisionen durch Potenzen der Zahl 2 können bei Vernachlässigung des Rests durch eine Verschiebung der Bits nach rechts realisiert werden, allerdings lassen sich damit nicht allgemeinere Divisoren realisieren. Die allgemeine Division wird auf eine Folge von Subtraktionen zurückgeführt, indem man zum Beispiel schriftlich wie im Dezimalsystem subtrahiert. Das Vorgehen ist nachfolgend am Beispiel $17 : 5 = 3$ Rest 2 gezeigt:

$$
\begin{array}{rl}
 & 10001 : 101 = 011 \text{ Rest } 010 \\
- & \underline{\quad 0} \\
 & 1000 \\
- & \underline{101} \\
 & \quad 111 \\
 & - \underline{101} \\
 & \quad 010
\end{array}
$$

Wir sehen also, dass sich sämtliche Rechnungen durch einfache Verschiebungen und Binäradditionen realisieren lassen.

2.6. Aussagenlogik

Die Aussagenlogik bildet die theoretische Grundlage für die technischen Umsetzung von Rechenoperationen im Binärsystem. Für die Herleitung geeigneter logischer Ausdrücke werden die Gesetze der *Booleschen Algebra* genutzt. Dabei stehen die Werte 0 und 1 für den Wahrheitswert einer Aussage und repräsentieren die Bewertungen falsch und wahr. Beispiele für Aussagen sind »Heute scheint die Sonne« und »Morgen ist Dienstag«, die abhängig vom Ort und Zeitpunkt ihrer Auswertung falsch oder wahr sein können. Besonders interessant sind Verknüpfungen von Aussagen. Sind A und B zwei Aussagen, so ist beispielweise die UND-Verknüpfung $A \wedge B$ eine Aussage, die dann und nur dann wahr ist, wenn A und B gleichzeitig wahr sind. Weitere sind das nicht-ausschließende ODER, in Zeichen $A \vee B$, sowie die Implikation $A \implies B$. Die in Tabelle 2.1 dargestellte *Wahrheitstabelle* gibt an, wie diese Verknüpfungen definiert sind.

A	B	$A \wedge B$	$A \vee B$	$A \implies B$	$\neg A$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	0
1	1	1	1	1	0

Tabelle 2.1. Wertetabelle zur Definition der logischen Verknüpfungen UND, ODER, IMPLIZIERT und NEGATION. Für verschiedene Belegungen der booleschen Variablen A und B sind die Ausdrücke $A \wedge B$, $A \vee B$, $A \implies B$ und $\neg A$ entweder falsch oder wahr beziehungsweise haben den Wert 0 oder 1.

Während die ODER-Verknüpfung gut nachvollziehbar ist, da sie genau dann wahr ist, wenn mindestens eine der beiden Aussagen wahr ist, ist die Festlegung der Implikation zunächst überraschend, insbesondere, dass aus einer falschen Aussage eine wahre folgen darf. Dies ist durchaus sinnvoll, denn aus einer falschen Voraussetzung kann man keine sinnvollen Folgerungen ableiten. Entscheidend für die Gültigkeit der Implikation $A \implies B$ ist, dass B immer dann wahr ist, wenn A wahr ist. Das macht man sich am besten an einfachen Aussagen wie $A =$ »Es regnet« und $B =$ »Die Straße wird nass« sowie der Implikation

$$(A \implies B) = \text{»Wenn es regnet, dann wird die Straße nass«}$$

klar. Diese Implikation bleibt wahr, selbst in einer Welt, in der es aufgrund des Klimawandels nur noch Wüsten und keinerlei Niederschläge mehr gibt. Ein anderes Beispiel wäre die

Aussage »Wenn Freiburg in der Antarktis liegt, dann ist Schnee grün«. Auch bei dieser Implikation handelt es sich um eine wahre Aussage, zumindest solange, wie die Stadt Freiburg in Mitteleuropa liegt und der Planet Erde seine Orientierung und Umlaufbahn relativ zur Sonne beibehält.

Die Verknüpfungen $\wedge$ und $\vee$ erfüllen Kommutativ-, Assoziativ- und Distributivgesetze sowie unter Hinzunahme der Negation, die durch $\neg A$ dargestellt wird, die sogenannten *DeMorganschen Gesetze*

$$\neg(A \wedge B) = \neg A \vee \neg B, \quad \neg(A \vee B) = \neg A \wedge \neg B.$$

Dabei bedeutet Gleichheit zweier Ausdrücke, dass sie für jede Belegung der beteiligten Variablen denselben Wahrheitswert ergeben. Mit den Konstanten 0 und 1 bezeichnet man in Ausdrücken die Aussagen, die stets falsch beziehungsweise wahr sind. Mit ihnen gilt beispielsweise $A \wedge 1 = A$ und $A \vee 0 = A$ sowie $A \vee \neg A = 1$.

2.7. Addition mit logischen Operationen

Wir identifizieren Belegungen von Bits mit den logischen Werten falsch und wahr und versuchen, die Addition zweier Bits mit Übertrag durch einen logischen Ausdruck darzustellen. Die Wertetabelle für die Addition $A + B$ mit Summe S und Übertrag (*Carry Over Bit*) C ist in Tabelle 2.2 dargestellt. Die Operation $(A, B) \mapsto (S, C)$ wird als *Halbaddierer* bezeichnet.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabelle 2.2. Die als Halbaddition bezeichnete Addition zweier Bits A und B ergibt die Summe S und den Übertrag C.

Um logische Ausdrücke zu erhalten, verknüpfen wir die Zeilen, die den Wert 1 ergeben, jeweils mit der ODER-Operation und erhalten

$$\begin{aligned} S &= (\neg A \wedge B) \vee (A \wedge \neg B) = A \vee_! B, \\ C &= A \wedge B, \end{aligned}$$

wobei der Ausdruck für S dem ausschließenden ODER, das hier durch $\vee_!$ symbolisiert wird, entspricht.

Als nächstes betrachten wir die Addition von drei Bits A, B und C_{in}, wobei C_{in} ein Übertragsbit aus der Addition zweier niederer Bits sei. Dies ergibt ein Summationsbit S und ein übertragsbit C_{out}, deren Werte in Tabelle 2.3 aufgeführt sind. Die Operation $(A, B, C_{\text{in}}) \mapsto (S, C_{\text{out}})$ wird als *Volladdierer* bezeichnet.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabelle 2.3. Die als Volladdierer bezeichnete Addition dreier Bits A, B und C_{in} ergibt die Summe S und den übertrag C_{out}.

Aus der Wertetabelle ergeben sich die Ausdrücke

$$\begin{aligned} S &= (\neg A \wedge \neg B \wedge C_{\text{in}}) \vee (\neg A \wedge B \wedge \neg C_{\text{in}}) \\ &\quad \vee (A \wedge \neg B \wedge \neg C_{\text{in}}) \vee (A \wedge B \wedge C_{\text{in}}) \\ &= (A \vee_! B) \vee_! C_{\text{in}} \\ C_{\text{out}} &= (\neg A \wedge B \wedge C_{\text{in}}) \vee (A \wedge \neg B \wedge C_{\text{in}}) \\ &\quad \vee (A \wedge B \wedge \neg C_{\text{in}}) \vee (A \wedge B \wedge C_{\text{in}}) \\ &= ((A \vee_! B) \wedge C_{\text{in}}) \vee (A \wedge B). \end{aligned}$$

Die Volladdition können wir mit Hilfe zweier Halbaddierer und einer ODER-Operation darstellen, wie in Abbildung 2.2 gezeigt ist.

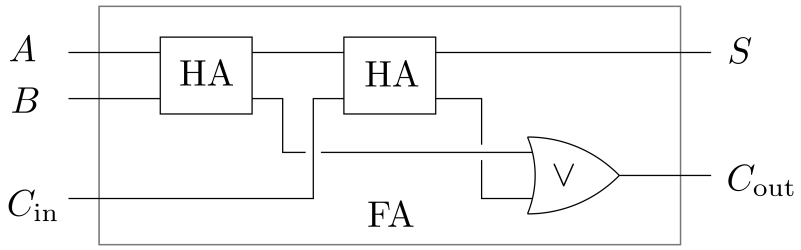

Abbildung 2.2. Schematische Konstruktion eines Volladdierers (FA) mittels zweier Halbaddierer (HA) und einer ODER-Operation.

Sollen nun die Binärzahlen $a = a_k \dots a_1 a_0$ und $b = b_k \dots b_1 b_0$ addiert werden, so kann dies mit einem Halbaddierer für die niedersten Bits sowie $k-1$ Volladdierern für die Addition der höheren Bits unter Berücksichtigung der vorigen Überträge geschehen. Die Realisierung ist in Abbildung 2.3 gezeigt.

Die in Abbildung 2.3 gezeigte Schaltung ist einfach zu realisieren, aber nicht besonders effizient, da die Volladdierer jeweils das Übertragsbit des vorigen Voll- beziehungsweise Halbaddierers benötigen. Um die Rechenzeit zu verkürzen, halbiert man die Bitfolgen und berechnet

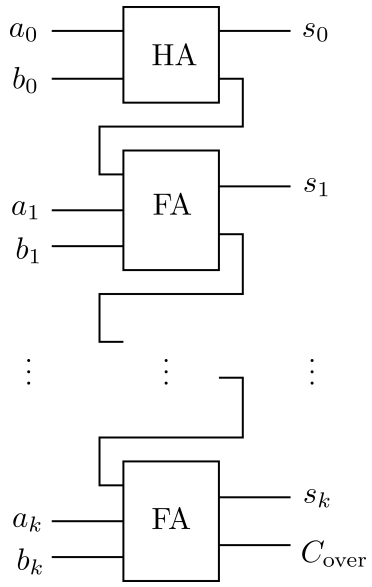

Abbildung 2.3. Addition zweier k-stelliger Binärzahlen mittels eines Halbaddierers (HA) und $(k-1)$ Volladdierern (FA) unter Berücksichtigung eines *Overflow*-Bits C_{over}.

die Summe der niederen Bits

$$a_{k/2} \dots a_0 + b_{k/2} \dots b_0 = [S_{\text{low}}, C_{\text{low}}]$$

sowie parallel dazu zwei Summen der höheren Bits, einmal mit und einmal ohne Übertragsbit, das heißt

$$0 + a_k \dots a_{k/2+1} + b_k \dots b_{k/2+1} = [S^0_{\text{high}}, C^0_{\text{over}}]$$
$$1 + a_k \dots a_{k/2+1} + b_k \dots b_{k/2+1} = [S^1_{\text{high}}, C^1_{\text{over}}].$$

Die korrekte Summe ist $S^0_{\text{high}} S_{\text{low}}$ mit Überlaufbit C^0_{over} falls $C_{\text{low}} = 0$ und $S^1_{\text{high}} S_{\text{low}}$ mit Überlaufbit C^1_{over} anderenfalls. Diese Entscheidung wird von einem sogenannten *Multiplexer* durchgeführt. Durch wiederholte Anwendung dieser Argumentation lässt sich die Rechenzeit mehrfach nahezu halbieren.

2.8. Realisierung mit Transistorschaltungen

Einzelne Bits beziehungsweise die logischen Werte falsch und wahr lassen sich sinnvoll mit elektrischen Signalen darstellen, das heißt an einer Leitung liegt entweder eine Spannung an und es fließt Strom oder nicht. Da sich jede logische Operation mittels der Grundfunktionen UND, ODER sowie NEGATION darstellen lässt, diskutieren wir nur deren technischen Realisierungen.

Mit Transistoren kann über einen kleinen Stromfluss ein größerer kontrolliert werden, ähnlich wie es in Abbildung 2.4 für zwei Wasserkanäle illustriert ist. Liegt an der Basis eine Spannung an, so wird der Stromfluss vom Kollektor zum Emitter ermöglicht.

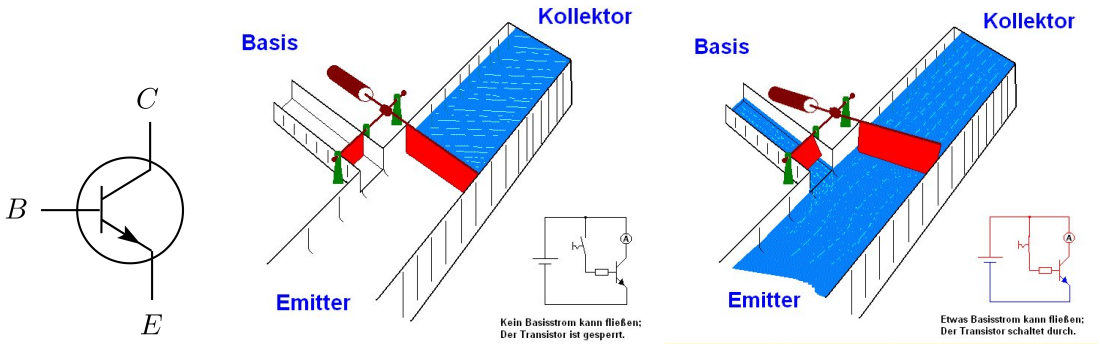

Abbildung 2.4. Schaltsymbol und Funktionsweise eines Transistors: ein kleiner Stromfluss von der Basis zum Emitter ermöglicht einen größeren vom Kollektor zum Emitter. (Quelle: Stefan Riepl (`https://commons.wikimedia.org/wiki/File:Transistor_animation.gif`), Lizenz: CC BY-SA 2.0 DE)

Abbildung 2.5 zeigt Transistorschaltungen für die logischen Grundoperationen. Die UND-Schaltung ist leicht verständlich: nur wenn an den Basen der beiden Transistoren Spannungen anliegen, das heißt nur wenn A und B wahr sind, kann Strom von der Spannungsquelle zur Erdung fließen und am Ausgang $A \wedge B$ liegt eine Spannung an. Bei der ODER-Schaltung überprüft man, dass am Ausgang Spannung anliegt, sofern mindestens einer der Transistoren durchgeschaltet ist, also an Eingang A oder Eingang B Spannung anliegt. Etwas anders ist die Funktionsweise der NEGATION-Schaltung. Liegt am Eingang A eine Spannung an, so fließt der gesamte Strom aufgrund des Widerstands R2 zur Erdung und es liegt keine Spannung am Ausgang an. Tatsächliche Realisierungen sind in der Regel etwas komplexer, da beispielsweise wiederholte Spannungsabfälle vermieden werden müssen.

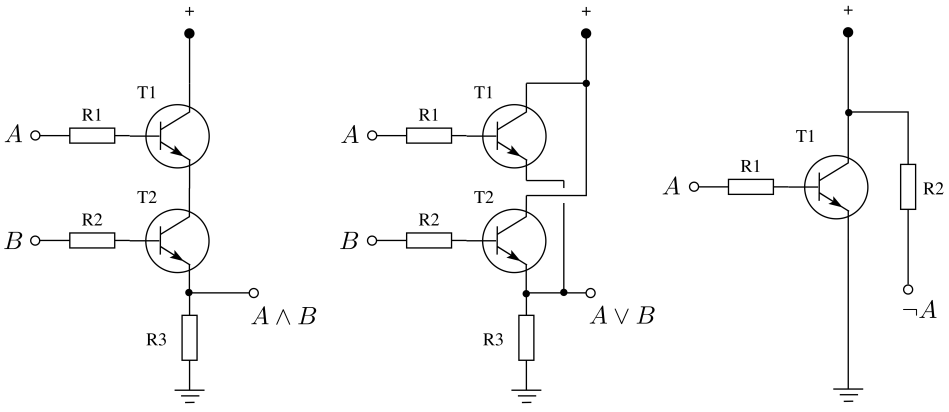

Abbildung 2.5. Transistorschaltungen für die konzeptionelle Realisierung der logischen Operationen UND, ODER und NEGATION.

KAPITEL 3

Fortgeschrittene Konzepte in C++

Wir behandeln in diesem Kapitel weitere Aspekte der Programmierung in C++, wie das Arbeiten mit Zeigern, Zeichenketten und Vektoren variabler Länge sowie die Ein- und Ausgabe in Dateien.

3.1. Zeiger und Referenzen

Das Konzept der *Zeiger* (engl. *Pointer*) unterscheidet die Sprache C++ von vielen anderen Programmiersprachen. Ein Zeiger ist eine Speicheradresse im Hauptspeicher des Computers und die Verwendung von Zeigern erlaubt es, sehr effiziente Programme zu schreiben. Unter anderem ermöglichen sie die dynamische Allokation von Speicherplatz zur Laufzeit eines Programms.

Zeiger. Um die Verwendung von Zeigern zu verstehen, betrachten wir zunächst ein vereinfachtes Speichermodell eines hypothetischen Rechners:

Variable	*Speicheradresse*	*Inhalt*
...	...	...
i	1773	5
j	1774	7
k	1775	2
p	1776	1773
...	...	...

Generell besteht ein Speicherinhalt lediglich aus einer Folge von Bits und man kann ihm nicht ansehen, ob es sich dabei um Daten, Speicheradressen oder auch Maschinenbefehle handelt. In höheren Programmiersprachen arbeitet man meistens nicht mit den tatsächlichen Speicheradressen, sondern stellvertretend mit den Bezeichnern der entsprechenden Variablen, wodurch implizit klar ist, welchen Typ der Inhalt des entsprechenden Speicherplatzes hat. Im Speicher unseres hypothetischen Rechners könnten beispielsweise drei `int`-Variablen abgelegt sein, welche durch

```
int i=5, j=7, k=2;
```

deklariert und initialisiert wurden.

Eine *Zeigervariable* dient nun dem Speichern einer Speicheradresse für einen bestimmten Typ. Die Deklaration einer Zeigervariable geschieht durch Angabe des Typs und eines Bezeichners mit vorangestelltem `*` nach der Syntax

```
TYP *bezeichner;
```

Eine Zeigervariable p zum Speichern der Adresse einer **int**-Variable wird beispielsweise mit der Anweisung

```
int *p;
```

deklariert. Die Speicheradresse einer deklarierten Variablen lässt sich mithilfe des *Adressoperators* & ermitteln. Im Speicher unseres hypothetischen Rechners enthält die Zeigervariable p die Adresse der Variable i. Ist die Zeigervariable p wie oben deklariert, erfolgt diese Zuweisung durch

```
p = &i;
```

Häufig sagt man in diesem Fall: »p zeigt auf i.«

Möchte man auf den Wert im Speicher zugreifen, der sich an der in einer Zeigervariablen gespeicherten Adresse befindet, so verwendet man dazu den *Dereferenzierungsoperator* *. Die Anweisung

```
*p = 0;
```

würde beispielsweise den Wert der Variablen i auf 0 setzen. Zeiger können bei Bedarf verändert werden, etwa könnten wir durch die beiden Anweisungen p=&j; und *p=0; auch den Wert der Variable j auf 0 setzen.

Das folgende Beispielprogramm fasst die Verwendung von Zeigern noch einmal kurz zusammen:

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5   int wert1, wert2;
 6   int * zeiger1, * zeiger2;
 7
 8   zeiger1 = &wert1; // zeiger 1 zeigt auf wert1
 9   *zeiger1 = 50;
10   zeiger1 = &wert2; // zeiger 1 zeigt jetzt auf wert2
11   zeiger2 = zeiger1; // zeiger 2 zeigt auch auf wert2
12   *zeiger2 = 100;
13   zeiger1 = &wert1; // zeiger1 zeigt wieder auf wert1
14
15   std::cout << wert1 << std::endl; // Ausgabe: 50
16   std::cout << zeiger1 << std::endl; // Ausgabe: Adresse von wert1
17   std::cout << wert2 << std::endl; // Ausgabe: 100
18   std::cout << zeiger2 << std::endl; // Ausgabe: Adresse von wert2
19   return 0;
20 }
```

Hier werden in Zeile 6 zwei Zeiger auf noch nicht initialisierte Variablen vom Typ **int** deklariert. Anschließend werden die Zeiger verwendet, um den Variablen Werte zuzuweisen. Die zugewiesenen Werte sowie die dazugehörenden Speicheradressen (im Hexadezimalsystem) werden anschließend ausgegeben.

Hinweis: Die kleinste adressierte Speichereinheit auf modernen Systemen ist in der Regel ein Byte (=8 Bit). Dies entspricht normalerweise genau dem Speicherbedarf des Typs **char**. Eine Variable vom Typ **float**, die beispielsweise einen Speicherbedarf von 4 Byte (=32 Bit) hat, ist also tatsächlich in vier aufeinanderfolgenden Speicherplätzen abgelegt. Eine entsprechende Zeigervariable enthält dann immer die Adresse des ersten Speicherplatzes, der zur Speicherung der Variable verwendet wird.

Referenzen. Mit Zeigern eng verwandt und eine Erweiterung von C++ gegenüber klassischem C sind sogenannte *Referenzen*. Auch eine Referenzvariable verweist intern auf den Speicherplatz einer Variablen. Im Gegensatz zu Zeigern müssen Referenzen jedoch direkt initialisiert werden und können während ihres Bestehens nicht mehr verändert werden, das heißt sie verweisen immer auf dieselbe Speicheradresse. Die Deklaration und Initialisierung einer Referenz erfolgt durch Angabe eines Typs und eines Bezeichners mit vorangestelltem `&` mittels

```
TYP &bezeichner = variable;
```

Eine Referenzvariable `r` mit Verweis auf die **int**-Variable `k` erstellt man beispielsweise durch

```
int &r = k;
```

Referenzvariablen werden ohne Zeigersyntax verwendet und verhalten sich so, als ob sie das referenzierte Objekt selbst wären. Sie können daher als Synonyme zu den ursprünglichen Variablen betrachtet werden. Durch die Anweisung

```
r = 0;
```

könnten wir also nun den Wert der Variablen `k` auf 0 setzen.

Referenzparameter in Funktionen. Eine typische Anwendung von Zeigern findet sich in der Parameterübergabe an Funktionen. Klassischerweise werden bei einem Funktionsaufruf Kopien der Parameterwerte übergeben, was auch als *call-by-value* bezeichnet wird (vgl. Hinweis auf Seite 18). Dies hat zur Folge, dass die entsprechenden Werte im aufrufenden Programm nicht verändert werden können und stellt gleichzeitig einen gewissen Schutzmechanismus dar, da das Hauptprogramm dadurch von unerwünschten Veränderungen abgeschirmt wird.

Allerdings hat diese Vorgehensweise einen erhöhten Speicherbedarf zur Folge, denn die entsprechenden Kopien der Werte müssen separat im Speicher abgelegt werden. Möchte man dies Umgehen oder aus sonstigen Gründen einer Funktion das Verändern der Parameterwerte im Hauptprogramm ermöglichen, beispielsweise weil mehr als ein Rückgabewert gewünscht ist, so kann man dies Umsetzen, indem man anstatt der Variablen selbst entsprechende Zeiger beziehungsweise Referenzen an die Funktion übergibt. Dieses Vorgehen wird als *call-by-reference* bezeichnet und ist im folgenden Beispiel illustriert:

```
1  #include <iostream>
2
3  void setze_null( int &a ) {
4    a = 0;
5  }
6
7  void inkr( int *b ) {
8    *b = *b + 1;
9  }
10
11 int main()
12 {
13   int x = 10;
14   setze_null(x);
15   std::cout << x << std::endl;
16
17   int *p = &x;
18   inkr( p );
19   std::cout << x << std::endl;
20
21   return 0;
22 }
```

Im obigen Code bekommt die Funktion `setze_null()` eine Referenz auf eine `int`-Variable übergeben und setzt den Wert dieser Variable auf 0. Nach dem Funktionsaufruf im Hauptprogramm in Zeile 14 ist der Wert der Variable `x` dementsprechend 0. Die Funktion `inkr()` bekommt bei ihrem Aufruf in Zeile 18 einen Zeiger auf die `int`-Variable `x` übergeben und erhöht den Wert der Variable um 1. Dementsprechend ist der Wert von `x` nach dem Funktionsaufruf 1.

Achtung: Effektiv macht es keinen Unterschied, ob man sich für die Übergabe eines Zeigers oder einer Referenz entscheidet. Bei Übergabe einer Referenz muss man jedoch beachten, dass beim Aufruf im Hauptprogramm in keinster Weise ersichtlich ist, ob der entsprechende Parameter nun als Kopie oder als Referenz übergeben wird und sich sein Wert durch den Funktionsaufruf eventuell ändert. Generell stellt die Arbeit mit Zeigern beziehungsweise Referenzen eine häufige Fehlerquelle dar. Fehler, die dabei entstehen, sind häufig nur sehr schwer zu finden. Man sollte die Konzepte also gut verstanden haben und auf eine präzise Arbeitsweise achten.

Felder und Zeiger. Wir haben in Abschnitt 1.5 bereits die Verwendung von Feldern zum Zusammenfassen mehrerer Variablen gleichen Typs kennengelernt. Die Konzepte von Feldern und Zeigern sind in C++ eng miteinander verknüpft. Tatsächlich entspricht der Bezeichner eines Feldes gerade einem Zeiger auf das erste Element und es handelt sich bei beiden Konstrukten um ein und dasselbe Konzept. Angenommen, in einem Quelltext sind die Deklarationen

```
float zahlen[5];
```

```
float *p;
```

gegeben. Dann ist die Zuweisung

```
p = zahlen;
```

eine gültige Anweisung und die Bezeichner `p` und `zahlen` sind im weiteren Verlauf des Programms äquivalent und können synonym verwendet werden. Beispielsweise könnte man mit `p[2]` auf den dritten und mit `*zahlen` auf den ersten Eintrag des Feldes zugreifen, da sowohl `p` als auch `zahlen` die Speicheradressen des ersten Eintrags enthalten.

Eine weitere Konsequenz ergibt sich aus der *Zeigerarithmetik*: Nach einer Erhöhung des Werts von `p` mittels

```
p++;
```

oder

```
p = p+1;
```

wird der tatsächlich in `p` gespeicherte Wert nicht um 1, sondern genau um **`sizeof(float)`** erhöht und `p` enthält danach die Speicheradresse des zweiten Elements aus dem Feld. Analog könnte man der Zeigervariablen `p` durch

```
p = (zahlen+3);
```

die Speicheradresse des vierten Elements des Feldes, also die Adresse von `zahlen[3]`, zuweisen oder mittels `*(zahlen+3)` auf dieses Element zugreifen. Damit ist nun auch klar, weswegen bei der Deklaration von Zeigervariablen der jeweilige Typ angegeben werden muss. Für eine sinnvolle Zeigerarithmetik müssen die Zeiger den Speicherbedarf der gespeicherten Werte kennen.

Als zusammenfassendes Beispiel soll das folgende Programm dienen:

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5    int feld[3] = {5,7,9};
 6    int *p;
 7    p = feld;
 8
 9    std::cout << p[0] << std::endl;
10    std::cout << p[1] << std::endl;
11    std::cout << p[2] << std::endl << std::endl;
12
13    std::cout << *p << std::endl;
14    p++;
15    std::cout << *p << std::endl;
16    std::cout << *(feld+2) << std::endl;
17
```

```
18    return 0;
19  }
```

Hier wird in den Zeilen 5–7 ein äquivalentes Feld-Zeiger-Paar erzeugt. Anschließend werden die gespeicherten Werte zwei mal ausgegeben, wobei in der ersten Ausgabe der Zugriffsoperator [] und in der zweiten Ausgabe Zeigeroperatoren verwendet werden.

Achtung: Die Äquivalenz von Zeigern und Feldern führt dazu, dass Felder generell als Referenz und nicht als Kopie an Funktionen übergeben werden. Jede Änderung an einem Feld in einer Unterfunktion wirkt sich also auf das entsprechende Feld im Hauptprogramm aus!

Die Schlüsselwörter `new` und `delete`. Bis jetzt hatten wir in allen Programmen immer genau so viel Speicherplatz zur Verfügung, wie zur Speicherung der von uns deklarierten Variablen benötigt wurde. Häufig kennt man den genauen Speicherbedarf jedoch erst zur Laufzeit eines Programms und nicht bereits bei der Kompilierung, etwa, wenn nicht klar ist, wie viele Daten in einem Programm verarbeitet werden sollen. Die *dynamische Verwaltung* von Speicherplatz kann man mithile der Operatoren **`new`** und **`delete`** realisieren.

Um dynamischen Speicherplatz für eine Variable anzufordern, verwendet man einen noch nicht definierten Zeiger vom Typ der Variable und das Schlüsselwort **`new`**, gefolgt vom Typ der Variable:

```
TYP *zeiger;
zeiger = new TYP;
```

Dieser Code bewirkt, dass im Hauptspeicher des Computers der Speicherplatz für eine Variable des Typs `TYP` dynamisch, also zur Laufzeit des Programms, reserviert wird. Man bezeichnet dies als *Speicherallokation*. Mithilfe des Schlüsselworts **`delete`** kann man den allozierten Speicher später wieder freigeben:

```
delete zeiger;
```

Die Verwendung von **`new`** ist zwingend notwendig, wenn man mit einem Zeiger arbeiten möchte, der nicht durch die Speicheradresse einer bereits deklarierten Variablen gegeben ist. Betrachten wir das folgende kurze Programm:

```
1  int main()
2  {
3    float *p; // deklariere Zeiger
4    *p = 3.14; // fuehrt zu Speicherzugriffsfehler
5    return 0;
6  }
```

Dieses Programm wird vom Compiler ohne Fehlermeldung übersetzt. Jedoch tritt zur Laufzeit des Programms ein Speicherzugriffsfehler auf, da die Anweisung in Zeile 4 auf einen noch nicht zugewiesenen Speicherplatz zuzugreifen möchte. Der Operator **`new`** schafft Abhilfe:

```
1    int main()
2    {
3      float *p; // deklariere Zeiger
4      p = new float; // weise Speicherplatz zu
```

```
5      *p = 3.14 // schreibe Wert in Speicher
6      delete p; // gebe Speicher wieder frei
7      return 0;
8    }
```

Dieses Programm wird fehlerfrei ausgeführt, da die **new**-Anweisung in Zeile 4 dazu führt, dass der entsprechende Speicherplatz reserviert wird. Die **delete**-Anweisung in Zeile 6 gibt den Speicherplatz wieder frei.

Möchte man zur dynamischen Speicherung eines Feldes mehrere zusammenhängende Speicherplätze gleichzeitig allozieren, so geschieht dies nach der Syntax

```
zeiger = new TYP[elementanzahl];
```

Die spätere Freigabe des allozierten Speichers sollte in diesem Fall durch

```
delete [] zeiger;
```

erfolgen. Das nachfolgende Beispielprogramm ermittelt die dynamische Größe eines Feldes exemplarisch über eine Benutzereingabe in der Konsole:

```
1  #include <iostream>
2
3  int main()
4  {
5    int dim;
6    std::cout << "Dimension eingeben: ";
7    std::cin >> dim;
8    int * a;
9    a = new int[dim];
10   for ( int i = 0; i < dim; ++i ) {
11     a[i] = i;
12   }
13   for ( int i = 0; i < dim; ++i ) {
14     std::cout << "a[" << i << "] = " << a[i] << std::endl;
15   }
16   delete [] a;
17   return 0;
18 }
```

Der Speicher zur Bereitstellung der vom Benutzer gewünschten Feldgröße wird hier in Zeile 9 alloziert. Anschließend wird das Feld mit Werten gefüllt und diese ausgegeben. In Zeile 16 wird der Speicherplatz wieder freigegeben.

Hinweis: Manche neueren Compiler erlauben die Deklaration sogenannter *Variable Length Arrays*, das sind Felder, deren Größe durch eine Variable angegeben wird, etwa durch die Anweisung `int feld[N];`, wobei der Wert von `N` zur Laufzeit des Programms vom Benutzer eingelesen wird. Von der Verwendung solcher Konstruktionen wird aber dringend abgeraten, da diese Felder in einem speziellen Speicher (dem sogenannten *Stack*) abgelegt werden. Dieser ist sehr begrenzt, was zu nahezu unauffindbaren Fehlern führen kann.

Zeiger auf Funktionen. Der Vollständigkeit halber seien an dieser Stelle noch Funktionszeiger erwähnt. Auch Funktionen liegen nach ihrer Übersetzung als Bitfolge im Speicher des Computers vor. Konsequenterweise kann man auch mit Zeigern auf Funktionen arbeiten. Ähnlich der Äquivalenz von Feldern und Zeigern entspricht der Bezeichner einer Funktion gewissermaßen einem Zeiger auf ihren Anfang im Speicher. Das Vorgehen ist beispielhaft in dem folgenden Programm veranschaulicht:

```
1  #include <iostream>
2
3  int addition( int a, int b ) {
4    return(a+b);
5  }
6
7  int subtraktion( int a, int b ) {
8    return(a-b);
9  }
10
11 int operation( int a, int b, int (*funktion)(int,int) ) {
12   return( funktion(a,b) );
13 }
14
15 int main()
16 {
17   int m,n;
18   int a=5, b=3;
19   m = operation(a,b,addition);
20   n = operation(a,b,subtraktion);
21   std::cout << m << std::endl << n << std::endl;
22   return 0;
23 }
```

Die Funktion `operation`, die ab Zeile 11 definiert wird, hängt zusätzlich zu den zwei `int`-Parametern `a` und `b` noch von einer zunächst unbekannten Funktion `funktion` mit Rückgabetyp `int` ab, welche selbst wiederum von zwei Parametern des Typs `int` abhängt. Der Rückgabewert von `operation` entpricht dem Rückgabewert von `funktion( a, b )`. Im Hauptprogramm wird die Funktion `operation` in den Zeilen 19 und 20 zwei mal aufgerufen, wobei sie einmal die Funktion `addition` und einmal die Funktion `subtraktion` als Parameter übergeben bekommt.

Bei Verwendung eines modernen Compilers kann man den $*$-Operator in Zeile 11 auch durch den Adressoperator & ersetzen oder sogar ganz weglassen. Intern wird der Code vom Compiler dann trotzdem so angepasst, dass ein Zeiger auf die Funktion übergeben wird.

3.2. Zeichenketten

Unter einer *Zeichenkette* (engl. *String*) versteht man eine Folge von Zeichen, also eine Folge von Werten des Typs **char**. Konstante Zeichenketten, die von doppelten Anführungszeichen umschlossen sind, kennen wir bereits seit unserem allerersten Beispielprogramm, dessen Funktion darin bestand, die Zeichenkette

```
"Hello World!"
```

auf dem Bildschirm auszugeben. Konstante Zeichenketten werden auch als *String-Literale* bezeichnet. Die einfachste Möglichkeit zur Speicherung von Zeichenketten in C++ ist die Verwendung von Arrays des Typs **char**. Beispielsweise wird durch die Anweisung

```
char s[] = "Hello";
```

ein **char**-Feld der Länge 6 mit dem folgenden Inhalt erstellt:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
'H'	'e'	'l'	'l'	'o'	'\0'

Der letzte Eintrag `'\0'` wird als *NULL character* bezeichnet und kennzeichnet das Ende der Zeichenkette. Die Arbeit mit **char**-Arrays ist jedoch mühsam. Etwa kann man ein einmal deklariertes Feld nur elementweise ändern. Die Anweisungen

```
char s[] = "Max";
s = "Tom"; // Fehler beim Kompilieren
```

würden vom Compiler nicht akzeptiert werden. Stattdessen müsste man folgenden Code verwenden, um den gewünschten Effekt zu erzielen:

```
char s[] = "Max"
s[0] = 'T';
s[1] = 'o';
s[2] = 'm';
```

Um die Arbeit mit Strings komfortabler zu gestalten, bietet sich daher die Verwendung der Klasse `std::string` an. Dazu ist das Einbinden der Standardbibliothek `<string>` notwendig. Was genau eine Klasse ist, werden wir in einem späteren Kapitel besprechen. Für den Moment betrachten wir sie einfach als einen weiteren Variablentyp und nehmen hin, dass wir die entsprechenden Variablen auch als Objekte bezeichnen. Ein Objekt vom Typ `std::string` kann so verwendet werden, wie wir es bereits von den klassischen Variablentypen kennen. Der Quelltext

```
std::string s;
s = "Tom\n"
std::cout << s;
s = "Maximilian";
std::cout << s << std::endl;
```

deklariert beispielsweise ein Objekt des Typs `std::string`. Anschließend bekommt dieses Objekt den Wert `"Tom\n"` zugewiesen. Die Ausgabe des Objekts `s` führt also zur Anzeige

der Zeichenkette "Tom" in der Konsole, gefolgt von einem Zeilenumbruch. Anschließend wird dem Objekt den Wert "Maximilian" zugewiesen. Nun führt die Ausgabe von s zur Anzeige der Zeichenkette "Maximilian" in der Konsole, diesmal zunächst ohne einen Zeilenumbruch '\n', welcher nachträglich durch std::endl angehängt wird.

Mit std::string-Objekten können unter anderem die folgenden Operationen durchgeführt werden:

Operator	Effekt
=	Zuweisung
+	Verknüpfung zweier String
+=	Anhängen eines Strings
<<	Ausgabe
>>	Eingabe
[]	Zugriff auf einzelne Zeichen
> bzw. <	lexikographischer Vergleich

Tabelle 3.1. Operationen mit Objekten vom Typ std::string.

Außerdem kann man die Länge der in deinem std::string-Objekt s gespeicherten Zeichenkette mit der Methode length() ermitteln und beispielsweise durch

```
int laenge = s.length();
```

in einer int-Variable mit Namen laenge speichern.

Die Verwendung der Operatoren aus Tabelle 3.1 ist exemplarisch im folgenden Beispielprogramm veranschaulicht:

```
 1 #include <iostream>
 2 #include <string>
 3
 4 int main()
 5 {
 6   std::string wort_1 = "Hallo";
 7   std::string wort_2 = "Welt!";
 8   std::string wort_3;
 9   if ( wort_1 < wort_2 )
10     wort_3 = wort_1 + ", " + wort_2;
11   std::cout << wort_3 << std::endl;
12   std::cout << wort_3.length() << std::endl;
13   wort_3[7]='Z';
14   std::cout << wort_3 << std::endl;
15 }
```

Der lexikographische Vergleich in Zeile 9 liefert den Rückgabewert true, da "Hallo" in einem Wörterbuch vor "Welt!" stehen würde. Deswegen wird die Anweisung in Zeile 10 ausgeführt und das Objekt wort_3 erhält den Wert "Hallo, Welt!", welcher in Zeile 11 ausgegeben wird. Diese Zeichenkette besteht aus 12 Zeichen, daher führt Zeile 12 zur Ausgabe des Werts

12. In Zeile 13 wird das siebte Zeichen des Strings durch 'Z' ersetzt, weswegen Zeile 13 zur Ausgabe von "Hallo, Zelt!" führt.

3.3. Die Klasse std::vector

Die dynamische Speicherallokation für Felder haben wir bereits im Zusammenhang mit Zeigern kennengelernt. Noch flexibler und in vielen Anwendungen komfortabler sind Objekte der Klasse std::vector, die wir im Folgenden auch einfach als Vektoren bezeichnen. Analog zu Feldern können sie verwendet werden, um mehrere Variablen gleichen Typs zusammenzufassen und in aufeinanderfolgenden Speicherplätzen abzulegen. Die größere Flexibilität ergibt sich aus der Möglichkeit, die Anzahl der gespeicherten Werte auch nachträglich je nach Bedarf zu erhöhen oder zu verringern. Zur Verwendung von Vektoren muss die Standardbibliothek <vector> mit einer **#include**-Anweisung in das Programm eingebunden werden. Die Deklaration eines Vektors erfolgt dann nach der Syntax

```
std::vector<TYP> bezeichner;
```

wobei TYP den Typ der Werte angibt, die in dem Vektor gespeichert werden sollen. Ein so erstellter Vektor ist zunächst leer, das heißt die Anzahl seiner Elemente ist 0. Sei N eine positive ganzzahlige Variable oder Konstante und wert eine Variable oder Konstante vom Typ TYP. Dann kann man durch

```
std::vector<TYP> bezeichner(N);
```

direkt einen Vektor aus N (zunächst undefinierten) Elementen erzeugen, oder durch

```
std::vector<TYP> bezeichner(N, wert);
```

einen Vektor aus N Elementen erzeugen, deren Werte alle gleich dem Wert wert sind.

Der Zugriff auf die Elemente eines Vektors erfolgt analog dem Zugriff auf die Elemente eines Feldes mit dem Zugriffsoperator []. Beispielsweise wird durch die Anweisung

```
std::vector<int> v(5,1);
```

ein Vektor v aus 5 ganzzahligen Elementen erzeugt, welche zunächst alle gleich 1 sind. Mittels

```
v[0] = 2;
```

kann man dem ersten Element den Wert 2 zuweisen. Möchte man allen Elementen des Vektors gleichzeitig einen bestimmten Wert, beispielsweise den Wert 7, zuweisen, so kann man die durch die Anweisung

```
std::fill( v.begin(), v.end(), 7  );
```

realisieren. Die Methoden begin() und end() liefern dabei sogenannte *Iteratoren*, die dem Anfang und Ende des Vektors im Speicher entsprechen. Iteratoren sind in C++ eine Verallgemeinerung von Zeigern und werden im Rahmen dieser Vorlesung nicht weiter besprochen. Was genau eine Methode ist, werden wir hingegen in einem späteren Kapitel noch genauer erläutern. Vereinfacht können wir zunächst davon ausgehen, dass es sich bei einer Methode

um eine Funktion handelt, die nur auf bestimmte Objekte angewendet werden kann. Im Gegensatz zu klassischen Funktionsaufrufen ist die Syntax eines Methodenaufrufs für ein Objekt mit dem Bezeichner objekt durch

```
objekt.methode( parameter1, parameter2, ... );
```

gegeben, das heißt der Methodenname wird mit einem Punkt an den Bezeichner des Objekts angehängt. Für Objekte vom Typ std::vector stehen unter anderem die folgenden Methoden zur Verfügung:

Methode	Effekt
resize(N)	Ändern der Größe auf N
size()	Gibt aktuelle Größe zurück
back()	Zugriff auf letztes Element
push_back(wert)	Anhängen von wert ans Ende des Vektors
pop_back()	Entfernen des letzten Elements

Tabelle 3.2. Methoden der Klasse std::vector. Hierbei ist N eine nichtnegative ganzzahlige Variable oder Konstante und der Typ von wert sollte dem Typ der Elemente des Vektors entsprechen.

Des Weiteren kann man Elemente an einer beliebigen Stelle eines Vektors einfügen und alle darauffolgenden Elemente im Speicher um eins nach hinten verschieben. Dies geschieht durch Anwenden der Methode insert() unter Verwendung des begin-Iterators, welcher um den Index erhöht wird, den das eingefügte Element erhalten soll, also beispielsweise nach der Deklaration

```
std::vector<float> vek(5, 1.0);
```

eines Vektors vek der Größe 5 durch

```
vek.insert(vek.begin()+3, 2.7); // fuegt 2.7 an vierter Stelle ein
```

wonach der Vektor die Größe 6 hat. Das Einfügen von Elementen an beliebigen Stellen eines Vektors sollte jedoch vermieden werden, da es für den Rechner einen hohen Aufwand bedeutet: Hat man etwa einen Vektor der Größe 100000 und möchte ein Element an der dritten Stelle einfügen, so müssen dafür 99998 abgespeicherte Werte im Speicher verschoben werden.

Zur Veranschaulichung des Umgangs mit den Methoden aus Tabelle 3.2 betrachten wir wieder ein kurzes Beispielprogramm:

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6    int N = 3;
7    std::vector<double> u( N );        // [??? ??? ???]
8    std::fill(u.begin(), u.end(), 1.); // [1.0 1.0 1.0]
```

```
 9    u.pop_back();                      // [1.0 1.0]
10    u.back() = 2.1;                    // [1.0 2.1]
11    N = 4;
12    u.resize( N );                     // [1.0 2.1 ??? ???]
13    u[2] = 5.0; u[3] = 7.0;            // [1.0 2.1 5.0 7.0]
14    u.push_back( 9.8 );                // [1.0 2.1 5.0 7.0, 9.8]
15    u.insert( u.begin()+2, 3.3 );      // [1.0 2.1 3.3 5.0, 7.0, 9.8]
16    for ( int j = 0; j < u.size(); j++ )
17      std::cout << "u[" << j << "] = " << u[j] << std::endl;
18    return 0;
19  }
```

In diesem Beispiel wird in Zeile 7 ein Vektor der Größe 3 erzeugt, welcher in Zeile 8 mit dem Wert 1.0 gefüllt wird. In Zeile 9 wird das letzte Element gelöscht, wonach der Vektor die Größe 2 hat. In Zeile 10 wird dem letzten Element der Wert 2.1 zugewiesen. In Zeile 12 wird die Größe des Vektors auf 4 geändert und in Zeile 13 der Wert des dritten beziehungsweise vierten Elements auf 5.0 beziehungsweise 7.0 gesetzt. In Zeile 14 wird der Wert 9.8 als fünftes Element hinten angehängt. In Zeile 15 wird der Wert 3.3 an dritter Stelle eingefügt. Die `for`-Schleife in Zeile 16 iteriert über die Indizes des Vektors und die Anweisung im Schleifenkörper veranlasst die Ausgabe des gespeicherten Werte.

Hinweis: Da Vektoren im Speicher wie Felder abgelegt werden und die beiden Konzepte in ihrer fundamentalen Verwendung übereinstimmen, ist es sogar möglich, einen Vektor an eine Funktion zu übergeben, die eigentlich einen Zeiger auf den Anfang eines Feldes als Parameter erwartet. Dies verdeutlicht noch einmal die Große Flexibilität von Vektoren. Der etwas höhere Rechenaufwand bei der Verwendung von Vektoren im Vergleich zu dynamisch allozierten Feldern ist in fast allen Fällen vernachlässigbar.

3.4. Dateioperationen

Um das Ergebnis einer umfangreichen Rechnung nicht jedes Mal neu berechnen zu müssen oder auch um Programmzustände nach dem Beenden eines Programms zu erhalten, macht es durchaus Sinn, gewisse Daten über die Laufzeit des Programms hinaus in Dateien beispielsweise auf der Festplatte des Computers zu speichern, sodass diese zu einem späteren Zeitpunkt weiterverarbeitet werden können. Auch Ergebnisdaten aus wissenschaftlichen Experimenten liegen häufig als Computerdateien vor und sollen dann durch ein Programm in geeigneter Weise aufbereitet oder analysiert werden. Wir wollen nun betrachten, wie wir die Ein- und Ausgabe von Daten anstatt in der Konsole durch passende Dateioperationen realisieren können. Die dazu notwendige Funktionalität wird von der Standardbibliothek `<fstream>` bereitgestellt, welche für die im Folgenden erläuterten Dateioperationen mittels

```
#include <fstream>
```

in das Programm eingebunden werden muss.

Ausgabe von Daten in eine Datei. Zur Ausgabe von Daten in eine Datei erzeugt man zunächst ein Objekt vom Typ `std::ofstream` und öffnet anschließend mit der Methode `open()` eine Datei, in welche die Daten geschrieben werden sollen. Die Anweisungen

```
std::ofstream ausgabedatei;
ausgabedatei.open("daten.txt");
```

erzeugen Beispielsweise ein `std::ofstream`-Objekt mit dem Bezeichner `ausgabedatei` und öffnen eine Datei namens `daten.txt` für die Ausgabe, welche von nun an mit dem Objekt `ausgabedatei` verknüpft ist. Falls eine Datei mit dem angegebenen Dateinamen zum Zeitpunkt des Öffnens noch nicht existiert, so wird sie erzeugt. Das tatsächliche Schreiben in die Datei geschieht dann analog zur Ausgabe in der Konsole mithilfe des Einfügeoperators <<, wobei als linker Operand anstatt des Standard-Ausgabestreams das erzeugte `std::ofstream`-Objekt angegeben wird. Nach dem obigen Öffnen der Datei `daten.txt` schreibt etwa die Anweisung

```
ausgabedatei << "Ausgabetext" << std::end;
```

die Zeichenkette `"Ausgabetext"` in diese Datei. Selbstverständlich könnte man anstatt einer Zeichenkette auch einen oder mehrere Zahlenwerte auf diese Art in die Datei schreiben. Falls die Datei zum Zeitpunkt des Öffnens bereits existiert und Daten enthält, so wird sie in diesem Fall einfach überschrieben. Möchte man stattdessen Daten zu einer bereits existierenden Datei hinzufügen, so übergibt man dem `open`-Befehl zusätzlich zum Dateinamen das Schlüsselwort `std::ios::app`. Im obigen Beispiel würde man die Datei unter diesen Umständen also mittels

```
ausgabedatei.open("daten.txt", std::ios::app);
```

öffnen. Auch in diesem Fall wird eine neue Datei erzeugt, falls noch keine mit dem entsprechenden Namen vorhanden ist. Hat man alle Daten in die Datei geschrieben, so sollte diese mit der Methode `close()` wieder geschlossen werden, also beispielsweise durch

```
ausgabedatei.close();
```

Man kann ein Objekt des Typs `std::ofstream` direkt bei seiner Erzeugung mit einer Datei verknüpfen. Der Dateiname wird dazu bei der Erzeugung des Objekts in Klammern angehängt, das heißt die Syntax ähnelt derjenigen bei der Erzeugung eines Vektors mit einer bestimmten Größe. Mit der Methode `is_open()` kann man überprüfen, ob eine Datei erfolgreich geöffnet werden konnte, etwa so, wie dies in dem folgenden Beispielprogramm geschieht:

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6    std::ofstream f("test.dat");
7    if ( f.is_open() ) {
8      f << "Text zur Ausgabe in die Datei" << std::endl;
9      f << "Noch mehr Text" << std::endl;
```

```
10    }
11    else {
12      std::cout << "Datei konnte nicht geoeffnet werden.\n" ;
13    }
14    f.close();
15    return 0;
16  }
```

Hier wird in Zeile 6 ein `fstream`-Objekt mit dem Bezeichner `f` erzeugt und direkt mit einer Datei mit dem Namen `test.dat` verknüpft. Falls eine Datei mit diesem Namen erfolgreich geöffnet werden konnte, werden im **`if`**-Block ab Zeile 7 zwei Textzeilen in diese Datei geschrieben. Anderenfalls wird eine Fehlermeldung ausgegeben.

Einlesen von Daten aus einer Datei. Zum Einlesen von Daten aus einer Datei erzeugt man zunächst ein Objekt das Typs `std::ifstream`, etwa durch die Anweisung

```
std::ifstream eingabedatei;
```

wodurch ein solches Objekt mit dem Bezeichner `eingabedatei` deklariert wird. Analog der Vorgehensweise bei der Ausgabe öffnet man anschließend mithilfe der Methode `open()` die Datei, aus der die Daten gelesen werden sollen und welche fortan mit dem erzeugten `std::ifstream`-Objekt verknüpft ist. Beispielsweise könnten wir das Objekt `eingabedatei` durch den Befehl

```
eingabedatei.open("dateiname.dat")
```

mit einer Datei mit dem Namen `dateiname.dat` verknüpfen. Auch die Objekte vom Typ `std::ifstream` können durch Angabe des Dateinamens in Klammern direkt bei ihrer Erzeugung mit einer Datei verknüpft werden und stellen zur Überprüfung des Erfolgs eine `is_open`-Methode zur Verfügung.

Das Einlesen der Daten aus der Datei geschieht wieder analog der Vorgehensweise beim Einlesen von Daten aus der Konsole mithilfe des Extraktionsoperators >>, wobei das entsprechende `std::ifstream`-Objekt als linker Operand angegeben wird. Für das obige Objekt `eingabedatei` bewirken die Anweisungen

```
int n;
eingabedatei >> n;
```

beispielsweise das Einlesen und Speichern eines **`int`**-Werts aus der Datei `dateiname.dat` in der **`int`**-Variable `n`. Dabei muss man logischerweise darauf achten, dass der Typ der verwendeten Variable zum Typ des eingelesenen Werts passt. Wendet man den Extraktionsoperator mehrfach an, so wird wie bei der Eingabe über die Konsole jeweils der nächste Wert vom Typ des rechten Operanden eingelesen, wobei Zeichenketten jeweils bis zum nächsten *whitespace character* – also beispielsweise Leerzeichen `' '`, Tabulatorzeichen `'\t'` oder Zeilenumbruch `'\n'` – gelesen werden. Dies funktioniert so lange, bis man das Ende der Datei erreicht hat oder sie durch die Methode `close()` wieder schließt, was man in jedem Fall tun sollte, wenn man alle benötigten Daten eingelesen hat. Nach einem erneuten Öffnen der Datei beginnt das Einlesen wieder von vorne.

Häufig weiß man nicht, wieviele Werte in einer Datei abgespeichert sind. Möchte man eine Datei bis zu ihrem Ende einlesen, so kann man dafür den Rückgabewert des Extraktionsoperators >> verwenden. Dieser ist genau dann **false**, wenn das Ende der Datei erreicht ist und kein Wert mehr eingelesen werden kann. Möchte man beispielsweise fortlaufend **int**-Werte aus der mit dem Objekt `eingabedatei` verknüpften Datei einlesen, so hat eine entsprechende **while**-Schleife die Form

```
int n;
while (eingabedatei >> n) {
  ...
}
```

Dabei muss man beachten, dass das tatsächliche Einlesen der Werte bereits im Schleifenkopf geschieht. Der jeweils eingelesene Wert wird direkt in der **int**-Variable `n` gespeichert und kann im Schleifenkörper verarbeitet werden. Diese Schleife wird so lange wiederholt, bis das Ende der Datei erreicht ist.

Wir veranschaulichen das Einlesen von Daten mit einem Beispielprogramm, welches einen Satz von Koordinaten aus einer Datei mit dem Namen `koordinaten.txt` einliest und in einem geeigneten Vektor abspeichert. Die Datei habe dazu folgenden Inhalt:

```
3.4   3.7   2.9
2.9   1.3   0.1
1.0   2.5   0.3
3.4   0.0   2.0
```

Wir interpretieren die Zahlen in jeder Zeile als die Koordinaten eines Punktes im dreidimensionalen Raum und möchten diese in einem Vektor abspeichern, dessen Elemente selbst wieder durch dreielementige Vektoren mit Elementtyp **float** gegeben sind. Dies wird durch das folgende Programm realisiert:

```
 1  #include <iostream>
 2  #include <fstream>
 3  #include <vector>
 4
 5  int main()
 6  {
 7    std::ifstream input( "koordinaten.txt" );
 8    if (!input.is_open()){
 9      std::cout << "Datei konnte nicht geoeffnet werden\n";
10      return 1;
11    }
12    std::vector< std::vector<float> > vek;
13    std::vector< float > koord(3);
14    while (input >> koord[0] >> koord[1] >> koord[2]) {
15      vek.push_back( koord );
16    }
```

```
17    for (int j = 0; j < vek.size(); ++j) {
18      std::cout << vek[j][0] << "  "
19                << vek[j][1] << "  "
20                << vek[j][2] << std::endl;
21    }
22    return 0;
23  }
```

In diesem Programm wird in Zeile 7 ein Objekt des Typs `std::ifstream` mit dem Bezeichner `input` erzeugt und direkt mit der Datei `koordinaten.txt` verknüpft. Falls diese Datei nicht geöffnet werden konnte, sorgt der **`if`**-Block ab Zeile 8 dafür, dass eine Fehlermeldung ausgegeben und das Programm direkt beendet wird. In Zeile 12 wird ein zunächst leerer Vektor `vek` mit Elementen des Typs `std::vector<float>` erzeugt, in welchem wir später die Koordinaten speichern. Die Anweisung in Zeile 13 erzeugt einen Vektor `koord` mit drei Einträgen vom Typ float, welcher uns während des Einlesens der Werte als Zwischenspeicher dient. Die **`while`**-Schleife ab Zeile 14 wird nun so lange wiederholt, bis das Ende der Datei `koordinaten.txt` erreicht ist. Die Anweisung im Schleifenkopf sorgt gleichzeitig dafür, dass eine Zeile aus dieser Datei eingelesen wird und die entsprechenden drei Werte im Vektor `koord` zwischengespeichert werden. Die `push_back()`-Anweisung in Zeile 15 hängt in jedem Schleifendurchlauf die zwischengespeicherten Werte an das Ende des Vektors `vek` an. Die **`for`**-Schleife ab Zeile 17 iteriert nun über die Indizes des Vektors `vek` und sorgt für die Ausgabe der eingelesenen Koordinaten.

Hinweis: Natürlich könnten wir die Werte im Beispielprogramm auch in ein Feld, beispielsweise `float vek[4][3]`, einlesen. Dieses Vorgehen bietet sich insbesondere dann an, wenn man die Menge der einzulesenden Daten im Voraus kennt. Anderenfalls entstehen leicht Speicherzugriffsfehler.

3.5. Parameterübergabe an die `main`-Funktion

Zum Abschluss dieses Kapitels erläutern wir noch die Parameterübergabe an die `main`-Funktion eines Programms. Darunter verstehen wir die Möglichkeit, einen oder mehrere Werte, sogenannte *Argumente*, direkt beim Aufruf eines Programms in der Konsole mit anzugeben, sodass diese dann in der `main`-Funktion zur Verfügung stehen. Diese Vorgehensweise stellt häufig eine sinnvolle Alternative zur Werteingabe per Benutzerabfrage dar und bietet sich insbesondere dann an, wenn die Programmaufrufe später automatisiert erfolgen sollen, etwa durch ein *Shell-Skript* (dabei handelt es sich um ein mehr oder weniger umfangreiches Program mit einer eigenen Syntax, das im Grunde genommen das Ausführen von Konsolenbefehlen erledigt).

Die Funktion `main()` hängt immer von zwei Parametern ab, und die Bezeichner dieser Parameter lauten in jedem Fall `argc` und `argv`, wobei der erste Parameter `argc` vom Typ **`int`** und der zweite Parameter `argv` vom Typ **`char`** `**` ist. Möchte man mit diesen Parametern arbeiten, so müssen sie bei der Definition der `main`-Funktion mit angegeben werden, das heißt die Definition wird in diesem Fall durch

```
int main( int argc, char **argv )
```

oder

```
int main( int argc, char *argv[] )
```

eingeleitet, wobei beide Möglichkeiten äquivalent sind (wir erinnern uns an die Äquivalenz von Zeigern und Feldern). Die Bedeutung der beiden Parameter ist wie folgt:

argc: Anzahl der Argumente beim Aufruf (argument count),
argv: Array von **char**-Arrays, welches die Argumente enthält (argument value).

Der Befehl zum Aufruf des Programms selbst zählt dabei stets als erstes Argument. Dies bedeutet, dass ohne zusätzliche Argumente der Wert von argc gleich 1 ist und das Feld argv[0] den Programmaufruf enthält.

Betrachten wir dazu das folgende Beispielprogramm:

```
 1 #include<iostream>
 2
 3 int main( int argc, char *argv[] )
 4 {
 5   std::cout << "Argumentanzahl: " << argc << std::endl;
 6   for (int i=0; i<argc; ++i) {
 7     std::cout << argv[i] << std::endl;
 8   }
 9   return 0;
10 }
```

Die Anweisung in Zeile 5 gibt die Anzahl der Argument aus. Anschließend iteriert die **for**-Schleife ab Zeile 6 über diese Anzahl und gibt gerade die Argumente aus dem Aufruf in der Konsole aus. Wenn wir dieses Programm in eine ausführbare Datei mit dem Namen main-param übersetzen und anschließend in der Konsole durch die Eingabe von

```
./main-param
```

ausführen, so erhalten wir auf dem Bildschirm die Ausgabe:

```
Argumentanzahl: 1
./main-param
```

Rufen wir das Programm hingegen mit mehreren Argumenten auf, beispielsweise durch die Eingabe von

```
./main-param Hello World! 2 7.4
```

in der Konsole, so erhalten wir die Ausgabe:

```
Argumentanzahl: 5
./main-param
Hello
World!
2
7.4
```

Möchte man mehrere Argumente zusammenfassen, so kann man dazu Anführungszeichen verwenden, etwa führt der Aufruf

```
./main-param "Hello World!"
```

zur Ausgabe

```
Argumentanzahl: 2
./main-param
Hello World!
```

Wenn man mit als Argumenten eingegebenen Zahlenwerten arbeiten möchte, so müssen diese zuerst in den richtigen Typ konvertiert werden, da sie im Parameter `argv` lediglich als **`char`**-Feld vorliegen. Dafür kann man die Funktionen `std::atoi` (Rückgabetyp **`int`**), `std::atol` (Rückgabetyp **`long`**) oder `std::atof` (Rückgabetyp **`double`**) aus der Standardbibiliothek `<cstdlib>` verwenden. Beispielsweise könnte man das erste auf den Programmaufruf folgende Argument mittels

```
int N = std::atoi( argv[1] );
```

in einer **`int`**-Variable speichern. Wie bei allen anderen Formen des Einlesens von Werten muss auch hierbei stets auf die Verwendung adäquater Variablentypen und passender Konvertierungsfunktionen geachtet werden.

KAPITEL 4

Algorithmik

Unter *Algorithmik* versteht man die Entwicklung und Analyse von Algorithmen, also von Computer-realisierbaren Verfahren zur systematischen Lösung einer Aufgabe. Typische Aufgaben sind das Sortieren von Listen, die Bestimmung der Lösung eines mathematischen Problems oder die Berechnung der Wahrscheinlichkeit eines Ereignisses auf Basis gegebener Daten. Bei der Analyse eines Algorithmus sind die folgenden Kriterien relevant.

(i) *Durchführbarkeit:* Es muss sicher gestellt werden, dass alle Schritte wohldefiniert sind, sodass zum Beispiel keine Divisionen durch Null auftreten. Außerdem muss zu jedem Zeitpunkt des Verfahrens klar sein, welcher Schritt als nächstes auszuführen ist (*Determinismus*).
(ii) *Terminierung:* Es muss gewährleistet sein, dass das Verfahren regulär stoppt und nicht beispielsweise in eine Endlosschleife gerät.
(iii) *Korrektheit:* Es muss nachgewiesen werden, dass für alle zulässigen Eingaben das richtige Ergebnis berechnet wird. Außerdem sollten identische Eingaben zum gleichen Ergebnis führen (*Determiniertheit*).
(iv) *Effizienz:* Es muss untersucht werden, ob der Algorithmus für realistische Problemgrößen in akzeptabler Zeit terminiert.

Häufig wird außerdem gefordert, dass die Anweisungen in einem endlichen Text darstellbar sind (*Finitheit*) und dass in jedem Schritt des Algorithmus nur endlich viel Speicherplatz benötigt wird (*dynamische Finitheit*). Einige dieser Aspekte erscheinen auf den ersten Blick trivial, ihre Relevanz zeigt sich jedoch, wenn man unterschiedliche Beispiele von Algorithmen betrachtet.

Wenn wir von einem Algorithmus sprechen, so meinen wir immer das Verfahren an sich, unabhängig von einer konkreten Realisierung in einer Programmiersprache. Mit der Implementierung eines Verfahrens sollte man idealerweise erst dann beginnen, wenn man sein Verhalten unter gegebenen Voraussetzungen bereits verstanden verstanden hat. Bei der Formulierung eines Algorithmus lohnt es sich jedoch, die spätere Umsetzung zu bedenken. Das effizienteste Verfahren ist praktisch nutzlos, wenn man nicht in der Lage ist, es adäquat zu implementieren.

Die ersten Algorithmen wurden bereits im antiken Griechenland formuliert. Als Beispiel sei hier etwa das *Sieb des Eratosthenes* zur Identifikation von Primzahlen genannt. Der erste für die Umsetzung auf einem Computer gedachte Algorithmus wurde im Jahr 1843 von der britischen Mathematikerin Ada Lovelace zur Berechnung der Bernoullizahlen formuliert. Sie wird daher häufig als die erste Programmiererin angesehen. Das Wort »Computer« ist in diesem Zusammenhang allerdings noch etwas übertrieben. Tatsächlich war ihr Algorithmus

für die vom Mathematiker und Philosophen Charles Babbage im Jahr 1837 beschriebene *Analytical Engine* gedacht. Dabei handelt es sich um den Entwurf einer Rechenmaschine, bestehend aus rund 55000 Teilen, welche von einer Dampfmaschine angetrieben werden sollte. Sie wurde jedoch nur in Teilen fertiggestellt. Zur Einordnung: Die praktische Nutzung der Elektrizität durch Menschen begann mit der Inbetriebnahme des Schreibtelegraphen von Samuel Morse ebenfalls im Jahr 1837. Der erste funktionsfähige Digitalrechner der Welt war der *Z3* und wurde 1941 von Konrad Zuse gebaut. Erst im Jahr 1960 erreichten moderne Computer eine Rechengenauigkeit, wie sie für die Analytical Engine vorgesehen war.

4.1. Algorithmusarten

Die am häufigsten autretenden Algorithmen sind iterative und rekursive Algorithmen, welche wir im Folgenden erläutern werden. Je nach Sichtweise können diese weiter unterteilt werden (exaktes Verfahren, Approximationsalgorithmus, ...). Außerdem existieren noch weitere Algorithmusarten, etwa randomisierte Algorithmen, welche eine Zufallskomponente enthalten.

Iterative Algorithmen. Bei *iterativen Algorithmen* wird eine Rechenvorschrift, die auch Fallunterscheidungen enthalten darf, wiederholt auf einen Datensatz angewendet, bis ein geeignetes Abbruchkriterium erfüllt ist. Abstrakt lässt sich dies mit einer Abbildung $T : X \to X$ beschreiben, die für einen Startwert $x^0 \in X$ die *Iterierten* $(x^k)_{k=1,2,\dots}$ durch die *Iterationsvorschrift*

$$x^k = T(x^{k-1})$$

definiert, bis eine vorgegebene Anzahl an Iterationen erreicht ist oder ein bestimmtes Abbruchkriterium erfüllt ist. Ein übliches Abbruchkriterium ist etwa, dass sich die Iterierten nicht mehr stark ändern, also bis $\|x^{k+1} - x^k\| \leq \delta$ für eine kleine Zahl $\delta > 0$ gilt. Die Terminierung und Korrektheit lässt sich in einigen Situationen mit dem Banachschen Fixpunktsatz nachweisen. Ein Beispiel eines iterativen Verfahrens ist die Berechnung der Quadratwurzel nach Heron. Man beachte, dass hier eine Aufgabe häufig nur approximativ gelöst wird.

Rekursive Algorithmen. *Rekursive Algorithmen* basieren auf der Beobachtung, dass sich ein Problem einer bestimmten Größe häufig auf ähnliche Probleme kleinerer Größe zurückführen und für eine gewisse minimale Problemgröße direkt lösen lässt. Ein Beispiel ist die Fakultätsfunktion $f(n) = n!$, die die *Rekursionsformel*

$$f(j) = jf(j-1)$$

für $j \geq 1$ und die *Rekursionsverankerung*

$$f(0) = 1$$

erfüllt. In diesem Fall ist die Problemgröße das Argument n und wir führen die Berechnung von $n!$ auf die Berechnung von $(n-1)!$ zurück. Rekursionen zeichnen sich dadurch aus, dass Funktionen oder Routinen sich selbst aufrufen und entstehen auf natürliche Weise bei induktiv definierten Objekten. Es gibt aber Fälle, die in keiner Form dem intuitiven menschlichen Handeln entsprechen.

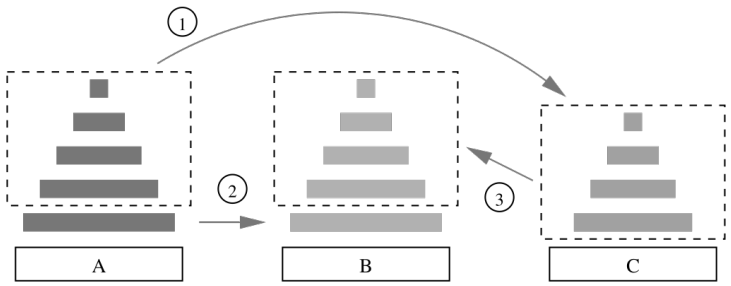

Abbildung 4.1. Lösung des Problems der Türme von Hanoi durch Reduktion auf kleinere Probleme derselben Art.

Beispiel 4.1 (Türme von Hanoi)**.** Beim Problem der Türme von Hanoi soll ein der Größe nach sortierter Stapel mit n Scheiben unterschiedlicher Größe von einer Standposition A auf eine Zielposition B unter Verwendung einer Hilfsposition C versetzt werden. Dabei dürfen nur einzelne Scheiben versetzt werden und die Stapel müssen stets der Größe nach geordnet sein, das heißt eine Scheibe darf immer nur auf einer größeren abgelegt werden. Im Fall von drei Scheiben ist das Problem noch intuitiv lösbar, aber schon bei fünf Scheiben müssen viele Bewegungen durchgeführt werden. Angenommen, wir wissen, wie man einen Stapel mit $(n-1)$ Scheiben von einer Position auf eine andere unter Verwendung der dritten als Hilfsposition versetzen kann. Dann können wir den Stapel mit n Scheiben als Zusammensetzung eines Stapels mit $(n-1)$ Scheiben und der untersten, größten Scheibe betrachten und das Problem rekursiv in drei Schritten lösen:

(1) Versetze den oberen $(n-1)$-Teilstapel von A nach C.
(2) Versetze den unteren 1-Teilstapel von A nach B.
(3) Versetze den $(n-1)$-Stapel von C nach B.

Das Vorgehen ist in Abbildung 4.1 illustriert. In diesen Schritten ist nur die Versetzung kleinerer Stapel erforderlich. Während die Versetzung eines Stapels mit einer Scheibe trivial ist, lässt sich die Versetzung von Stapeln mit $(n-1)$ Scheiben wieder jeweils auf zwei Versetzungen von Stapeln mit $(n-2)$ Scheiben und eine eines Stapels mit einer Scheibe zurückführen.

Im Beispiel 4.1 wird eine sehr komplexe Aufgabenstellung mit einer einfachen Rekursionsvorschrift gelöst. Etwas problematisch ist dabei, dass der Aufwand in jedem Reduktionsschritt um einen Faktor 2 vergrößert wird, was zu mehr als 2^n Operationen führt. Vor diesem Hintergrund ist es häufig sinnvoll, eine Rekursion zu vermeiden und durch einen dynamischen oder iterativen Algorithmus zu ersetzen, was beispielsweise bei der Berechnung der Fakultät einfach möglich ist, nicht jedoch im Fall der Türme von Hanoi.

Dynamische Algorithmen. Bei *dynamischen Algorithmen* unterscheiden wir zwischen dynamischen iterativen und dynamischen rekursiven Algorithmen. Sie stellen also eine Sonderform der beiden bereits vorgestellten Arten von Algorithmen dar. Auch bei einem dynamischen Algorithmus wird eine Folge von Anweisungen sukzessive abgearbeitet. Entscheidend ist, dass alle zur Berechnung benötigten Wert dynamisch gespeichert werden und in keinem

Fall ein Wert mehrfach berechnet werden muss. Abstrakt entspricht ein dynamischer iterativer Algorithmus einer Folge von Abbildungen $T^k : X^k \to X$, die für einen Startwert $x^0 \in X$ die Iterierten $(x^k)_{k=1,2,\ldots}$ durch die Iterationsvorschrift

$$x^k = T^k(x^{k-1}, x^{k-2}, \ldots, x^0)$$

definiert und dies solange fortsetzt, bis eine vorgegebene Anzahl an Schritten erreicht oder ein bestimmtes Abbruchkriterium erfüllt ist. Um den Unterschied zwischen einem einfachen rekursiven und einem dynamischen rekursiven Algorithmus zu verdeutlichen, betrachten wir als Beispiel die Berechnung der Fibonacci-Zahlen.

Die sogenannte Fibonacci-Folge ist eine Folge natürlicher Zahlen, welche nach dem Mathematiker Leonardo Fibonacci benannt ist, der damit im Jahr 1202 das Wachstum einer isolierten Kaninchenpopulation beschrieb. Er ging davon aus, dass jedes Kaninchenpaar pro Monat ein neues Paar Kaninchen wirft. Außerdem nahm er an, dass neugeborene Paare erst in ihrem zweiten Lebensmonat selbst Nachwuchs bekommen. Beginnend mit einem einzigen neugeborenen Paar, bilden die Anzahlen der Kaninchenpaare in jedem Monat die Fibonacci-Zahlen:

$$1, 1, 2, 3, 5, 8, 13, 21, 34 \ldots$$

Die zwei ersten Glieder f_1, f_2 der Fibonacci-Folge sind jeweils 1. Alle weiteren Folgenglieder f_n ergeben sich als Summe ihrer beiden Vorgänger, das heißt es gilt die Regel

$$f_n = f_{n-1} + f_{n-2}.$$

Algorithmus 4.2 (Fibonacci-Zahl, dynamisch iterativ)**.** *Eingabe:* Natürliche Zahl n.

(1) Speichere $f_1 = 1$ und $f_2 = 1$ und setze $k = 2$.
(2) Falls $n \leq k$, so stoppe.
(3) Erhöhe k um 1, d. h. setze $k = k + 1$.
(4) Berechne und speichere $f_k = f_{k-1} + f_{k-2}$.
(5) Gehe zu Schritt (2).

Ausgabe: Fibonacci-Zahl f_n.

Der iterative Algorithmus ist zwar effizient, jedoch in gewisser Weise komplizierter als das Ursprüngliche Problem. Ein rekursiver Algorithmus kommt der natürlichen Formulierung des Problems deutlich näher.

Algorithmus 4.3 (Fibonacci-Zahl, naiv rekursiv)**.** *Eingabe:* Natürliche Zahl n.

(1) Falls $n = 1$ oder $n = 2$, so setze $f_n = 1$ und stoppe.
(2) Setze $f_n = f_{n-1} + f_{n-2}$.

Ausgabe: Fibonacci-Zahl f_n.

Für eine dynamische Variante dieses Algorithmus gehen wir davon aus, dass wir einen unabhängigen Speicher, etwa ein genügend großes Feld, für die Speicherung der berechneten Werte $(f_n)_{n=1,2,\ldots}$ zur Verfügung haben. Zu Beginn seien die Einträge f_1 und f_2 mit jeweils mit 1 sowie alle weiteren Einträge des Feldes mit 0 initialisiert.

Algorithmus 4.4 (Fibonacci-Zahl, dynamisch rekursiv)**.** *Eingabe:* Natürliche Zahl n.

(1) Falls $f_n > 0$, so stoppe.
(2) Berechne und speichere $f_n = f_{n-1} + f_{n-2}$.

Ausgabe: Fibonacci-Zahl f_n.

Während bei der naiv-rekursiven Variante gleiche Werte unter Umständen sehr oft berechnet werden müssen, ist dies bei der dynamisch-rekursiven Variante nicht der Fall. Der Unterschied wird besonders deutlich, wenn man tatsächliche Implementierungen in Bezug auf ihre Laufzeit vergleicht.

4.2. Komplexität

Wir wollen die Zeit, die ein Algorithmus zur Lösung eines Problems benötigt, unabhängig von der Art und Weise seiner praktischen Umsetzung quantifizieren. Dazu untersuchen wir, wie sich die Anzahl der benötigten Rechenschritte, wie arithmetischen Operationen oder Vergleichen, ändert, wenn wir die Problemgröße verändern.

Definition 4.5. (i) Die *(Problem-) Größe* einer Aufgabe ist eine charakteristische Größe des Problems, wie etwa die Anzahl der zu bearbeitenden Daten, die Anzahl von Summanden oder der Index eines zu berechnenden Folgeglieds.
(ii) Der *Aufwand* eines Algorithmus zur Lösung einer Aufgabe der Größe n ist die Anzahl $\mathcal{A}(n)$ der dazu erforderlichen arithmetischen Operationen und Vergleiche.

Wir sind meist an einer oberen Schranke für den maximalen Aufwand interessiert, das heißt dem Aufwand in einem *Worst-Case-Scenario*.

Beispiele 4.6. (i) Das Suchen eines Elements in einer Liste mit n Elementen erfordert maximal n Vergleiche, das heißt $\mathcal{A}(n) \leq n$.
(ii) Sind k arithmetische Operationen zur Bestimmung der Summanden s_j in der Summe $S_n = s_1 + s_2 + ... + s_n$ erforderlich, so erhalten wir den Gesamtaufwand $\mathcal{A}(n) \leq (k+1)n$. Im Fall der Gauß-Summe mit $s_j = j$ lässt sich der Aufwand mit der Formel $S_n = n(n+1)/2$ auf drei Operationen reduzieren, unabhängig von n.
(iii) Die rekursive Berechnung des Folgeglieds $f_n = f_{n-1} + f_{n-2}$ mit Initialisierung $f_1 = 1$ und $f_2 = 1$ der Fibonacci-Folge erfordert einen Aufwand $\mathcal{A}(n) \geq \phi^n$, wobei $\phi \approx 1.618$ die goldene Schnittzahl ist. Eine iterative Realisierung hingegen nur $\mathcal{A}(n) \leq n$.

Um die Praktikabilität eines Algorithmus einordnen zu können, führen wir gewisse Aufwandsklassen ein, die prägnant angeben, wie sich der Aufwand vergrößert, wenn die Problemgröße beispielsweise verdoppelt wird.

Definition 4.7. (i) Der Aufwand $\mathcal{A}(n)$ ist *polynomiell*, wenn Zahlen $p, c \geq 0$ existieren, sodass für alle $n \geq 1$ gilt

$$\mathcal{A}(n) \leq cn^p.$$

Im Fall $p = 0, 1, 2, 3$ nennen wir den Aufwand *konstant*, *linear*, *quadratisch* beziehungsweise *kubisch*.
(ii) Der Aufwand $\mathcal{A}(n)$ ist *logarithmisch*, wenn er nicht linear ist und Zahlen $c > 0, s \geq 1$ existieren, sodass für alle $n \geq 1$ gilt

$$\mathcal{A}(n) \leq c \log_2(sn)$$

(iii) Der Aufwand $\mathcal{A}(n)$ ist *exponentiell*, wenn Zahlen $c > 0, s > 1$ existieren, sodass für all $n \geq 1$ gilt

$$\mathcal{A}(n) \geq cs^n.$$

Mit diesen Begriffen lassen sich verschiedene Algorithmen bezüglich der erforderlichen Rechenschritte übersichtlich einordnen.

Beispiele 4.8. (i) Das dynamische Suchen eines Elements in einer Liste erfordert linearen Aufwand. Eine binäre Suche in einer sortierten Liste erfordert logarithmischen Aufwand.
(ii) Die Bestimmung der Gauß-Summe erfordert linearen Aufwand bei iterativer Realisierung und konstanten Aufwand bei Verwendung der Summenformel.
(iii) Die Berechnung eines Glieds der Fibonacci-Folge besitzt exponentiellen Aufwand bei rekursiver Realisierung und linearen Aufwand bei iterativer Realisierung.
(iv) Das Erraten eines Passworts durch Ausprobieren aller Möglichkeiten (*Brute Force*) erfordert exponentiellen Aufwand bezüglich der Passwortlänge.

In der Regel erfordert die Lösung einer Aufgabe der Größe n mindestens n Operationen, da meist jede der n Informationen in die Berechnungen eingeht. Ein Algorithmus mit linearem Aufwand ist daher meist optimal. Die Aufgabe kann in diesem Fall für eine große Spanne von Problemgrößen *effizient* vom Computer gelöst werden. Bei exponentiellem Aufwand, beispielsweise $\mathcal{A}(n) = 2^n$, stößt man sehr schnell an Grenzen des Realisierbaren, denn schon für Problemgrößen $n \geq 100$ ergeben sich

$$\mathcal{A}(n) \geq 1.2 \times 10^{30}$$

Operationen. Auf einem 2-GHz-Rechner, der 2×10^9 Operationen pro Sekunde durchführen kann, würde eine solche Berechnung mehr als 2×10^{13} Jahre dauern. Selbst wenn man eintausend dieser Rechner parallel einsetzen könnte, würde sich eine Rechenzeit von mehr als 2×10^{10} Jahren ergeben. Dies ist eine durchaus lange Zeit, wenn man bedenket, dass das Alter des Universums nach heutigem Kenntnisstand etwas weniger als $1.4 * 10^{10}$ Jahre beträgt. Wir erkennen also, dass exponentieller Aufwand eines Verfahrens inakzeptabel oder zumindest alles andere als wünschenswert ist. Selbst kubischer Aufwand führt schnell zu Laufzeitproblemen. In vielen Anwendungen ist ist quadratischer Aufwand noch vertretbar. Besser ist es jedoch, wenn man wenigstens ein Verfahren mit superlinearem Aufwand, der zwischen linearem und quadratischem Aufwand liegt, finden kann.

Hinweis: Neben der Wahl eines Algorithmus beeinflusst natürlich auch die konkrete Implementierung die Laufzeit eines Programms. Beispielsweise ist der Aufwand des Zugriffs auf die Einträge einer Liste abhängig von deren technischer Realisierung.

Probleme, die auf Algorithmen mit exponentiellem Aufwand führen, gehören häufig zur Klasse der *NP-vollständigen Probleme*. Dies ist eine Reihe von Problemen, welche insofern als gleichwertig anzusehen sind, als dass zu ihrer Lösung bis heute keine Verfahren mit polynomiellem Aufwand bekannt sind, die Lösung eines dieser Probleme jedoch gleichwertig zur Lösung aller NP-vollständigen Probleme wäre. Dazu gehört beispielsweise das *Problem des Handlungsreisenden*, der eine kürzeste Rundreise durch mehrere Städte sucht. Es wird vermutet, dass für diese Aufgaben keine Lösungsverfahren mit polynomiellem Aufwand existieren. Ein Beweis dieser Nichtexistenz konnte bis heute jedoch ebenso wenig erbracht werden. Die Frage, ob ein NP-vollständiges Problem mit polynomiellem Aufwand gelöst werden kann oder ob für

keine dieser Problemstellungen ein solches Verfahren existiert, wird auch als *P-NP-Problem* bezeichnet. Es handelt sich dabei um eines der wichtigsten ungelösten Probleme der modernen Mathematik beziehungsweise theoretischen Informatik. Konsequenterweise wurde es in die Liste der sieben Millennium-Probleme aufgenommen, welche im Jahr 2000 vom Clay Mathematics Institute in Cambridge veröffentlicht wurde. Für die Lösung des P-NP-Problems ist – ebenso wie für die Lösung der anderen Millennium-Probleme – ein Preisgeld von einer Million US-Dollar ausgelobt.

4.3. Sortierverfahren

Das Sortieren von Listen von Datensätzen ist eine zentrale Aufgabe der Datenverarbeitung. Um die wichtigsten Ansätze darzustellen, betrachten wir eine Liste $L = [a_1, a_2, \ldots, a_n]$ mit n ganzen Zahlen, die aufsteigend sortiert werden soll.

Bubblesort. Das *Bubblesort*-Verfahren ähnelt dem Sortieren von Büchern in einem Regal, die der Reihe nach geprüft und gegebenenfalls nach vorne versetzt werden.

Algorithmus 4.9 (Bubblesort)**.** *Eingabe:* Liste $L = [a_1, a_2, \ldots, a_n]$.

(1) Setze $i = 2$.
(2) Setzte $k = 0$.
(3) Falls $a_{i-k} < a_{i-k-1}$, so vertausche die beiden Einträge, andernfalls gehe zu (5).
(4) Falls $k < i - 2$, so erhöhe k um 1 und gehe zu (3).
(5) Falls $i < n$, so erhöhe i um 1 und gehe zu (2).

Ausgabe: Sortierte Liste $L = [a_1, a_2, \ldots, a_n]$.

Im i-ten Schritt dieses iterativen Algorithmus fällt zunächst ein Vergleich, sowie anschließend bis zu $i - 2$ weitere an, sofern Einträge vertauscht werden müssen. Im schlechtesten Fall erhalten wir also

$$\mathcal{A}(n) \leq \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \leq \frac{1}{2}n^2,$$

das heißt, dass das Verfahren quadratischen Aufwand besitzt. Dies ist für Listen mit wenigen ($n \leq 10000$) Einträgen akzeptabel, insbesondere da dieses Sortierverfahren sehr leicht zu implementieren ist.

Mergesort. Das *Mergesort*-Verfahren basiert auf der rekursiven Verkleinerung der zu sortierenden Liste. Nach Auffüllen der Liste mit Nullen oder sehr großen Einträgen, können wir ohne Probleme annehmen, dass $n = 2^k$ mit einer ganzen Zahl k gilt, sodass wir die Liste wiederholt halbieren können, bis wir zu einelementigen Teillisten kommen. Zur Motivation des Verfahrens nehmen wir an, dass die linke und rechte Teilliste $L^{\ell}_{n/2}$ sowie $L^{r}_{n/2}$ einer n-elementigen Liste L bereits sortiert sind. Die sortierte Liste L_n erhalten wir dann durch ein einfaches Zusammenführen, den sogenannten *merge*-Schritt, der beiden Teillisten. Dies führt auf den folgenden rekursiven Algorithmus, der für eine Liste L eine sortierte Liste $\widetilde{L}$ zurückgibt.

Algorithmus 4.10 (Mergesort). *Eingabe:* Liste L der Länge $n = 2^k$, wobei $k \geq 0$. Aufruf des Algorithmus durch $\mathrm{msort}(L, k)$.

(1) Gilt $k = 0$, so ist die Liste bereits sortiert, das heißt setze $\widetilde{L} = L$.
(2) Gilt $k > 0$, so setze

$$\widetilde{L} = \mathrm{merge}\left(\mathrm{msort}(L^{\ell}_{n/2}, k-1), \mathrm{msort}(L^{r}_{n/2}, k-1)\right).$$

Ausgabe: Sortierte Liste $\widetilde{L}$.

Die Realisierung der Zusammenführung zweier Listen der Länge m kann mit einem iterativen Algorithmus mit linearem Aufwand $\mathcal{A}_{\mathrm{merge}}(m) \leq 2m$ realisiert werden. Der Aufruf von msort mit einer Liste der Länge n führt zu einem Aufruf von merge mit zwei Listen halber Länge und zwei Aufrufen von msort mit Listen halber Länge. Insgesamt erhalten wir den Aufwand

$$\mathcal{A}(n) \leq n + 2\mathcal{A}(n/2).$$

Diese Argumentation wenden wir wiederholt an und erhalten

$$\begin{aligned}\mathcal{A}(n) &\leq n + 2(n/2 + 2\mathcal{A}(n/4)) = 2n + 4\mathcal{A}(n/4)\\ &\leq 2n + 4(n/4 + 2\mathcal{A}(n/8)) = 3n + 8\mathcal{A}(n/8)\\ &\leq \cdots \leq kn + 2^k\mathcal{A}(n/2^k) = kn + 2^k\mathcal{A}(1) = kn,\end{aligned}$$

wobei wir verwendet haben, dass das Sortieren einelementiger Listen keinen Aufwand erfordert, also $\mathcal{A}(1) = 0$ gilt. Da $k = \log_2(n)$ ist, folgt

$$\mathcal{A}(n) \leq \log_2(n)n.$$

Dies ist ein Beispiel eines superlinearen Aufwands, der sich allerdings kaum von linearem Aufwand unterscheidet, da zum Beispiel $\log_2(10^6) \leq 20$. Problematisch ist bei rekursiven Algorithmen oft eine effiziente Speicherverwaltung.

Andere Sortierverfahren. Real auftretende Datensätze besitzen in der Regel zusätzliche Strukturen hinsichtlich ihrer Verteilung, beispielsweise sind die Geburtsdaten einer Personengruppe meist gleichmäßig über einen gewissen Zeitraum verteilt oder die Anfangsbuchstaben von Nachnamen treten mit bekannten Häufigkeiten auf. In diesem Fall kann man den Datenbereich einfach unterteilen, die Einträge den Teilbereichen zuordnen und sich auf das Sortieren der in der Regel erheblich kleineren Teillisten beschränken. Auf solchen Beobachtungen basieren die *Bucketsort*- und *Quicksort*-Verfahren. Unter geeigneten Bedingungen an die Einträge der Liste führen sie zu sehr effizienten Verfahren mit geringem Speicherbedarf, können aber im schlechtesten Fall auch zu quadratischem Aufwand führen.

4.4. Künstliche Intelligenz

In klassischen Algorithmen werden vom Nutzer erdachte Lösungsstrategien realisiert, die meist auf einem sehr guten Verständnis der zugrundeliegenden Aufgabe basieren. Algorithmen der *künstlichen Intelligenz*, kurz KI, versuchen, auf für Menschen schwer überschaubare oder nur mit sehr hohem Aufwand lösbare Probleme, Antworten mit vertretbarem Aufwand zu erzeugen. Wir folgen in diesem Abschnitt der Darstellung des Buchs *Algorithmen kompakt*

und verständlich von M. von Rimscha (Springer, 2008) und verweisen auf dieses Buch für weitere Details.

Maschinelles Lernen. Beim *maschinellen Lernen* wird ein menschliches Entscheidungsverhalten unter Berücksichtigung verschiedener Kriterien beobachtet und anschließend ausgewertet. Mit der statistischen Bewertung der Kriterien lässt sich dann ein Entscheidungsbaum konstruieren, der das Entscheidungsverhalten strukturiert wiedergibt. Ein populäres Beispiel ist die Freizeitgestaltung einer Person, die in Abhängigkeit von den Kriterien Wetter (Sonne/bedeckt/Regen) und Wochenende (ja/nein) sowie Verfügbarkeit interessanter Kinofilme (ja/nein) und Freunden (haben Zeit/keine Zeit) eine Entscheidung zwischen den Aktivitäten Radtour, Kino, Café, Schwimmen und zu Hause bleiben trifft. Die beobachteten Daten führen auf Tabelle 4.1, die jedoch nur einen Auszug eines größeren Datensatzes darstellt.

Freunde	Wetter	Kinofilm	Wochenende	Entscheidung
Ja	Sonne	Ja	Ja	Radtour
Ja	Regen	Nein	Nein	zu Hause
Nein	Sonne	Nein	Nein	Schwimmen
Nein	Bedeckt	Nein	Nein	Radtour
⋮	⋮	⋮	⋮	⋮

Tabelle 4.1. Entscheidungsverhalten bei Berücksichtigung verschiedener Kriterien.

Abhängig von der Häufigkeit des Auftretens eines Kriteriums wird sein Informationsgehalt quantifiziert und damit die Priorität im Entscheidungsbaum, der in Abbildung 4.2 dargestellt ist, festgelegt.

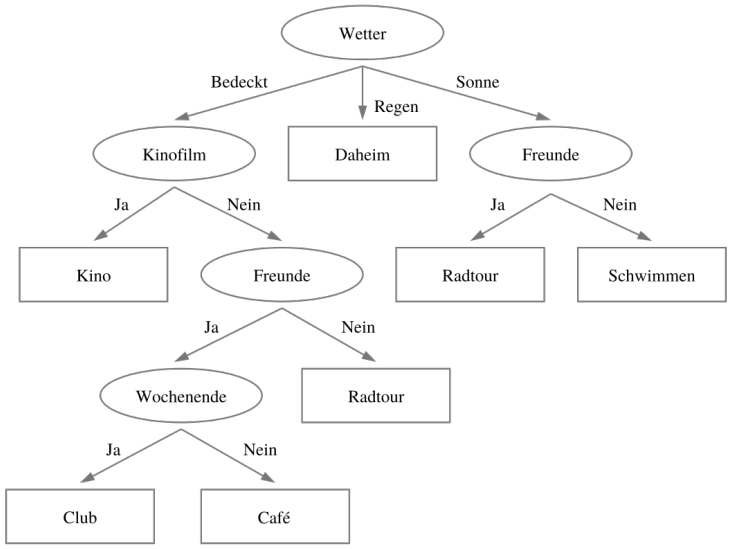

Abbildung 4.2. Aus (einer Erweiterung von) Tabelle 4.1 gewonnener Entscheidungsbaum.

Schwarmintelligenz. Das Konzept der *Schwarmintelligenz* imitiert den in der Natur beobachteten Effekt, dass eine Gruppe einfach entscheidender Individuen in ihrer Gesamtheit zu Lösungen für komplexe Aufgabenstellungen gelangen kann. Ameisen beispielsweise hinterlassen auf dem Weg von ihrem Nest zur Nahrungsstelle Duftmarken, an denen sie sich später orientieren. Der kürzeste Weg ist höher frequentiert, sodass eine größere Konzentration der Duftstoffe auftritt, und dieser Weg sich im Laufe der Zeit als der von den Ameisen bevorzugte durchsetzt. Dieses Populationsverhalten lässt sich algorithmisch beschreiben und führt zu effizienten Lösungsansätzen für schwer handhabbare Probleme, wie dem des Handlungsreisenden. Dabei werden für einen gewissen Zeitraum virtuelle Ameisen auf Reisen zwischen den Städten geschickt, die Markierungen unterschiedlicher Intensität hinterlassen. Dabei ist die Intensität der Markierung um so größer, je kürzer der zurückgelegte Weg seit der letzten Stadt war. Man sorgt nun dafür, dass die virtuellen Ameisen Wege mit einer intensiveren Markierung mit einer größeren Wahrscheinlichkeit wählen. Nach einer gewissen Zeit kristallisiert sich eine bevorzugte Rundreise der Ameisen heraus, die in der Regel gut, jedoch nicht optimal ist. Abbildung 4.3 zeigt einen so konstruierten Weg.

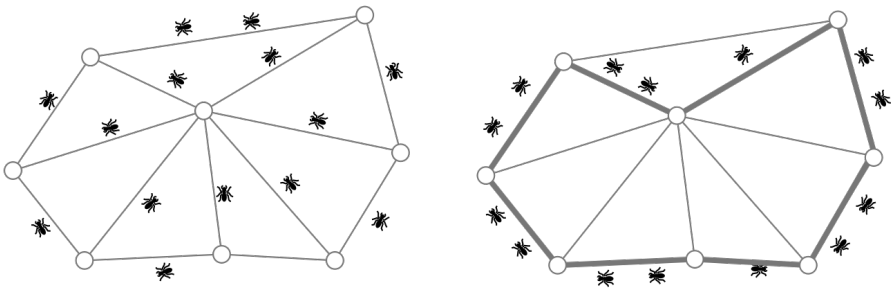

Abbildung 4.3. Für einige komplexe Problemstellungen führen Ameisenalgorithmen zu qualitativ hochwertigen Ergebnissen.

Neuronale Netze. Bei *neuronalen Netzen* wird die Funktionsweise der Neuronen im menschlichen Gehirn imitiert, indem diese als Schaltzellen mit mehreren Eingangs- und einem Ausgangssignal beschrieben werden, die miteinander vernetzt sind. Das Ausgangssignal ergibt sich dabei als gewichtete Summe der Eingangssignale, wobei die Gewichte in einer Trainingsphase zu bestimmen sind. Besonders gut eignen sich neuronale Netze zur Mustererkennung, das heißt, wenn beliebige Muster mit Hilfe gewisser Referenzmuster klassifiziert werden sollen, wie in Abbildung 4.4 schematisch dargestellt ist. Die Bestimmung der erforderlichen Gewichte erfolgt in einer Trainingsphase, in der viele bekannte Muster den jeweiligen Referenzmustern zugeordnet werden.

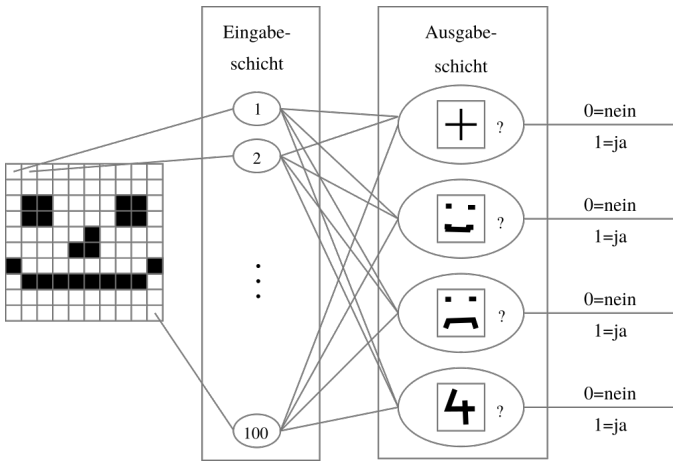

Abbildung 4.4. Musterklassifizierung mit Hilfe eines neuronalen Netzes. Jedem Bildpixel und jedem Referenzmuster ist ein Neuron zugeordnet.

KAPITEL 5

Visualisieren und Programmieren in MATLAB

MATLAB steht für *Matrix Laboratory* und ist eine Programmierumgebung, die viele mathematische Operationen wie das effiziente Lösen linearer Gleichungssysteme und gewöhnlicher Differentialgleichungen sowie Methoden zur Eigenwertberechnung von Matrizen bereitstellt. Darüberhinaus gibt es zahlreiche Möglichkeiten zur Visualisierung mathematischer Objekte. Im Gegensatz zu C++ müssen Programme nicht kompiliert werden und die Verwendung von Listen und Feldern ist deutlich einfacher. Insbesondere müssen Variablentypen nicht spezifiziert werden, sondern werden automatisch angepasst. Ein Nachteil ist, dass die Laufzeit von Programmen gerade bei der Verwendung von Schleifen meist länger als in kompilierten Programmiersprachen ist. Daher ist es oft sinnvoll, Probleme mit einem C++-Programm zu lösen, die Daten in einer Datei abzuspeichern und schließlich mit MATLAB die Datei auszulesen und die Daten grafisch aufzubereiten.

Die freie Software GNU Octave ist zu MATLAB weitestgehend kompatibel.

5.1. Aufbau der Programmierumgebung

Die Programmierumgebung besteht aus verschiedenen Teilen. Zentrale Bestandteile sind das Eingabefenster, der Editor und die Hilfefunktion, Abbildung 5.1 zeigt die Standardoberfläche von MATLAB.

Eingabefenster Im Eingabefenster oder *Command Window* können MATLAB-Befehle direkt und interaktiv eingegeben und ausgeführt werden. Auch einige Unix-Befehle für Dateioperationen können wie in einer Konsole ausgeführt werden. Werden Befehle nicht mit einem Semikolon abgeschlossen, so wird das Ergebnis der Berechnung direkt angezeigt. Mit `clc` wird das Fenster geleert und mit `clear` die aktuellen Variablen gelöscht.

Editor Im Editor werden Programme geschrieben, die dann aus dem Eingabefenster heraus gestartet werden können. Ein in MATLAB geschriebenes Programm wird häufig auch als MATLAB-Skript bezeichnet. Der von MATLAB bereitgestellte Editor verfügt über ein hilfreiches Syntax-Highlighting sowie Einrückfunktionen zur Verbesserung der Übersichtlichkeit. Der Editor kann über das Schaltsymbol *New Script* geöffnet werden. Selbstverständlich können MATLAB-Skripte auch mit jedem anderen Texteditor geschrieben werden.

Im Fenster *Workspace* wird eine Übersicht der aktuell definierten Objekte angezeigt. Außerdem findet sich in der Standardoberfläche links ein Dateimanager, in dem das aktuelle Arbeitsverzeichnis geöffnet ist.

Hilfe Erklärungen und Beispiele zu MATLAB-Befehlen findet man über den entsprechenden Menüpunkt oder durch Verwendung der Befehle `help` und `doc` im Eingabefenster.

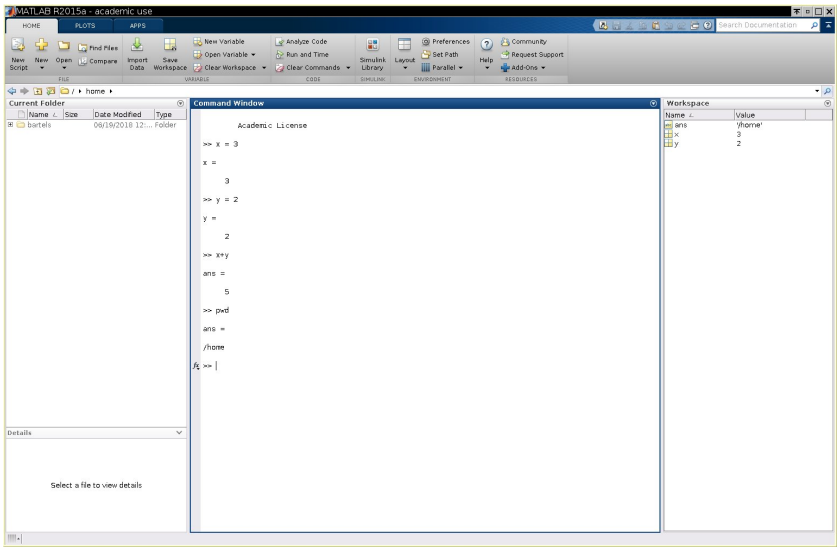

Abbildung 5.1. Standardoberfläche von MATLAB mit dem Eingabefenester (Command Window) in der Mitte.

5.2. Matrizen, Vektoren und Listen

Wichtigstes Konzept von MATLAB ist die Verwendung mehrdimensionaler Felder, die auf unterschiedliche Weisen benutzt werden. Wir verwenden die folgende Terminologie spezieller Felder:

- Unter *Matrizen* werden zweidimensionale Felder verstanden, deren Einträge meist Gleitkommazahlen sind.
- Unter *Vektoren* werden Matrizen verstanden, die nur aus einer Zeile oder einer Spalte bestehen.
- Unter *(Index-) Listen* werden Vektoren verstanden, die nur positive ganze Zahlen oder nur boolesche Werte als Einträge enthalten.

Man beachte, dass Vektoren und Listen spezielle Matrizen sind.

Erzeugung von Matrizen. Eine Matrix kann direkt durch eckige Klammern und die Einträge definiert werden, wobei Einträge einer Zeile durch Kommas und verschiedene Zeilen durch Semikolons getrennt werden. Zuweisungen werden mit dem Gleichheitszeichen realisiert. Durch den Befehl

```
A = [1.1,2.2;3.3,4.4;5.5,6.6];
```

wird beispielsweise eine Matrix `A` mit 3 Zeilen und 2 Spalten definiert. Das trennende Komma zwischen Zeileneinträgen kann durch ein oder mehrere Leerzeichen ersetzt werden. Spezielle Matrizen wie die Einheitsmatrix oder Matrizen, die in jedem Eintrag den Wert 0 oder 1 haben, können mit den Befehlen **`eye`** sowie **`zeros`** und **`ones`** definiert werden:

```
E = eye(5);
```

```
Z = zeros(2,4);
A = ones(5,3);
```

Die Zahl(en) in Klammern geben dabei die entsprechenden Dimensionen an. Listen und Vektoren mit gleichen Abständen zwischen den Einträgen lassen sich durch Angabe des Abstands oder der Anzahl der Einträge mit den Kommandos

```
I = a:incr:i;
X = linspace(a,b,N);
```

generieren. Durch den ersten Befehl wird eine Liste oder ein Vektor mit Einträgen erzeugt, die bei `a` beginnen und in Abständen von `incr` bis maximal `b` gehen, wobei `incr` auch negativ sein darf. Im zweiten Fall ist das Inkrement gegeben durch $(\mathtt{b}-\mathtt{a})/(\mathtt{N}-1)$. Beispielsweise werden durch die Kommandos

```
I = 1:2:9;
X = 0.0:.1:1.0;
Y = linspace(0,1,11);
```

eine Indexliste `I` mit Einträgen 1, 3, 5, 7, 9 sowie identische Zeilenvektoren `X` und `Y` mit den elf Einträgen 0.0, 0.1, 0.2, ..., 1.0 generiert.

Rechnen mit Matrizen. Die komponentenweise Addition und Subtraktion für Matrizen gleicher Größe lassen sich direkt angeben:

```
J = [1,2,3] + [4,5,6];
Y = [1.1;2.2] - [0.2;0.3];
```

Die Multiplikation von Matrizen entspricht der gewöhnlichen Matrixmultiplikation und muss wohldefiniert sein, das heißt die Größen der beteiligten Matrizen müssen kompatibel sein. Durch

```
b = [1.0,2.0,3.0;4.0,5.0,6.0] * [1.0;2.0;3.0];
```

wird beispielsweise ein Vektor `b` mit den Einträgen 14.0 und 32.0 definiert. Die Indizierung von Matrizen beginnt in MATLAB mit dem Index 1, der Zugriff auf Einträge erfolgt durch Verwendung runder Klammern:

```
A = [1,2;3,4];
det_A = A(1,1)*A(2,2)-A(1,2)*A(2,1);
```

Mit zulässigen Indexlisten kann man auf Teilmatrizen zugreifen, etwa wird durch

```
A = [1,2;3,4;5,6]; I = [1,2]; J = 2; B = A(I,J);
```

eine Matrix `B` mit zwei Zeilen und einer Spalte und den Einträgen 2 und 4 definiert. Wenn auf alle Zeilen oder Spalten zugegriffen werden soll, kann ein Doppelpunkt als Platzhalter verwendet werden: Durch

```
A = [1,2;3,4;5,6]; I = [1,2]; B = A(I,:);
```

wird eine Matrix B mit zwei Zeilen und zwei Spalten und den Einträgen 1, 2, 3 und 4 definiert. Die Transposition (das Spiegeln der Einträge an der Hauptdiagonalen) einer Matrix erfolgt mit einem Apostroph:

```
A = [1,2;3,4;5,6]; B = A';
```

Algebraische Operationen können durch Voranstellen eines Punktes vor dem Operator komponentenweise angewendet werden, um zwei Listen oder Matrizen gleicher Größe zu verknüpfen. Durch die Folge von Anweisungen

```
I = [1,2]; J = [3,4]; K = I.*J;
```

wird beispielsweise eine Liste K mit den zwei Einträgen 3 und 8 erzeugt. Auf Matrizen können mathematische Funktionen angewendet werden, was in der Regel komponentenweise geschieht, beispielsweise erhält man mittels

```
X = 0.0:0.01:1.0;
Y = sin(X);
```

einen Vektor Y der die Werte der Sinusfunktion in den Punkten $0.0, 0.1, \ldots, 1.0$ enthält. In Tabelle 5.1 sind einige wichtige Kommandos zum Arbeiten mit Matrizen, Vektoren und Listen zusammengefasst.

`[a,b,...;x,y,...]`	Definition eines Arrays
`[a,b,...],[x;y;...]`	Definition eines Zeilen- oder Spaltenvektors
`A(i,j)`, `I(j)`	Zugriff auf die Einträge eines Arrays
`a:b`, `a:step:b`	Liste von a bis b mit Schrittweite 1 oder step
linspace`(a,b,N)`	Äquidistante Partitionierung des Intervalls $[a, b]$
`A(i,:)`, `A(:,j)`	i-te Zeile und j-te Spalte von A
`A(I,J)`	Teilmatrix definiert durch Listen I und J
ones`(n,m)`	Array mit Einträgen 1
zeros`(n,m)`	Array mit Einträgen 0
randn`(n,m)`	Matrix mit zufällig generierten Einträgen
`A'`	Transponierte Matrix
`A+B`, `A-B`, `A*B`	Addition, Subtraktion und Produkt von Matrizen
`A.^n`	Komponentenweises potenzieren
sort`(A)`	Sortierung der Einträge eines Arrays
sum`(A,1)`, **sum**`(A,2)`	Spalten- und zeilenweise Summenbildung
max`(A)`, **min**`(A)`	Spaltenweise Extremwerte eines Arrays
size`(A)`, **length**`(I)`	Dimensionen eines Arrays und Länge einer Liste
find`(A)`	Finden der Nichtnulleinträge einer Matrix
unique`(I)`	Eliminieren doppelter Einträge einer Liste

Tabelle 5.1. Erstellung und Manipulation von Matrizen.

5.3. Plotten von Funktionen und Vektorfeldern

MATLAB stellt eine Vielzahl von Routinen zur ansprechenden grafischen Darstellung von Datensätzen und mathematischen Objekten bereit. Eine Übersicht hilfreicher Befehle, welche wir im Folgenden zum Teil genauer erläutern, findet sich in Tabelle 5.2.

`plot(X,Y,'-*')`	Polygonzug durch Punkte $(\texttt{X(k)},\texttt{Y(k)})$ in $\mathbb{R}^2$
`plot3(X,Y,Z)`	Darstellung von Kurven in $\mathbb{R}^3$
`hold on, hold off`	Darstellung mehrerer Objekte in einer Grafik
`meshgrid`	Erzeugung eines Gitters
`mesh(X,Y,Z), surf(X,Y,Z)`	Darstellung eines zweidimensionalen Graphen
`contour`	Darstellung von Höhenlinien
`quiver`, `quiver3`	Visualisierung von Vektorfeldern
`axis([x1,x2,...])`	Begrenzung des dargestellten Bereichs
`pbaspect`	Änderung des Verhältnisses der Achseneinheiten
`xlabel`, `ylabel`	Beschriftung der Achsen
`legend`	Einfügen einer Legende
`figure(k)`	Öffnen oder Auswahl eines Grafikfensters
`clf`	Leeren des aktuellen Grafikfensters
`subplot(n,m,j)`	Darstellung mehrerer Plots in einem Fenster
`print`	Exportieren einer Grafik

Tabelle 5.2. Darstellung und Bearbeitung grafischer Objekte.

Graphen eindimensionaler Funktionen. Zur grafischen Darstellung eindimensionaler Funktionen eignet sich der Befehl `plot`. Dazu werden zwei Vektoren mit Argumenten und zugehörigen Funktionswerten benötigt. Die Routine erzeugt dann einen Polygonzug durch die dadurch definierten Punkte:

```
X = 0:.01:1;
U = sin(X);
plot(X,U);
```

Mit weiteren optionalen Angaben kann die Darstellung der Kurve beeinflusst werden, beispielsweise wird der Polygonzug durch

```
plot(X,U,'+r');
```

rot eingefärbt und durch Kreuze $(+)$ dargestellt. Sollen mehrere Graphen in einem Plot erscheinen, bieten sich die Kommandos `hold on` beziehungsweise `hold off` an, wobei ersteres vermeidet, dass die Grafik bei einem neuen Aufruf von `plot` gelöscht wird und letzteres genau dies bewirkt. Mit dem Befehl `legend` kann eine Legende erzeugt werden:

```
X = 0:.01:1;
U = sin(X); plot(X,U,'r'); hold on;
V = cos(X); plot(X,V,'b');
W = X.^2;   plot(X,W,'k'); hold off;
legend('sin','cos','x^2');
```

Darstellung von Kurven. Eine Kurve, das heißt eine Abbildung $c : [a, b] \to \mathbb{R}^3$ mit einem Parameterintervall $[a, b] \subset \mathbb{R}$, lässt sich mit Punkten im Intervall $[a, b]$ und drei Vektoren, die die drei Komponenten der Funktionswerte an diesen Punkten enthalten, mit dem Befehl **plot3** darstellen. Für die Helix $t \mapsto (\sin(t), \cos(t), 5t)$, $t \in [0, 10\pi]$ erfolgt dies beispielsweise mit folgenden Kommandos:

```
T = [0:.01:10*pi];
C1 = sin(T); C2 = cos(T); C3 = 5*T;
plot3(C1,C2,C3);
```

Man beachte, dass die Punkte im Parameterbereich nicht für den Aufruf der Routine **plot3** benötigt werden.

Graphen zweidimensionaler Funktionen. Die Visualisierung von Funktionen die in zweidimensionalen Gebieten definiert sind, ist etwas aufwendiger, da ein geeignetes Gitter benötigt wird. Dies lässt sich mit dem Befehl **meshgrid** generieren:

```
[X,Y] = meshgrid(a:dx:b,c:dy:d);
```

Nach diesem Befehl sind `X` und `Y` Matrizen, die jeweils zusammengehörende x- und y-Koordinaten enthalten, welche die Gitterpunkte $p_{ij} = (x_{ij}, y_{ij})$ im Rechteck $[\mathtt{a}, \mathtt{b}] \times [\mathtt{c}, \mathtt{d}]$ mit Abständen `dx` und `dy` definieren. Mit zugehörigen Funktionswerten $f(p_{ij})$ erlauben die Befehle **surf** oder **mesh** dann eine grafische Darstellung. Die Funktion $f(x, y) = \sin(x)\cos(y)$ kann so im Bereich $[0, 1] \times [\pi/2, 3\pi/2]$ beispielsweise folgendermaßen dargestellt werden:

```
[X,Y] = meshgrid(0:.1:1,pi/2:.1:3*pi/2);
U = sin(X).*cos(Y);
surf(X,Y,U);
colorbar;
```

Das Kommando **colorbar** sorgt für die Einblendung einer Farbskala.

Darstellung von Höhenlinien und Vektorfeldern. Wird mittels des Befehls **meshgrid** und Angabe zugehöriger Funktionswerte eine Gitterfunktion $\phi : Q \to \mathbb{R}$ auf einem Rechteck $Q = [a, b] \times [c, d]$ definiert, so erlaubt die Routine **contour** die Darstellung von Höhenlinien:

```
[X,Y] = meshgrid(0:.1:1,pi/2:.1:3*pi/2);
U = sin(X).*cos(Y);
contour(X,Y,U);
```

Zur Visualisierung von Vektorfeldern, das heißt von vektorwertigen Abbildungen $F : Q \to \mathbb{R}^d$ mit $Q \subset \mathbb{R}^d$ für $d = 2$ oder $d = 3$, eignen sich die Befehle **quiver** und quiver3. Dabei werden die Koordinaten von Punkten im Definitionsbereich Q sowie die Komponenten der zugehörigen Funktionswerte mit vier beziehungsweise sechs Vektoren oder Matrizen spezifiziert. Mit den Befehlen

```
[X,Y] = meshgrid(0:.1:1,0:.1:1);
F_1 = sin(Y); F_2 = cos(X);
quiver(X,Y,F_1,F_2);
```

wird beispielsweise das zweidimensionale Vektorfeld $F(x, y) = [\sin(y), \cos(x)]$ im Bereich $[0, 1]\times[0, 1]$ grafisch dargestellt. Eine wichtige Klasse von Vektorfeldern sind Gradientenfelder von Funktionen $f : Q \to \mathbb{R}$, also die entsprechenden Abbildungen $F = \nabla f : Q \to \mathbb{R}^d$. Für eine Gitterfunktion ϕ wie oben kann ihr Gradient mit der Routine **gradient** approximativ berechnet werden. Interessant ist beispielsweise die gemeinsame Darstellung von Höhenlinien und dem Gradientenfeld einer Funktion:

```
dx = .05; dy = .05;
[X,Y] = meshgrid(0:dx:1,0:dy:1);
U = sin(X).*cos(Y);
[F_1,F_2] = gradient(U,dx,dy);
contour(X,Y,U); hold on
quiver(X,Y,F_1,F_2); hold off
```

Dabei bestätigt sich die Tatsache, dass Gradienten orthogonal zu Höhenlinien stehen und in die Richtung des steilsten Anstiegs zeigen.

Abspeichern von Grafiken. Einfache Grafiken wie Graphen eindimensionaler Funktionen speichert man am besten als **eps**- oder pdf-Datei ab, wobei das **eps**-Format für Zwecke der grafischen Nachbereitung besser geeignet ist. Aufwendige Grafiken wie Graphen zweidimensionaler Funktionen sollten speichereffizient zum Beispiel im jpg-Format abgespeichert werden. Durch die Befehle

```
figure(1); print -deps einfacher_plot.eps
figure(2); print -djpeg aufwaendiger_plot.jpg
```

werden die im ersten Grafikfenster dargestellten Informationen im **eps**- und die im zweiten im jpg-Format abgespeichert.

5.4. Programmieren in MATLAB

MATLAB-Befehle können nicht nur interaktiv im Eingabefenster verwendet werden, sondern auch in Form von Skripten oder Funktionen zu in sich geschlossenen Einheiten zusammengefasst werden. Als vollwertige Programmiersprache enthält MATLAB außerdem die üblichen Kontrollstrukturen, die wir bereits aus der Sprache C++ kennen.

Programmdateien. MATLAB-Programme, die auch als M-Skripte bezeichnet werden, sind Folgen von MATLAB-Kommandos, die in einer Datei mit der Endung .m abgespeichert werden also beispielsweise matlab_prog.m. Solche Programme können aus dem Eingabefenster heraus durch Angabe des Dateinamens ohne die Endung gestartet werden, sofern sie sich im aktuellen Arbeitsverzeichnis befinden. Alternativ kann ein Programm aus der Menüleiste heraus gestartet werden. Befindet sich der folgende Quelltext beispielsweise in der Datei mit dem Namen hello.m, so kann das Programm durch Eingabe von hello im Eingabefenster gestartet werden:

```
1 % Programm hello.m, Start durch Eingabe "hello"
2 disp('Hello World!')
```

Bei der ersten Zeile handelt es sich um eine Kommentarzeile, welche in MATLAB durch das Prozentzeichen eingeleitet wird. Der Befehl **disp** sorgt für die Ausgabe eines Textes oder Wertes auf dem Bildschirm. Die interaktive Eingabe von Werten während der Programmlaufzeit kann durch den Befehl **input** realisiert werden. Dabei kann ein als Eingabeaufforderung dienender Text direkt als Argument angegeben werden, so wie im folgenden Beispielskript:

```
1 % Programm zur Berechnung eines Quotienten
2 x = input('Dividend eingeben: ');
3 y = input('Divisor eingeben: ');
4 disp('Der Quotient ist');
5 disp(x/y);
```

Zahlenwerte werden in der Regel als Gleitkommazahlen interpretiert. Wird hier beispielsweise für `x` der Wert 1 und für `y` der Wert 2 eingegeben, so liefert dieses Programm im Gegensatz zu C++ den Wert 0.5. Laufende MATLAB-Programme können im Eingabefenster durch die Tastenkombination `Ctrl-C` abgebrochen werden.

Variablen. In MATLAB stehen die üblichen Variablentypen wie `int`, `double`, `logical` und `char` zur Verfügung. Eine Deklaration ist nicht erforderlich, diese erfolgt automatisch bei der Definition von Variablen. Darüberhinaus wird der Typ automatisch an den Wert eines Ausdrucks angepasst. In den Zeilen

```
k = 1;
k = k/2;
str = 'zeichenkette';
```

wird `k` bei Definition als Variable vom Typ `double` festgelegt und hat nach dem zweiten Kommando den Wert 0.5. Mit den arithmetischen Operationen und Vergleichen sowie konstanten Werten, die für die verschiedenen Variablen definiert sind, lassen sich Ausdrücke bilden, die dann Variablen zugewiesen werden können. Eine Auswahl der wichtigsten Operationen und konstanten Werte sowie mathematischer Grundfunktionen ist in Tabelle 5.3 aufgeführt.

`+,-,*,/`	Arithmetische Grundoperationen
`a==b`, `a~=b`	Logischer Test auf Gleich- oder Ungleichheit
`a<b`, `a<=b`	Logischer Vergleich zweier Variablen
`b1&&b2`, `b1||b2`, `~b`	Logisches UND und ODER sowie NEGATION
`true, false`	Logische Werte wahr und falsch
`sqrt``(x)`, `x^y`	Quadratwurzel und Potenzen
`exp``(x)`, **`log`**`(x)`	Exponentialfunktion und Logarithmus
`sin``(x)`, **`cos`**`(x)`, **`pi`**	Trigonometrische Funktionen und Konstante π
`norm``(x,p)`	p-Norm eines Vektors

Tabelle 5.3. Grundoperationen, Vergleiche und elementare Funktionen in MATLAB.

Kontrollstrukturen. In MATLAB stehen die üblichen Kontrollstrukturen zur Verfügung, deren Syntax wir im Folgenden kurz erläutern.

`if-else`-Verzweigungen haben in MATLAB die folgende Syntax:

```
if bed_1
  block_1
elseif bed_2
  block_2
...
elseif bed_n
  block_n
else
  block
end
```

Dabei sind `bed_1`,...,`bed_n` boolesche Ausdrücke und `block_1`,...,`block_n` sowie `block` Blöcke von Kommandos, also Folgen von Zuweisungen, Funktionsaufrufen oder Kontrollstrukturen. Besonders zu beachten ist das gesonderte Schlüsselwort **`elseif`**, welches anstatt einer Schachtelung separater **`if`**-Abfragen verwendet wird.

`switch`-Verzweigungen können für eine bessere Lesbarkeit eines Programms sorgen und stellen häufig eine Alternative zu **`if-else`**-Abfragen mit vielen Fallunterscheidungen dar. Ihre Syntax ist gegeben durch

```
switch(a)
case wert_1
  block_1
case wert_2
  block_2
...
case wert_n
  block_n
otherwise
  block
end
```

Dabei ist `a` ein Ausdruck und die Verzweigung bewirkt die Ausführung des entsprechenden Anweisungsblocks `block_1`, `block_2`,... oder `block_n`, falls der Wert von `a` mit einem der Werte `wert_1`,...,`wert_n` übereinstimmt. Anderenfalls wird der Anweisungsblock `block` ausgeführt. Im Gegensatz zu C++ sind hier keine **`break`**-Anweisungen nötig.

`while`-Schleifen können durch die Syntax

```
while bed
  block
end
```

realisiert werden, wobei bed wieder ein boolescher Ausdruck und `block` ein Anweisungsblock ist, der wiederholt wird, solange der Ausdruck bed den Wahrheitswert `true` hat. Ein Analogon zu den **do-while**-Schleifen aus C++ existiert in MATLAB nicht.

for-Schleifen iterieren in MATLAB über eine Indexliste oder einen Vektor und besitzen die Syntax

```
for var = Z
  block
end
```

Dabei ist `Z` die Indexliste beziehungsweise der Vektor, welcher von der Schleifenvariable `var` durchlaufen wird und `block` ein Anweisungsblock, der in jedem Schleifendurchlauf ausgeführt wird.

Die folgenden vier Beispielprogramme veranschaulichen die Verwendung von **if-else**- und **switch**–Verzweigungen sowie **while**- und **for**-Schleifen. Das erste Programm entscheidet mithilfe einer **if-else**-Verzweigung, ob eine eingegebene Zahl größer, kleiner oder gleich Null ist:

```
1  x = input('x = ');
2  if x < 0
3      disp('x ist kleiner als Null');
4  elseif x > 0
5      disp('x ist groesser als Null');
6  else
7      disp('x ist Null');
8  end
```

Als zweites Beispiel betrachten wir ein Programm, welches eine als ganze Zahl eingegebene Schulnote mit Hilfe einer **switch**–Verzweigung auswertet:

```
1  note = input('Note eingeben: ');
2  switch note
3      case 1
4          disp('Sehr gut!')
5      case 2
6          disp('Gut')
7      case 3
8          disp('Befriedigend')
9      case 4
10         disp('Ausreichend')
11     case 5
12         disp('Mangelhaft')
13     case 6
14         disp('Ungenuegend')
15     otherwise
16         disp('Ungueltige Note')
17 end
```

Das folgende Programm veranschaulicht die endliche Rechengenauigkeit von MATLAB, indem es mithilfe einer **while**-Schleife eine Variable x so lange fortlaufend halbiert, bis aufgrund der endlichen Zahlendarstellung nicht mehr zwischen 1 und 1+x unterschieden werden kann.

```
1  x = 1;
2  while 1+x > 1
3      x = x/2;
4  end
5  disp(x);
```

Das letzte Beispielprogramm verwendet eine **for**-Schleife zur Berechnung der Summe aller ungeraden Zahlen zwischen 1 und 10:

```
1  J = 1:2:10;
2  sum = 0;
3  for j = J
4      sum = sum+j;
5  end
6  disp(sum);
```

Funktionen. In MATLAB lassen sich eigene Funktionen definieren. Die Syntax einer Funktionsdefinition ist wie folgt:

```
function [val1,...,valm] = funktions_name(arg1,...,argn)
  block
end
```

Dabei ist [val1,...,valm] ein Vektor aus Rückgabewerten und funktions_name der Bezeichner der Funktion, welche von den Argumenten arg1,...,argn abhängt. Das schließende **end** ist optional. Ähnlich wie C++ erlaubt auch MATLAB die Definition von rekursiven Funktionen, das heißt von Funktionen, die sich innerhalb ihres Anweisungsblocks selbst aufrufen. Hervorzuheben ist die Möglichkeit der Verwendung mehrerer Rückgabewerte, wobei die zugehörigen Variablen im Anweisungsblock definiert werden müssen. Funktionen, welche in Dateien im aktuellen Arbeitsverzeichnis definiert sind, werden über den Dateinamen aufgerufen, sodass es sich als sinnvolle Konvention etabliert hat, als Dateinamen den Funktionsbezeichner mit der Endung .m zu verwenden.

Als Beispiel betrachten wir den folgenden Quelltext, in welchem eine Funktion zur Bestimmung der Länge eines Vektors $v \in \mathbb{R}^n$ definiert ist.

```
1  % M-Skript veknorm.m
2  function val = veknorm(z)
3  val = 0.0;
4  for j = 1:length(z)
5      val = val+z(j)^2;
6  end
7  val = sqrt(val);
8  % end
```

Der Aufruf dieser Funktion erfolgt beispielsweise durch Eingabe der Anweisung

```
laenge = veknorm([1,2,3])
```

im Eingabefenster oder durch die Verwendung dieser Anweisung im Quelltext eines anderen Programms, vorausgesetzt es wurde `veknorm.m` als Dateiname des M-Skripts gewählt.

Es ist zu beachten, dass bei mehreren Funktionsdefinitionen innerhalb einer Datei immer nur die erste in dieser Datei definierte Funktion von außerhalb aufgerufen werden kann. Bei allen weiteren Funktionen handelt es sich automatisch um lokale Funktionen, welche nur innerhalb derselben Datei zur Verfügung stehen.

Einfache Funktionen können auch als *anonyme Funktionen* definiert werden, was besonders für die Definition im Eingabefenster relevant ist. Dazu verwendet man die Syntax

```
funktions_name = @(arg1,...,argn) ausdruck
```

Die Definition einer Funktion zur Berechnung der euklidischen Länge eines zweidimensionalen Vektors kann so beispielsweise über die Anweisung

```
laenge_2d = @(x) (x(1)^2+x(2)^2)^(1/2);
```

erfolgen. Der Aufruf erfolgt dann zum Beispiel durch `laenge_2d([1.0,2.0])`.

Zulässige Bezeichner. Bei der Wahl von Variablen-, Funktions- und Dateinamen gibt es auch in MATLAB Einschränkungen. Die folgenden Punkte sind zu beachten:

- Variablennamen bestehen aus Buchstaben, Ziffern und Unterstrichen, wobei das erste Zeichen keine Ziffer sein darf und der Variablenname nicht nur aus einem Unterstrich bestehen darf.
- Die Verwendung der Variablennamen `i` und **`pi`** sollte vermieden werden, da diese Bezeichner standardmäßig für die imaginäre Einheit i beziehungsweise die Kreiszahl π vorgesehen sind. Das Gleiche gilt für die Bezeichner mathematischer Funktionen, wie etwa **`sin`**, **`cos`** und **`exp`**, deren Verwendung als Variablenname zu Konflikten führen kann.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden, das heißt beispielsweise, dass `Hausnr` und `HausNr` zwei unterschiedliche Bezeichner sind.
- Schlüsselwörter wie **`if`**, **`elseif`**, **`else`**, **`while`**, `true` oder `false` dürfen nicht als Variablen- oder Funktionsnamen verwendet werden.
- Ebenso sollten sämtliche Schlüsselwörter sowie Bezeichner mathematischer Funktionen und Konstanten nicht als Dateinamen verwendet werden.

5.5. Weiterführende Aspekte

Numerische Mathematik. Effiziente Realisierungen gewisser Standardverfahren der numerischen Mathematik sind in MATLAB verfügbar und intuitiv benutzbar. Beispielsweise erfolgt die Berechnung der Determinante, die Bestimmung von Eigenvektoren und -werten sowie die Berechnung der Inversen einer quadratischen Matrix `A` über die folgenden Anweisungen:

```
val = det(A);
[V,D] = eig(A);
A_inv = inv(A);
```

Zur Lösung eines linearen Gleichungssystems $Ax = b$ sollte auf die Verwendung der Inversen verzichtet werden und stattdessen der *Backslash*-Operator benutzt werden:

```
x = A\b;
```

Dadurch wird ein effizienteres und stabileres Verfahren zur Lösung herangezogen. Die approximative Integration einer Funktion kann durch die Kommandos

```
f = @(x) 1./(1+x.^2);
int_f = quad(f,-1,1);
```

erfolgen. Dabei ist zu beachten, dass die zu integrierende Funktion in vektorieller Form realisiert ist, das heißt für einen Vektor als Argument einen Vektor gleicher Länge mit den entsprechenden Funktionswerten zurückgibt. Das approximative Lösen einer gewöhnlichen Differentialgleichung

$$y'(t) = f(t, y(t)), y(0) = y_0,$$

kann beispielsweise für $f(t, y) = \cos(2t)y^2$ mit der MATLAB-Routine `ode45` erfolgen:

```
T = 10; y_0 = 1;
f = @(t,y) cos(2*t)*y^2;
[t_list,y_list] = ode45(f,[0,T],y_0);
plot(t_list,y_list);
```

Die Routine `ode45` gibt dabei einen Vektor von Zeitpunkten im Intervall $[0, T]$ und die zugehörigen Funktionswerte einer Näherungslösung $\widetilde{y}$ zurück, die anschließend mittels `plot` grafisch dargestellt werden können.

Vektorisierung. Die Verwendung von Schleifen kann in MATLAB gelegentlich zu Laufzeitproblemen führen. Sofern es möglich ist, sollte die entsprechende Berechnung mittels Vektoroperationen durchgeführt werden, was als *Vektorisierung* bezeichnet wird. Als Beispiel betrachten wir die Berechnung des komponentenweisen Produkts zweier großer Vektoren:

```
 1  n = 1e8; x = rand(n,1); y = rand(n,1);
 2  res = zeros(n,1);
 3  tic;
 4  for j = 1:n
 5      res(j) = x(j)*y(j);
 6  end
 7  toc
 8  tic;
 9  res2 = x.*y;
10  toc
```

Die Befehle `tic` und `toc` erlauben eine Zeitmessung und es stellt sich heraus, dass die zweite Berechnung etwa viermal schneller ist als die erste.

Laden und Speichern von Daten in einer Datei. Für das Laden und Speichern von Daten in einer Datei existieren zwei Möglichkeiten. Einerseits können Daten in Textdateien, das heißt im lesbaren ASCII-Format, gelesen und abgespeichert werden, was beispielsweise für eine Datei `text_datei.dat` über die Befehle

```
A = load('text_datei.dat');
save text_datei.dat A -ascii;
```

erfolgen kann. Diese Befehle bewirken, dass ein Vektor oder eine Matrix `A` aus dieser Datei gelesen beziehungsweise in diese Datei geschrieben wird. Die Arbeit mit Textdateien bietet sich insbesondere dann an, wenn Daten mit Programmen anderer Programmiersprachen erzeugt worden sind und in MATLAB weiterverarbeitet werden sollen.

Um in MATLAB generierte Variablen zu späteren Weiterverwendung innerhalb MATLABs abzuspeichern, bietet sich das MATLAB-eigene `mat`-Format an. Beispielsweise werden durch die Anweisung

```
save mat_datei.mat A B c d x;
```

die Werte der Variablen `A`, `B`, `c`, `d` und `x` in der Datei `mat_datei.mat` gespeichert. Zu einem späteren Zeitpunkt können diese gespeicherten Werte dann durch die Anweisung

```
load mat_datei.mat;
```

wieder den zuvor verwendeten Variablen zugewiesen werden.

KAPITEL 6

Funktionsweise eines Compilers

6.1. Umwandlung lesbarer Programme

Eine Hochsprache wie C++ erlaubt durch die Verwendung von Schleifen, Funktionen, Rekursionen und Klassen eine sehr hohe Problemabstraktion. Diese Konstrukte müssen vom Compiler in für den Prozessor verarbeitbare Instruktionen übersetzt werden. Dieser kann jedoch kaum mehr, als Informationen aus Speicherregistern zu laden und einfache arithmetische und logische Verknüpfungen sowie Fallunterscheidungen durchzuführen. Entsprechende Prozessor-Anweisungen werden durch Folgen von Nullen und Einsen codiert, die für Menschen nur mit hohem Aufwand verständlich und zudem sehr fehleranfällig sind. Die Aufgabe von Compilern ist es, verständliche, lesbare Anweisungen in diesen systemabhängigen *Maschinencode* zu übersetzen. Dies ist jedoch mit einigen Schwierigkeiten verbunden, da beispielsweise sichergestellt werden muss, dass Befehle eindeutig definiert sind. Wie schwierig korrekte Spracherkennung im Allgemeinen sein kann, zeigt bereits der mehrdeutige Satz »Der Fahrer muss das Hindernis umfahren«, bei dem die Bedeutung nur aus dem Kontext oder einer geeigneten Betonung hervorgeht.

6.2. Beispiel eines Maschinencodes

Um die Struktur eines Maschinencodes zu illustrieren, folgen wir einem Beispiel aus dem Buch *Computer* von R. Drechsler, A. Fink und J. Stoppe (Springer, 2017) und betrachten einen 8-Bit Modellprozessor, das heißt, ein Prozessor, der aus 8 Bits bestehende Anweisungen verarbeiten kann. Das betrachtete Maschinencode-Programm besteht aus den folgenden 64 Bits:

```
00010110 00100011 00010010 00100100
00000011 01000100 00100101 11000000
```

Das Programm definiert eine Folge von acht Anweisungen und jede davon hat die Struktur:

$$\underbrace{b_0\, b_1\, b_2}_{\text{Befehl}} \quad \underbrace{b_3}_{\text{Nummer}} \quad \underbrace{b_4\, b_5\, b_6\, b_7}_{\text{Operand}}$$

Dabei wählen die Bits b_0, b_1, b_2 einen von acht Befehlen aus und Bit b_3 legt fest, ob die Bits $b_4 b_5 b_6 b_7$ als Zahl oder Adresse eines Registers interpretiert werden sollen. Wir nehmen an, dass die von 0 bis 7 numerierten Befehle gegeben sind durch folgende Operationen:

```
LOAD, STORE, ADD, SUB, COMP, JUMP, HALT, NOOP
```

Der Befehl `LOAD` kann beispielsweise eine Zahl aus einem bestimmten Register in das Arbeitsregister laden. Anschließend kann mittels `ADD` eine konkrete Zahl zum Wert des Arbeitsregister addiert werden. Die resultierenden Bedeutungen der Befehle des obigen Maschinencodes sind in Tabelle 6.1 erklärt. Es stellt sich heraus, dass die Rechnung $6 + 2$ durchgeführt wird, die sich auch mit weniger Befehlen realisieren ließe.

8-Bit-Wort	Befehl	Nr.	Op.	Interpretation
00010110	`LOAD`	1	6	Lade Zehl 6 ins Arbeitsregister (AR)
00100011	`STORE`	0	3	Speichere Wert des AR in Register 3
00010010	`LOAD`	1	2	Lade Zahl 2 ins AR
00100100	`STORE`	0	4	Speichere Wert des AR in Register 4
00000011	`LOAD`	0	3	Lade Wert des Registers 3 ins AR
01000100	`ADD`	0	4	Addiere Wert des Registers 4 zum Wert des AR
00100101	`STORE`	0	5	Speichere Wert des AR in Register 5
11000000	`HALT`	0	0	Stoppe

Tabelle 6.1. Interpretation eines Beispielmaschinencodes.

Deutlich übersichtlicher wird das Beispielprogramm in einer *Assemblersprache*. Dabei können die Befehle als Wörter angegeben und mit dem Symbol # Zahlen von Adressen unterschieden werden:

```
LOAD #6
STORE 3
LOAD #2
STORE 4
LOAD 3
ADD 4
STORE 5
HALT
```

Assemblersprachen definieren eine Zwischenstufe zwischen Maschinencodes und Programmen höherer Programmiersprachen wie MATLAB und C++.

6.3. Höhere Programmiersprachen

Unter *höheren Programmiersprachen* versteht man solche, welche durch die ihnen zugrunde liegenden Konzepte im Vergleich zum Maschinencode eine hohe Abstraktion erlauben. Insbesondere hängen sie nicht von den besonderen Eigenschaften des verwendeten Rechners ab, beispielsweise was die Speicherverwaltung betrifft. Übersetzer oder *Compiler* erzeugen aus einem Programm einer höheren Programmiersprache wie C++ einen Assembler- beziehungsweise Maschinencode. Die Programmiersprache C++ ist dabei sehr maschinennah, da sie beispielsweise durch den Einsatz von Zeigern einen sehr direkten Zugriff auf den Speicher ermöglicht. Im Gegensatz dazu wird MATLAB nicht als Compiler- sondern als *Interpretersprache* angesehen. Vereinfacht dargestellt werden dabei bereits übersetzte Programme

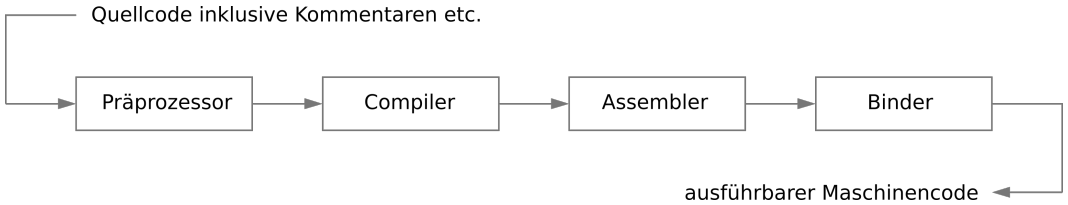

Abbildung 6.1. Schritte der Übersetzung eines Programms.

aufgerufen und weitere Teile des Programms wie Schleifen erst bei ihrem Auftreten und abhängig von den Eingabewerten durch einen *Interpreter* ausgewertet. Durch den Einsatz sogenannter *virtueller Maschinen*, die eine zusätzliche Ebene zwischen Programmiersprache und Maschinencode schaffen, verschmelzen die Konzepte von Compiler- und Interpretersprachen zunehmend. Noch etwas weiter geht die Methodik der *Just-In-Time*-Compiler, welche ein Programm analysieren und abhängig von Eingaben teilweise übersetzen oder bereits übersetzten Code wiederverwenden. Dieses Konzept kommt beispielsweise in den Programmiersprachen JavaScript und Julia sowie in neueren Versionen von MATLAB zum Einsatz. Eine Gegenüberstellung der Vor- und Nachteile klassischer Compiler- und Interpretersprachen findet sich in Tabelle 6.2.

Compilersprache (C++)	Interpretersprache (MATLAB)
⊕ schnelle Programme ⊕ Flexibilität durch direkten Zugriff auf Speicheradressen ⊕ Übersetzung unabhängig von Eingabedaten ⊖ lange Übersetzungsphase ⊖ kompliziert und fehleranfällig	⊕ direkte Nutzung, keine Übersetzung ⊕ keine Deklaration von Variablen ⊕ einfache Programme ⊖ langsame Schleifen ⊖ keine explizite Speicherverwaltung ⊖ Übersetzung abhängig von Eingabe

Tabelle 6.2. Vor- und Nachteile von Compiler- und Interpretersprachen am Beispiel von C++ und MATLAB.

6.4. Der Compiler

Zur Beschreibung der Arbeitsweise eines Compilers folgen wir in diesem Abschnitt der Darstellung des Vorlesungsskripts *Compilerbau* von U. Goltz, T. Gehrke und M. Lochau (TU Braunschweig, 2010) Ein Compiler, der aus einem Quellprogramm einen Maschinencode erzeugt, arbeitet zusammen mit einem *Präprozessor*, der aus dem rohen Quellcode Makros ersetzt und Kommentare entfernt, einem *Assembler*, der aus Assemblercode relokatiblen, das heißt verschiebbaren Maschinencode generiert, sowie einem *Binder*, der konkrete Sprungadressen einfügt und somit einen ausführbaren Machinencode erstellt. Diese Schritte sind in Abbildung 6.1 schematisch dargestellt.

Die Übersetzung eines Programms durch den Compiler erfolgt in einer Analyse- und einer Synthesephase, wobei die Analysephase aus der lexikalischen, der syntaktischen und der semantischen Analyse besteht. In der lexikalischen Analyse, die durch den *Scanner* durchgeführt wird, wird die gegebene Zeichenkette in Bestandteilte zerlegt und eine Symboltabelle angelegt. Variablen, die in diesem Kontext oft als Bezeicher betitelt werden, werden dabei numeriert. In der syntaktischen Analyse, die vom *Parser* durchgeführt wird, wird analysiert, ob beispielsweise Terme korrekt sind. Dazu wird ein Strukturbaum, auch als Syntaxbaum bezeichnet, erstellt, der die verwendeten Operationen widergibt. In der semantischen Analyse werden die auftretenden Bezeichner mit Attributen wie Variablentyp und Gültigkeitsbereich versehen und die Wohldefiniertheit der verwendeten Operationen geprüft. Dies garantiert die statische semantische Korrektheit, das heißt die formale Wohlgestelltheit der Berechnungen unabhängig von konkreten Eingabedaten, die möglicherweise zu dynamischen Semantikfehlern wie Division durch Null führen können. Die anschließende Synthese-Phase besteht aus verschiedenen Schritten wie der Zwischencode-Erzeugung, einer Code-Optimierung und der Maschinencode-Generierung. Im Zwischencode werden beispielsweise Formeln mit Hilfsvariablen in ihre Bestandteile zerlegt und als Folgen einfacher Dreiadressbefehle der Form

```
tmp_l = id_j op id_k;
```

dargestellt. Die darauf folgende Codeoptimierung entfernt überflüssige Zuweisungen. Bei der finalen Maschinencode-Erzeugung werden die einfachen Befehle des optimierten Assemblercodes in Maschinenbefehle übersetzt. Diese letzte Phase der Übersetzung ist von dem verwendeten Rechner abhängig und damit Teil der *Back-End-Phase*, die im Gegensatz zur *Front-End-Phase* den maschinenabhängigen Teil der Übersetzung bezeichnet. Die schrittweise Übersetzung der Zuweisung `position := initial + rate*60` in Assemblercode ist in Abbildung 6.2 dargestellt.

6.5. Methodik des Scanners

Die lexikalische Analyse eines Quellprogramms basiert auf der Theorie regulärer Sprachen. Eine Sprache ist dabei eine Menge L von Wörtern über einem Alphabet Σ beispielsweise

$$\Sigma = \{a, b, c\}, \quad L = \{a, b, ab, ac, cba\}.$$

Reguläre Sprachen sind solche, die sich aus aus der leeren Menge und den einelementigen Sprachen durch Vereinigung, Konkatenation und Exponenzieren bilden lassen. Mit regulären Ausdrücken können reguläre Sprachen durch Formeln definiert werden. Als Operationen sind dabei Auswahl, Konkatenation, und Exponenzieren mit den im Alphabet vorhandenen Buchstaben erlaubt. Zusätzlich steht das leere Wort ε zur Verfügung, das beispielsweise neutral bezüglich Konkatenation ist.

Beispiele 6.1. (i) Sei $\Sigma = \{a, b\}$. Die Auswahl $a|b$ erzeugt $L = \{a, b\}$, die Konkatenation ab erzeugt $L = \{ab\}$, das Exponenzieren a^* erzeugt $L = \{\varepsilon, a, aa, aaa, \dots\} = \{a^n : n \geq 0\}$.
(ii) Mit $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$ und dem regulären Ausdruck

$$(a|b|\dots|z)(a|b|\dots|z|0|1|\dots|9)^*$$

wird eine Menge zulässiger Variablennamen definiert, die mit einem Buchstaben beginnen.

Durch reguläre Ausdrücke definierte reguläre Sprachen lassen sich algorithmisch erkennen.

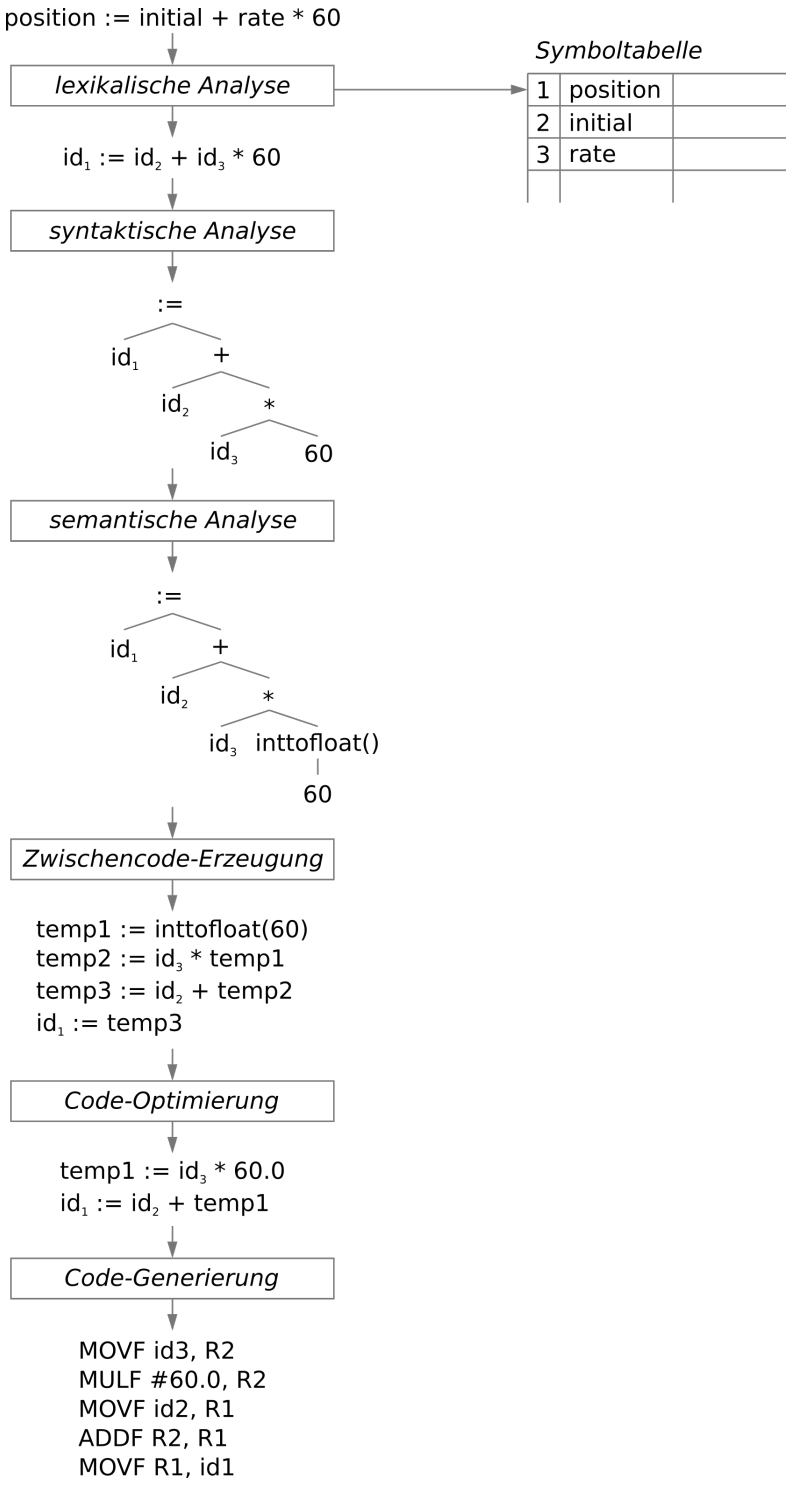

Abbildung 6.2. Schrittweise Übersetzung einer Zuweisung.

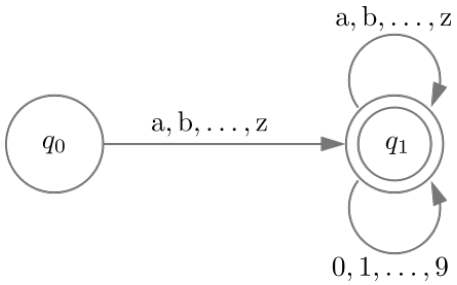

Abbildung 6.3. Deterministischer endlicher Automat, der eine Menge zulässiger Variablennamen erkennt.

Definition 6.2. Ein *nicht-deterministischer endlicher Automat* (NEA) ist ein Tupel $M = (\Sigma, Q, \Delta, q_0, F)$ mit den folgenden Eigenschaften:
(i) Σ ist ein endliches Alphabet,
(ii) Q ist eine endliche Zustandsmenge,
(iii) $q_0 \in Q$ ist ein Anfangszustand,
(iv) $F \subset Q$ ist eine Menge von Endzuständen,
(v) $\Delta \subset Q \times (\{\varepsilon\} \cup \Sigma) \times Q$ ist eine Übergangsrelation.
Ein NEA heißt *deterministischer endlicher Automat* (DEA), wenn die Übergangsrelation eine Funktion $\Delta : Q \times (\{\varepsilon\} \cup \Sigma) \to Q$ ist.

Ein Übergang von einem Zustand q und einem Wort zw zu einem neuen Zustand q' mit Wort w ist zulässig, wenn $(q, z, q') \in \Delta$ gilt beziehungsweise im deterministischen Fall $\Delta(q, z) = q'$ gilt. Die von einem NEA akzeptierte Sprache ist die Menge aller Wörter w_0, die sich in endlich vielen zulässigen Übergängen vom Anfangszustand q_0 auf das leere Wort ε und einen Endzustand $q_f \in F$ reduzieren lassen:

$$(q_0, w_0) \mapsto (q_1, w_1) \mapsto \cdots \mapsto (q_f, \varepsilon),$$

wobei jeweils $w_{k-1} = zw_k$ mit $z \in \Sigma \cup \{\varepsilon\}$ gelte. Der folgende Satz stellt eine Beziehung zwischen regulären Sprachen und Automaten her.

Proposition 6.3. (i) Zu jedem regulären Ausdruck r existiert ein NEA, der die von r definierte reguläre Sprache $L(r)$ akzeptiert.
(ii) Wird die reguläre Sprache L von einem NEA akzeptiert, so existiert ein DEA, der L akzeptiert.

Automaten werden am übersichtlichsten durch Graphen dargestellt. Die Menge der Variablennamen über dem Alphabet $\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9\}$, die mit einem Buchstaben beginnen, werden von dem in Abbildung 6.3 definierten Automaten mit $Q = \{q_0, q_1\}$ und $F = \{q_1\}$ akzeptiert.

Für die praktische Unterscheidung zwischen Bezeichnern für Variablen und Schlüsselwörtern wie `if`, `else`, `end`, `while`, `for`, `function` wird entweder ein geeigneter *lookahead* verwendet oder die Schlüsselwörter werden zunächst wie Variablennamen behandelt und anschließend extrahiert.

6.6. Methodik des Parsers

Das Konzept regulärer Ausdrücke und endlicher Automaten ist für eine Syntaxanalyse nicht ausreichend, da zum Beispiel Klammerstrukturen wie `begin ... end` nicht erkannt werden können. Ein Beispiel einer nicht-regulären Sprache sind die wohlgeformten Klammerterme, die auch als Dyck-Sprache bezeichnet werden,

$$\{w \in (a|b)^* : |w|_a = |w|_b, \forall u, v \in (a|b)^*,\, w = uv \implies |u|_a \geq |u|_b\},$$

wobei a und b öffnende beziehungsweise schließende Klammern repräsentieren und $|w|_a$ die Anzahl öffnender Klammern in w angibt. Endliche Automaten verfügen über keinen Mechanismus, der ein Zählen bestimmter Symbole realisiert. Das Konzept kontextfreier Grammatiken und damit verbundener Keller-Automaten erlaubt die Erkennung von Klammerausdrücken.

Definition 6.4. Eine *kontextfreie Grammatik* ist ein Tupel $G = (V_N, V_T, P, S)$ mit den folgenden Eigenschaften:
(i) V_N ist eine endliche Menge von Nichtterminalsymbolen,
(ii) V_T ist eine endliche Menge von Terminalsymbolen,
(iii) $P \subset V_N \times (V_N \cup V_T)^*$ ist eine Menge von Produktionen,
(iv) $S \in V_N$ ist ein Startsymbol.
Die von G generierte Sprache $L(G)$ ist die Menge aller Wörter über V_T, die sich mit den Produktionen aus dem Startsymbol ableiten lassen, indem sukzessive Nichtterminalsymbole gemäß der Produktionsregeln ersetzt werden.

Terminale stehen für Symbole, das heißt für Bezeichner, Schlüsselwörter, Operatorsymbole und Konstanten, die in der lexikalischen Analyse identifiziert worden sind. Eine Produktion $(A, \alpha) \in P$ schreibt man in der Form $A \to \alpha$ und zwei Produktionen $A \to \alpha$ und $A \to \beta$ werden zusammengefasst zu $A \to \alpha|\beta$.

Beispiel 6.5. Seien $V_N = \{\mathrm{expr}, \mathrm{op}\}$ und $V_T = \{\mathrm{id}, +, -, *, /, \hat{}, (,)\}$, wobei id für einen Bezeichner oder eine Konstante stehe, mit den Produktionen

$$\mathrm{expr} \to \mathrm{expr}\ \mathrm{op}\ \mathrm{expr} \,|\, (\mathrm{expr}) \,|\, -\mathrm{expr} \,|\, \mathrm{id}$$
$$\mathrm{op} \to + \,|\, - \,|\, * \,|\, / \,|\, \hat{}$$

sowie dem Startsymbol $S = \mathrm{expr}$. Mit der dadurch definierten kontextfreien Grammatik lassen sich zulässige arithmetische Ausdrücke definieren.

Zu jedem Wort der Sprache einer kontextfreien Grammatik lässt sich ein Strukturbaum angeben. Für die aus der mit Beispiel 6.5 erzeugten Formel $-(\mathrm{id} + \mathrm{id})$ ist der zugehörige Strukturbaum in Abbildung 6.4 gezeigt. Die Erkennung der Sprache einer kontextfreien Grammatik erfolgt mit Kellerautomaten, die einen Hilfsspeicher besitzen und eine Ausgabe erzeugen können.

Definition 6.6. Ein *Kellerautomat* ist ein Tupel $M = (\Sigma, \Gamma, \Delta, z_0, \mathcal{O})$ mit den Eigenschaften:
(i) Σ ist ein endliches Eingabealphabet,
(ii) Γ ist ein endliches Kelleralphabet,
(iii) $z_0 \in \Gamma$ ist ein Kellerstartsymbol,

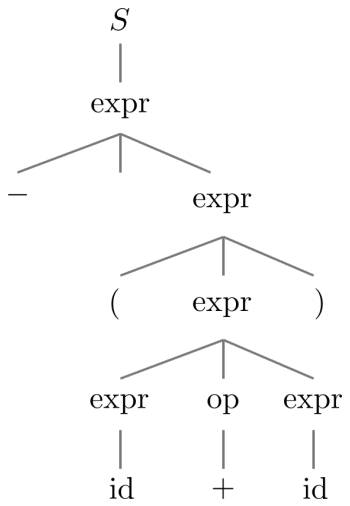

Abbildung 6.4. Strukturbaum zur Formel $-(\mathrm{id} + \mathrm{id})$.

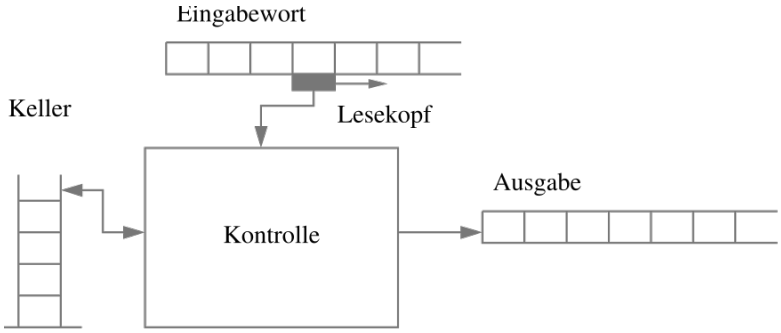

Abbildung 6.5. Schematische Darstellung der Funktionsweise eines Kellerautomaten.

(iv) $\mathcal{O}$ ist ein endliches Ausgabealphabet,
(v) $\Delta \subset \left((\Sigma \cup \{\varepsilon\}) \times \Gamma\right)^* \times (\Gamma^* \times \mathcal{O}^*)$ ist eine Übergangsrelation.
Die Menge der akzeptierten Wörter sind alle Wörter über Σ, die sich mit der Übergangsrelation auf das leere Wort und einen leeren Kellerinhalt reduzieren lassen.

Ein Kellerautomat analysiert ein Eingabewort über dem Alphabet Σ, indem er schrittweise Zeichen einliest und in Abhängigkeit vom Kellerinhalt eine Ausgabe und einen neuen Kellerinhalt erzeugt. Dieses Vorgehen ist in Abbildung 6.5 skizziert. Der Kellerinhalt dient beispielsweise dem Zählen noch offener Klammern und als Ausgabe kann der Strukturbaum erzeugt werden.

KAPITEL 7

Parallelisierung und Objektorientierte Programmierung in C++

7.1. Parallelisierung

Eine Beschleunigung von Algorithmen kann in einigen Fällen durch den parallelen Einsatz mehrerer Prozessoren erzielt werden. Soll etwa das Skalarprodukt zweier Vektoren $x, y \in \mathbb{R}^n$ berechnet werden, das heißt die Summe der Produkte der Komponenten

$$x \cdot y = \sum_{j=1}^{n} x_j y_j,$$

so kann diese Summe in Teilsummen zerlegt werden, also beispielsweise

$$x \cdot y = \sum_{j=1}^{n_1} x_j y_j + \sum_{j=n_1+1}^{n_2} x_j y_j + \cdots + \sum_{j=n_{p-1}+1}^{n_p} x_j y_j.$$

Die Teilsummen können auf verschiedenen Prozessoren gleichzeitig berechnet und anschließend zusammengefügt werden. Ein solches Vorgehen bezeichnet man als *paralleles Rechnen*. Bei der praktischen Umsetzung müssen verschiedene Aspekte beachtet werden:

- Existiert ein gemeinsamer Speicher, auf den alle Prozessoren zugreifen können, oder müssen die jeweils benötigten Daten an die Prozessoren verschickt werden?
- Greifen die verschiedenen Prozesse auf gemeinsame Variablen zu und ändern diese womöglich, so muss sichergestellt werden, dass dies in geordneter Weise passiert und nicht zu falschen Resultaten führt.

Im obigen Beispiel können Schwierigkeiten vermieden werden, indem Hilfsvariablen eingeführt werden, in denen die Teilsummen abgespeichert werden. Moderne Mehrkernrechner arbeiten mit 4 bis 32 Prozessoren, die effizient auf einen gemeinsamen Speicher zugreifen können, was als *Shared-Memory*-Architektur bezeichnet wird. Im Gegensatz dazu arbeiten große Rechencluster arbeiten mit einer *Distributed-Memory*-Architektur ohne einen gemeinsamen Speicher aller beteiligten Rechner. Die Anzahl der Teilaufgaben, die auch als Prozesse oder *Threads* bezeichnet werden, sollte die Anzahl verfügbarer Prozessoren des verwendeten Computers nicht übersteigen.

OpenMP. Die Bibliothek *OpenMP* (Open Multi-Processing) erlaubt eine sehr einfache Parallelisierung von C++-Programmen auf Rechnern mit gemeinsamem Speicher. Zur Realisierung werden mittels Compiler-Direktiven, die durch das Symbol **#** markiert werden, Teile des Programms automatisch verschiedenen Prozessoren zugewiesen. Besondere Sperr- und Synchronisationsmechanismen verhindern unerwünschte Effekte, die durch unkontrolliertes

Zugreifen auf gemeinsame Variablen entstehen könnten. Sogenannte Reduktionsmechanismen fügen private Hilfsvariablen der verschiedenen Prozesse zusammen. Solche Variablen müssen jedoch gesondert definiert werden, da standardmäßig alle Variablen gemeinsame Variablen sind. Eine Übersicht der wichtigsten Befehle in OpenMP findet sich in Tabelle 7.1.

`find /usr -name "libgomp*"`	Prüfen der Verfügbarket (unter Unix)
`g++ -fopenmp`	Kompilieren eines Programms
`#include <omp.h>`	Einbinden der OpenMP-Bibliothek
`omp_get_max_threads()`	Anzahl verfügbarer Teilprozesse
`omp_set_num_threads(np)`	Anzahl verwendeter Prozesse auf `np` festlegen
`omp_get_thread_num()`	ID des aktuellen Threads ermitteln
`omp_get_wtime()`	Messen der CPU-Zeit
`#pragma omp critical`	Anhalten aller anderen Prozesse bis die nächste Anweisung abgearbeitet ist
`#pragma omp barrier`	Warten, bis alle Prozesse an dieser Stelle im Programm angekommen sind
`#pragma omp parallel for reduction (op:var)`	Direktive zur Parallelisierung einer `for`-Schleife Erzeugung von privaten Hilfsvariablen und Reduktion der gemeinsamen Variablen `var` für den Operator `op`

Tabelle 7.1. Elementare Kommandos zur Verwendung von OpenMP.

Als erstes Beispiel betrachten wir eine parallele Version eines Hello-World-Programms, dessen Quelltext in einer Datei mit dem Namen `omp-hello.cc` abgespeichert sei:

```
1  // kompilieren: g++ omp-hello.cc -o omp-hello -fopenmp
2  #include <iostream>
3  #include <omp.h>
4  int main()
5  {
6    int np = 4;
7    omp_set_num_threads(np);
8    #pragma omp parallel
9    {
10     int tid = omp_get_thread_num();
11     #pragma omp critical
12     std::cout << "Hello World! from thread " << tid << std::endl;
13   }
14   std::cout << "Number of threads: " << np << std::endl;
15   return 0;
16 }
```

Die zur Verwendung von OpenMP benötigte Bibliothek wird in Zeile 3 in das Programm eingebunden. Die Anweisung in Zeile 7 bewirkt, dass alle im folgenden parallelisierten Programmteile von vier Threads bearbeitet werden. Anschließen bewirkt die Direktive in Zeile 8,

dass der folgende Anweisungsblock von allen Prozessen parallel ausgeführt wird: Jeder Prozess gibt den Text

```
Hello World! from thread <tid>
```

auf dem Bildschirm aus, wobei `<tid>` der ID des jeweiligen Prozesses entspricht. Die einzelnen Prozesse sind dabei mit 0 beginnend fortlaufend nummeriert. Die `critical`-Direktive in Zeile 11 ist notwendig, damit die Ausgabe geordnet stattfindet. Ohne sie würden die Ausgaben der einzelnen Prozesse in der Konsole wahllos miteinander kombiniert werden. Die Übersetzung des Programms geschieht in der Konsole mittels

```
g++ omp-hello.cc -o omp-hello -fopenmp
```

Nachdem wir die grundlegende Struktur eines mit OpenMP parallelisierten Programms nun kennengelernt haben, betrachten wir als nächstes Beispiel die parallele Berechnung der Summe

$$s = \sum_{j=1}^{n} j = \frac{1}{2} n(n+1).$$

Eine Realisierung in C++ ist durch den folgenden Quelltext gegeben:

```
 1 #include <iostream>
 2 #include <omp.h>
 3 int main()
 4 {
 5   int np = 4, sum = 0, n = 100;
 6   omp_set_num_threads(np);
 7 #pragma omp parallel for // reduction (+:sum)
 8   for (int i = 0; i <= n; ++i) {
 9     sum = sum+i;
10   }
11   std::cout << "sum = " << sum;
12   std::cout << ", correct result: " << n*(n+1)/2. << std::endl;
13   return 0;
14 }
```

Die Parallelisierung der for-Schleife geschieht im obigen Programm zunächst mit der Direktive

```
#pragma omp parallel for
```

in Zeile 7. Dies führt jedoch im Fall mehrerer Prozesse, das heißt falls `np > 1`, durch unkoordinierte Schreib- und Lesezugriffe auf die gemeinsame Variable `sum` zu fehlerhaften Ergebnissen. Durch die Verwendung der erweiterten Direktive

```
#pragma omp parallel for reduction (+:sum)
```

werden in den Prozessen Hilfsvariablen angelegt und nach Abarbeitung der Teilaufgaben zusammengefügt. Diese als Reduktion bezeichnete Vorgehensweise wird von OpenMP standardmäßig für die arithmetischen Operatoren `+`, `*` und `-` unterstützt. Außerdem werden Reduktionen für die Operatoren `max` und `min` unterstützt, etwa für den Fall dass die Bestimmung des größten Eintrags in einem Feld parallelisiert werden soll.

Skalarprodukt und Vektoraddition. Besonders effizient ist das Parallelisieren von `for`-Schleifen für elementare Operationen der linearen Algebra, wie beispielsweise zur Berechnung eines Skalarprodukts oder einer Linearkombination großer Vektoren. Das folgende C++-Programm enthält jeweils eine Unterfunktion zur Berechnung des Skalarprodukts $x \cdot y$ sowie zur Realisierung der Zuweisung

$$x := ax + by,$$

für Vektoren $x, y \in \mathbb{R}^n$. Die in den Unterfunktionen auftretenden `for`-Schleifen sind mithilfe von OpenMP parallelisiert:

```
1  #include <iostream>
2  #include <vector>
3  #include <omp.h>
4
5  typedef typename std::vector<double> doubleVec;
6
7  double scal_prod( doubleVec &x, doubleVec &y )
8  {
9    double val = 0.;
10 #pragma omp parallel for reduction (+:val)
11   for (int j = 0; j < x.size(); ++j) {
12     val += x[j]*y[j];
13   }
14   return val;
15 }
16
17 void update_vec( doubleVec &x, doubleVec &y, double a, double b )
18 {
19 #pragma omp parallel for
20   for (int j = 0; j < x.size(); ++j) {
21     x[j] = a*x[j]+b*y[j];
22   }
23 }
24
25 int main()
26 {
27   int np = 1;
28   int N = 1e8;
29   double t1, t2, val1, val2, q = .5;
30   doubleVec u(N);
31   omp_set_num_threads(np);
32   u[0] = 1.;
33   for (int j = 1; j < N; j++) {
34     u[j] = q*u[j-1];
35   }
36   t1 = omp_get_wtime();
37   val1 = scal_prod(u,u);
```

```
38    update_vec(u,u,1.,-1.);
39    val2 = scal_prod(u,u);
40    t2 = omp_get_wtime();
41    std::cout << "Ergebnis = [" << val1 << "," << val2 << "] ";
42    std::cout << "(korrekt: [4/3,0])\n";
43    std::cout << "Anzahl Prozesse = " << np << ", ";
44    std::cout << "benoetigte Zeit = " << t2-t1 << std::endl;
45    return 0;
46  }
```

Bei der komponentenweisen Berechnung der Linearkombination in der Funktion `update_vec` treten keine gemeinsamen Variablen auf, so dass keine besondere Vorsicht bei der Parallelisierung erforderlich ist. Im Hauptprogramm wird die Anzahl zu verwendender Prozesse über die Variable `np` definiert und ihre Variation erlaubt es zu untersuchen, welche Beschleunigung die Verwendung mehrerer Prozesse ermöglicht. Dabei stellt man fest, dass erstens keine weitere Beschleunigung möglich ist, wenn die Prozessanzahl die systembedingte Anzahl von Prozessoren überschreitet, und zweitens im Bereich der verfügbaren Prozessorenanzahl im Allgemeinen keine optimale Verbessung eintritt, das heißt eine Verdopplung der Prozesse nicht notwendigerweise zu einer Halbierung der Laufzeit führt. Letzteres wird durch sogenannte *Kommunikationskosten*, wie beispielsweise Wartezeiten beim Zugriff auf gemeinsame Variablen, verursacht. Falsche Resultate ergeben sich, wenn der Zusatz `reduction (+:val)` in der Unterfunktion `scal_prod` weggelassen wird.

7.2. Objektorientierte Programmierung in C++

Die Möglichkeit der objektorientierten Programmierung ist der Hauptunterschied zwischen der Sprache C++ im Vergleich zur etwas älteren Sprache C. Das Konzept der *Objektorientierung* basiert auf der Idee, den Aufbau einer Software an der Struktur des Problems auszurichten, zu dessen Lösung die Software gedacht ist. Zentraler Bestandteil der objektorientierten Programmierung sind *Klassen*, die manchmal auch als *Objekttypen* bezeichnet werden. Eine Klasse ist gewissermaßen ein erweiterter Datentyp, in welchem mehrere Variablen unterschiedlichen Typs zusammengefasst werden. Zusätzlich können Funktionen innerhalb eines solchen Variablenverbunds definiert werden. Variablen vom Typ einer Klasse werden als *Instanzen* bzw. *Objekte* dieser Klasse bezeichnet.

Ein wesentliches Ziel des objektorientierten Programmierens ist die Wiederverwendbarkeit von Quelltext und die bessere Strukturierung umfangreicher Programme. Man denke an Beispiele aus dem Alltag, wo man es gewöhnt ist verschiedene Dinge gewissen Klassen zuzuordnen. So gehören etwa die Objekte *Auto*, *Bus*, *Motorrad* und *Fahrrad* zur Klasse *Fortbewegungsmittel* oder die Objekte *Apfel*, *Birne* und *Pfirsich* zur Klasse *Obst*. Das objektorientierte Programmieren erlaubt daher häufig die wirklichkeitsnahe Lösung von Problemen.

Allerdings sei an dieser Stelle darauf hingewiesen, dass die objektorientierte Programmierung keinesfalls als Alternative zur konventionellen *imperativen Programmierung*, wie sie bisher in der Vorlesung behandelt wurde, gesehen werden sollte. Sie stellt vielmehr eine naheliegende Erweiterung des klassischen *Programmierparadigmas* dar.

Objekte und Klassen. Wir haben bereits die Klassen `std::vector` sowie `std::string` aus den Standardbibliotheken `<vector>` bzw. `<string>` kennengelernt und intuitiv benutzt. Im Folgenden erläutern wir, was genau eine Klasse in C++ auszeichnet und wie es möglich ist, eigene Klassen zu definieren und damit zu arbeiten.

Klassen sind benutzerdefinierte Datentypen deren Definition durch das Schlüsselwort **`class`** gekennzeichnet wird:

```
class Klassenbezeichner {
  typ1 var1;
  typ2 var2;
  ...
  rtyp1 methode1(arg_typ1 argument1,...) {
  ...
  }
  rtyp2 methode2(...) {
  ...
  }
};
```

Man beachte das erforderliche Semikolon am Ende der Klassendefinition. Klassen werden genau so wie Funktionen außerhalb der `main`-Funktion deklariert und definiert. Die Elemente (*Member*) einer Klasse können sowohl Variablen als auch Funktionen, die in diesem Zusammenhang als *Methoden* bezeichnet werden, sein. Die Variablen werden auch als *Attribute* der Klasse bezeichnet. Es ist eine häufig verwendete Konvention, dass der Bezeichner einer Klasse mit einem Groß-, Variablennamen hingegen mit einem Kleinbuchstaben beginnen sollten.

Eine Klasse `BankKonto` zur Verwaltung eines Kontos könnte beispielsweise die Attribute `kontonr` und `kontostand` sowie Methoden `eroeffnen()` zur Initialisierung eines neuen Kontos, `abfrage()` zur Kontostandsabfrage sowie `einzahlen()` und `auszahlen()` für entsprechende Buchungen enthalten. Das folgende Beispielprogramm enthält die Definition einer solchen Klasse sowie eine kurze `main`-Funktion zur Veranschaulichung der Erzeugung und Verwendung einer Instanz dieser Klasse:

```
 1 #include <iostream>
 2 class BankKonto {
 3   public:
 4     void eroeffnen( int num, double betrag ) {
 5       kontonr = num;
 6       kontostand = betrag;
 7     }
 8     double abfrage() {
 9       return kontostand;
10     }
11     void einzahlen( double betrag ) {
12       kontostand += betrag;
13     }
```

```
14      void auszahlen( double betrag ) {
15        kontostand -= betrag;
16      }
17    private:
18      int kontonr;
19      double kontostand;
20  };
21
22  int main() {
23    BankKonto konto;
24    konto.eroeffnen(123456, 72.15);
25    //std::cout<<konto.kontostand  // ergibt Fehler beim Kompilieren
26    std::cout << konto.abfrage() << std::endl; // Ausgabe 72.15
27    return 0;
28  }
```

Hier wird in Zeile 24 ein Objekt der Klasse `BankKonto` erzeugt (*instanziiert*), welches den Bezeichner `konto` trägt. Der Zugriff auf die Attribute einer Klasse beziehungsweise der Aufruf ihrer Methoden erfolgt durch den Punktoperator, beispielsweise wie in Zeile 25 durch `konto.eroeffnen()`. Durch sogenannte *Zugriffsmodifikatoren* kann festgelegt werden, welchen Programmteilen der Zugriff auf die jeweils folgenden Elemente einer Klasse erlaubt wird. Die in C++ existierenden Zugriffsmodifikatoren sind in der folgenden Tabelle 7.2 aufgelistet:

public:	Zugriff ist aus allen Programmteilen erlaubt.
private:	Zugriff ist nur innerhalb der Klassendefinition erlaubt.
protected:	Wie **private** und zusätzlich Zugriff aus erbenden Klassen.

Tabelle 7.2. Zugriffsmodifikatoren für Klassen in C++.

Die Deklaration der Attribute `kontonr` und `kontostand` als **private**-Elemente im obigen Beispiel verhindert einen direkten Zugriff auf diese Attribute aus der `main`-Funktion heraus, wie in der auskommentierten Zeile 26. Stattdessen kann die **public**-Methode `abfrage` verwendet werden, welche als Rückgabewert gerade den Wert des Attributs `kontostand` besitzt.

Standardmäßig, das heißt ohne Angabe eines Zugriffsmodifikators, sind alle Elemente einer Klasse **private**. Diese Abschirmung von Objekten nach außen ist ein Grundkonzept der objektorientierten Programmierung. Dadurch wird erreicht, dass der Programmierer einer Klasse die Kontrolle darüber erlangt, was mit den Attributen von Instanzen dieser Klasse geschieht, indem er eine Anzahl von Methoden vorgibt, mit denen sich die Attribute in einer vorgesehenen Art und Weise verwalten lassen. Insbesondere bei großen Projekten, an denen viele Menschen gemeinsam arbeiten, stellt dies eine Erleichterung dar und verringert die Fehleranfälligkeit. Zusätzlich erhöht dieses Vorgehen die Wiederverwertbarkeit: Da eine einmal programmierte Klasse in der Regel alle nötigen Funktionen zu ihrer Verwendung selbst mitbringt, müssen diese bei Wiederverwendung in einem neuen Programm lediglich aufgerufen und nicht neu implementiert werden.

Hinweis: Anstatt des Schlüsselworts **class** kann in C++ auch das Schlüsselwort **struct** zur Deklaration eines Klassentyps verwendet werden. Dieses Schlüsselwort entstammt der Programmiersprache C, wo es zur Definition von Variablenverbünden (sogenannten *Strukturen*) Verwendung findet. Diese fassen in C mehrere Variablen zu einem neuen Typ zusammen, können jedoch keine Methoden enthalten. In C++ können auch bei Verwendung des Schlüsselworts **struct** Methoden deklariert und definiert werden. Der einzige Unterschied zur Deklaration einer Klasse mit **class** ist der standardmäßig verwendete Zugriffsmodifikator: Im Falle von **struct** sind alle Attribute und Methoden standardmäßig **public**, im Falle von **class** wie oben bereits erwähnt **private**.

Kontruktoren und Destruktoren. Ein *Konstruktor* ist eine spezielle Methode einer Klasse, die immer dann aufgerufen wird, wenn ein Objekt dieser Klasse instanziiert wird. Damit stellen Kontruktoren eine Möglichkeit dar, ein neu erstelltes Objekt direkt zu initialisieren, ähnlich der Initialisierung einer Variablen einfachen Typs direkt bei ihrer Deklaration mithilfe des Zuweisungsoperators. Ein Konstruktor hat dabei immer den selben Bezeichner wie die Klasse selbst und *keinen* Rückgabetyp (auch nicht **void**). Konstruktoren sollten grundsätzlich im **public**-Bereich definiert werden. Im Beispiel der `BankKonto`-Klasse könnte man etwa die Methode `eroeffnen` auch als Konstruktor realisieren:

```
class BankKonto {
  public:
    BankKonto( int num, double betrag ) {
      kontonr = num;
      kontostand = betrag;
    }
  // ...
  private:
    int kontonr;
    double kontostand;
};
```

Ist der Konstruktor der Klasse `BankKonto` wie oben definiert, so kann beispielsweise innerhalb der `main`-Funktion mittels

```
BankKonto konto(123456,72.15);
```

eine entsprechend initialisierte Instanz dieser Klasse erzeugt werden. Analog zu den Konstruktoren einer Klasse existieren auch sogenannte *Destruktoren*, die immer dann automatisch aufgerufen werden wenn der Wirkungsbereich eines Objekts verlassen wird, also beispielsweise wenn die Funktion, in der das Objekt instanziiert wurde, durch eine **return**-Anweisung verlassen wird. Der Bezeichner des Destruktors entspricht immer dem Bezeichner der Klasse mit vorangestellter Tilde, in unserem obigen Beispiel lautet der Bezeichner des Destruktors also `~BankKonto()`.

Die meisten in der Praxis auftretenden Konstruktoren dienen lediglich der Initialisierung der Objektattribute. C++ bietet dazu in Form von *Initialisierungslisten* eine spezielle Syntax zur Vereinfachung des Codes für Konstruktoren, ähnlich den Initialisierungslisten für Arrays. Dabei werden die zu initialisierenden Attribute gefolgt von ihren Initialisierungswerten (in

Klammern) durch Kommas getrennt noch vor dem Anweisungsblock des Konstruktors angegeben, wobei zwischen dem Bezeichner des Konstruktors und der Initialisierungsliste ein Doppelpunkt steht:

```
KonstruktorName( typ1 val1, typ2 val2,... ) : attr1(val1), attr2(val2),...
{
  anweisungsblock;
}
```

Den obigen Konstruktor unserer Beispielklasse `BankKonnto` könnten wir mit dieser einfacheren Syntax äquivalent also folgendermaßen definieren:

```
BankKonto( int num, double betrag ) : kontonr(num), kontostand(betrag) {}
```

Der Anweisungsblock ist in diesem Beispiel leer.

Definition der Methoden außerhalb der Klassendeklaration. Ähnlich wie bei der separaten Definition und Deklaration von Funktionen kann auch die Definition und Deklaration von Klassenmethoden voneinander getrennt werden, was gerade beim Vorhandensein von vielen umfangreichen Methoden die Übersichtlichkeit des Quelltexts erhöht. Die Deklaration der Methoden erfolgt dabei im Rumpf der Klasse analog der bereits bekannten Deklaration einer Funktion, nämlich durch Angabe eines Rückgabetyps, Bezeichners sowie der Argumenttypen. Die Definition der Methode kann dann außerhalb der Klassendeklaration erfolgen, wobei der Klassenname gefolgt vom *Scope-Operator* `::` dem Bezeichner der Methode vorangestellt werden muss. Dies gilt in gleicher Weise für Konstruktoren und Destruktoren, auch bei der Verwendung einer Initialisierungsliste im Konstruktor.

Zur Veranschaulichung betrachten wir ein zum bereits bekannten Beispiel mit der Klasse `BankKonto` äquivalentes Programm, in dem die Definition der Methoden nun von den Deklarationen getrennt ist:

```
 1 #include <iostream>
 2 class BankKonto {
 3   public:
 4     BankKonto( int, double );
 5     double abfrage();
 6     void einzahlen( double );
 7     void auszahlen( double );
 8   private:
 9     int kontonr;
10     double kontostand;
11 };
12
13 int main() {
14   BankKonto konto(123456, 72.15);
15   std::cout << konto.abfrage() << std::endl;
16   return 0;
17 }
18
```

```
19 BankKonto::BankKonto( int num, double betrag )
20   : kontonr(num), kontostand(betrag) {
21 }
22 double BankKonto::abfrage() {
23   return kontostand;
24 }
25 void BankKonto::einzahlen( double betrag ) {
26   kontostand += betrag;
27 }
28 void BankKonto::auszahlen( double betrag ) {
29   kontostand -= betrag;
30 }
```

Die Definition der Methoden der Klasse `BankKonto` erfolgt in diesem Beispiel ab Zeile 19 separat von der Deklaration der Klasse ab Zeile 2, wobei im Konstruktor eine Initialisierungsliste verwendet wird.

Zeiger auf Klassen. Auch für Objekttypen können entsprechende Zeigervariablen deklariert und verwendet werden. Die Syntax dazu ist die gleiche, wie bei der Arbeit mit Zeigern auf Variablen eines fundamentalen Typs. Setzen wir beispielsweise die aus dem obigen Beispiel bekannte Definition der Klasse `BankKonto` voraus, so wird durch

```
BankKonto konto( 123456, 72.15 );
BankKonto* p_konto = &konto;
```

eine Zeigervariable `p_konto` deklariert und dieser die Speicheradresse der Instanz `konto` zugewiesen. Zum Zugriff auf die Attribute beziehungsweise Methoden der Klasse über eine Zeigervariable kann man eine Kombination des Dereferenzierungsoperators $*$ mit dem Punktoperator verwenden, wobei Klammern gesetzt werden müssen, da die Bindungsstärke des Punktoperators höher ist als diejenige des Dereferenzierungsoperators. Alternativ kann man den *Pfeiloperator* `->` verwenden, um direkten Zugriff auf die Attribute und Methoden der Klasse über einen Zeiger zu erhalten. Für die wie oben definierte Zeigervariable `p_konto` sind beispielsweise die Anweisungen

```
std::cout << (*p_konto).abfrage() << std::endl;
std::cout << p_konto->abfrage() << std::endl;
```

äquivalent und bewirken jeweils den Aufruf der Methode `abfrage()` für das Objekt `konto` sowie die Ausgabe des Rückgabewerts.

7.3. Überladen von Funktionen und Operatoren

In C++ können verschiedene Funktionen mit dem gleichen Bezeichner definiert werden, solange sich die Parameterypenliste der Funktionen unterscheiden, das heißt eine unterschiedliche Anzahl an Parametern enthalten oder mindestens an einer Stelle verschieden Typen enthalten. Dieses Vorgehen bezeichnet man als *Überladen* einer Funktion. Beispielsweise werden durch

```
float min( float x, float y ) {
  return x<y ? x : y;
}

int min( int j, int k ) {
  return  j<k ? j:k;
}

int min( int j, int k, int l ) {
  int m = j<k ? j : k;
  return m<l ? m : l;
}
```

drei Varianten einer überladenen Funktion `min` definiert, welche in Abhängigkeit der Parameter, mit denen sie aufgerufen wird, entweder das Minimum aus zwei Gleitkommazahlen oder das aus zwei beziehungsweise drei ganzen Zahlen zurückgibt. Die Bestimmung des Rückgabewerts erfolgt hierbei jeweils mithilfe des *Operators für bedingte Ausdrücke*. Dieser folgt der Syntax

```
bedingung ? ausdruck1 : ausdruck2
```

und der Wert dieses Ausdrucks entpricht dem von `ausdruck1`, falls `bedingung` den Wahrheitswert **`true`** hat und anderenfalls dem von `ausdruck2`.

Zusätzlich zum Überladen von Funktionen bietet C++ die Möglichkeit des Überladens auch für die meisten Operatoren. Beispielsweise ist der Divisionsoperator standardmäßig überladen: je nachdem ob er auf zwei Gleitkommatypen oder zwei ganze Typen angewendet wird, entspricht der Rückgabewert dem Wert der Gleitkomma- oder Ganzzahldivision. Das Überladen eines Operators macht Sinn, wenn man ihn in intuitiver Weise auf selbst definierte Klassen anwenden möchte. Die Syntax dazu ist an die Syntax einer Funktionsdefinition angelehnt, wobei der zu einem Operator gehörende Funktionname in den meisten Fällen durch **`operator`** gefolgt vom Operatorsymbol gegeben ist. Als Beispiel dazu betrachten wir eine Klasse `Punkt` zum Rechnen mit Punkten $(x, y) \in \mathbb{R}^2$:

```
 1 #include <iostream>
 2
 3 class Punkt {
 4 public:
 5   // leerer Standardkonstruktor
 6   Punkt() {}
 7   // ueberladener Konstruktor: erzeugt Punkt aus zwei Koordinaten
 8   Punkt( double xval, double yval ): x(xval), y(yval) { }
 9   // Zuweisungsoperator: realisiert Zuweisungen
10   Punkt& operator=( const Punkt& p ) {
11     x = p.x;
12     y = p.y;
13     return *this;
14   }
```

```
15    // Summenoperator: addiert Koordinaten zweier Punkte
16    Punkt operator+( const Punkt& p ) {
17      Punkt summe(x+p.x, y+p.y);
18      return summe;
19    }
20    // Ausgabeoperator: gibt Punktkoordinaten in Konsole aus
21    friend std::ostream& operator<<( std::ostream& ostr, const Punkt& p ) {
22      return ostr << "(" << p.x << "," << p.y << ")";
23    }
24  private:
25    double x,y;
26  };
27
28  int main()
29  {
30    Punkt punkt1(1.2,2.1), punkt2(1.5,1.1);
31    std::cout << "punkt1 = " << punkt1 << std::endl;
32    std::cout << "punkt2 = " << punkt2 << std::endl;
33    Punkt summe;
34    summe = punkt1 + punkt2;
35    std::cout << "Summe = " << summe << std::endl;
36    return 0;
37  }
```

Die Definition der Klasse `Punkt` beginnt in diesem Beispiel in Zeile 3. Objekte der Klasse Punkt verfügen über die privaten **`double`**-Attribute `x` und `y` (Zeile 25), welche der x- beziehungsweise y-Koordinate eines Punkts in $\mathbb{R}^2$ entsprechen.

Ein leerer Standardkonstruktor sowie ein Konstruktor mit Initialisierungsliste sind in den Zeilen 6 und 8 definiert. Ab Zeile 10 folgt die Definition des überladenen Zuweisungsoperators, welcher in Zeile 34 für eine Wertzuweisung benutzt wird. Die Zeiger-Variable **`this`** existiert in jeder Klasse und enthält einen Zeiger auf das aufrufende Objekt, das heißt der Rückgabewert des Zuweisungsoperators entspricht dem aufrufenden Objekt. Dieser Rückgabewert muss verwendet werden, wenn auch Zuweisungen der Form `x=y=z=wert;` erlaubt sein sollen.

Ab Zeile 16 folgt die Definition des Summenoperators, welcher ein Objekt der Klasse `Punkt` zurückgibt, dessen Koordinaten gerade der komponentenweisen Summe der Summanden entsprechen.

Ab Zeile 21 erfolgt die Überladung der Ausgabeoperators, dessen Rückgabetyp dem Typ `std::ostream` des Standardausgabe-Streams entspricht. Dies ist hier, ähnlich wie die Rückgabe von **`*this`** durch den Zuweisungsoperator, notwendig, damit mehrere Ausgaben durch mehrfache Verwendung der Ausgabeoperators aneinandergehängt werden können, so, wie es beispielsweise in Zeile 31 durch Anhängen der Ausgabe von `std::endl` an die Ausgabe von `punkt1` geschieht. Da Operatoren, welche von mehreren Parametern abhängen, intern nicht wie Methoden der Klasse behandelt werden, ist in diesem Fall das Schlüsselwort **`friend`** notwendig, welches eine Funktion als *Freund* einer Klasse kennzeichnet und ihr dadurch den

Zugriff auf private Member dieser Klasse erlaubt. Ohne dieses Schlüsselwort würde der Zugriff auf die privaten Attribute `x` und `y` in Zeile 22 zu einem Fehler bei der Übersetzung führen.

Im Hauptprogramm werden schließlich in Zeile 30 zwei Instanzen `punkt1` und `punkt2` der Klasse `Punkt` erzeugt sowie intitialisiert und ihre Koordinaten durch die Zeilen 31 und 32 mithilfe des überladenen Zuweisungsoperators ausgegeben. In Zeile 33 wird ein zunächst leeres `Punkt`-Objekt `summe` erzeugt, welchem in Zeile 34 mithilfe der überladenen Zuweisungs- und Summenoperatoren die Summe der Objekte `punkt1` und `punkt2` zugewiesen wird. Diese Summe wird anschließend in Zeile 35 wieder mithilfe des überladenen Ausgabeoperators ausgegeben.

7.4. Templates

Ein *Template* erlaubt die abstrakte Definition einer Funktions- bzw. Klassenschablone in Abhängigkeit eines zum Definitionszeitpunkt noch nicht näher bekannten generischen Datentyps. Dabei verlässt man sich darauf, dass alle Operationen, die mit diesem generischen Datentyp durchgeführt werden, für diesen bereits in adäquater Weise implementiert sind. Die Definition erfolgt dabei durch Voranstellen des Schlüsselwortes **`template`** gefolgt von einer Liste der verwendeten generischen Typen in spitzen Klammern:

```
template <class T, class S,...>
```

Hierbei kann anstelle von **`class`** auch das Schlüsselwort **`typename`** verwendet werden. In der sich anschließenden Definition dienen die *Template-Parameter* `T`, `S`,... als Platzhalter für die noch nicht spezifizierten Typen. Beim Aufruf eines *Funktionstemplates* beziehungsweise der Instanziierung eines Objekts eines *Klassentemplates* muss dann ein konkreter Datentyp in eckigen Klammern an den Bezeichner angehängt werden. Klassentemplates werden manchmal auch als *parametrisierte Typen* bezeichnet. Zusätzlich zu den genersichen Typen kann ein Template auch von konstanten Werten eines bereits vorhanden Typs abhängen. Eine Template-Definiton mit einem generischen Typ `T` und einem noch nicht bekannten **`int`**-Wert `n` wird beispielsweise durch

```
template <class T, int n>
```

eingeleitet.

Im folgenden Quelltext ist beispielsweise ab Zeile 3 ein Funktionstemplate implementiert, welches die Bestimmung des Maximums zweier Werte realisiert, unabhängig davon, ob diese etwa als **`int`**, **`double`** oder **`char`** vorliegen.

```
1 #include <iostream>
2
3 template <class T>
4 T maximum( T x, T y ) {
5   if (x < y) return y;
6   else return x;
7 }
8
```

```
 9  int main()
10  {
11    int i = 3, j = 4;
12    double x = 3.5, y = 4.5;
13    char c1 = 'd', c2 = 'v';
14    std::cout << maximum<int>(i,j) << std::endl;    // Ausgabe 4
15    std::cout << maximum<double>(x,y) << std::endl; // Ausgabe 4.5
16    std::cout << maximum<char>(c1,c2) << std::endl; // Ausgabe v
17    return 0;
18  }
```

Ab Zeile 14 folgen drei Aufrufe dieser Funktion, je einmal mit Werten vom Typ **int**, **double** und **char**. Da in diesem einfachen Beispiel der Typ des Template-Parameters `T` schon durch die Parameterliste der Funktion festgelegt ist, kann in den Funktionsaufrufen tatsächlich sogar auf die Typangabe verzichtet werden, das heißt die kürzeren Aufrufe `maximum(i,j)`, `maximum(x,y)` sowie `maximum(c1,c2)` würden hier ebenfalls zum gewünschten Ergebnis führen.

Hinweis: Funktionstemplates stellen in gewisser Weise eine Verallgemeinerung des Überladens von Funktionen dar, falls alle Varianten einer überladenen Funktion die gleichen Anweisungen ausführen. Im obigen Beispiel hätten wir alternativ für jeden vorkommenden Datentyp eine eigene Version der Funktion `maximum()` schreiben können.

Ein Beispiel für eine komplexes Klassentemplate ist die Klasse `std::vector` aus der Standardbibliothek `<vector>`, welche wir bereits in einem früheren Kapitel kennengelernt haben. Nach dem Einbinden dieser Standardbibliothek wird ein Vektorobjekt der Länge 10, dessen Einträge ganze Zahlen und mit 1 initialisiert sind, durch die Anweisung

```
std::vector<int> vek(10, 1);
```

instanziiert. Diese Anweisung bewirkt nichts anderes als den Aufruf des entsprechenden Konstruktors der Klasse `std::vector`. Beispiele für Methoden dieser Klasse sind etwa die Funktionen `push_back()`, `size()` oder auch der Zugriff auf Einträge mittels des überladenen Zugriffoperators `[]`. Die abstrakte Definition der Klasse als Template erlaubt deren komfortable Verwendung, unabhängig davon ob es sich bei den Einträgen des Vektors um ganze Zahlen, Gleitkommazahlen, Zeichenketten oder Instanzen einer benutzerdefinierten Klasse handelt.

7.5. Ausblick

Wir haben bis hier hin einen ganze Reihe von Konzepten der Programmiersprache C++ kennengelernt. Die Verinnerlichung der in diesem Skript vorgestellten Programmiertechniken versetzt einen in die Lage, viele praktische Probleme effizient mit einem Computer zu lösen. Die Minimalbeispiele aus den verschiedenen Kapiteln sollen als Bausteine dienen, welche zur Lösung komplexer Aufgaben beliebig miteinander kombiniert werden können.

Die Sprache C++ ist mittlerweile sehr umfangreich, sodass ein vollständige Behandlung aller Konzepte den Rahmen dieses Skripts sprengen würde. Das Verständnis der grundlegenden

Mechanismen ermöglicht jedoch ein zügiges Erlernen neuer Konzepte. In diesem Zusammenhang seien etwa das Konzept der *Vererbung* sowie der Umgang mit *Iteratoren* als Beispiel genannt.

Ebenso lassen sich fundamentale Denkweisen, wie sie beispielsweise zur Formulierung eines Algorithmus oder zur Planung eines Programmaufbaus nötig sind, leicht auf andere Programmiersprachen übertragen. Viele der vorgestellten Ideen finden sich zumindest in Teilen in anderen Programmiersprachen wieder. Die neben C++ sehr häufig verwendeten Programmiersprachen Python und Java enthalten beispielsweise ebenfalls das Prinzip der Objektorientierung, sodass diese sich zwar im Syntax, nicht jedoch in den zugrunde liegenden Konzepten unterscheiden.

KAPITEL 8

Aspekte der IT-Kommunikation

8.1. Aufbau und Organisation des Internets

Das Internet stellt die Basis der digitalen Welt und der damit verbundenen Kommunikation in der Informationstechnologie dar. Seine Aufgabe ist der Transport von Daten und diesem Zweck dienen eine technische Infrastruktur, die unter anderem aus Glasfaserkabeln, Routern und Netzwerkservern besteht, sowie Konzepte der Datenübermittlung. Der Austausch von Daten funktioniert dabei unabhängig von vorhandenen Geräten und (bisher) der Art der Daten. Die technische Infrastruktur setzt sich aus zahlreichen Einzelnetzwerken wie denen von Internet-Providern sowie Firmen und Universitäten zusammen, die jeweils mit Knotenpunkten verbunden sind. Von diesen Knotenpunkten existieren weltweit ca. 340, die untereinander vernetzt sind. Ihre Vernetzung garantiert, dass zwischen zwei Knotenpunkten mindestens zwei unabhängige Verbindungen existieren, und so eine hohe Ausfallsicherheit erreicht wird.

Der Austausch von Daten erfolgt über ein Internet-Protokoll (IP), welches ein plattform- und anwendungsunabhängiges Datenformat definiert. Informationen wie E-Mails, Inhalte von Internetseiten oder Videokonferenzen werden dabei in kleinere Datenpakete zerlegt und mittels eindeutiger IP-Adressen versendet. Die Datenpakete können bis zu 65.000 Byte groß sein und bestehen aus einem Kopfbereich, dem sogenannten *Header*, mit Absende- und Ziel-IP-Adresse und einem Nutzdatenbereich. Mittels *Routern* werden die Einzelpakete im Internet verschickt. Sie können dabei unterschiedliche Wege nehmen und kommen möglicherweise unsortiert beim Empfänger an. Durch die Informationen im Header können sie jedoch eindeutig zusammengefügt werden.

Beispiel 8.1. Wir betrachten den Aufruf einer Internetseite durch ein internetfähiges Gerät wie ein Smartphone oder ein Laptop. Das Gerät erhält vom Modem, das auch als Router fungiert, eines lokalen Netzwerks (oder LAN für *local area network*) eine interne IP-Adresse.

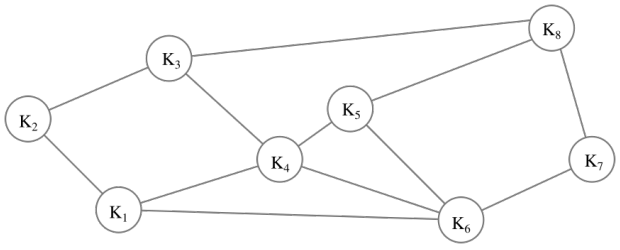

Abbildung 8.1. Schematische Darstellung der Vernetzung der Knotenpunkte des Internets.

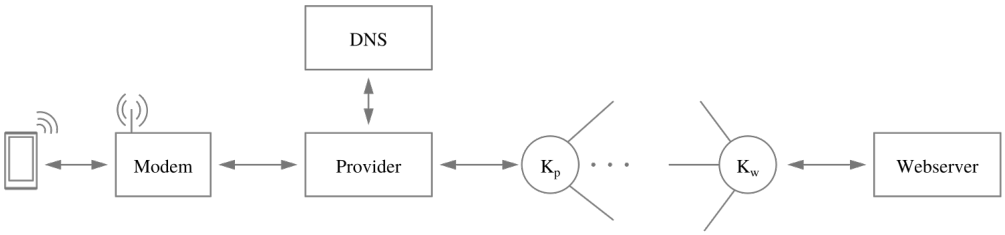

Abbildung 8.2. Kommunikationspunkte beim Aufruf einer Internetseite durch ein Endgerät über Knotenpunkte K_p und K_w des Internets und Ermittlung der IP-Adresse über einen Domain Name Server (DNS).

Beim Aufruf einer Internetseite wird die Anfrage zusammen mit der internen IP-Adresse an das Modem geschickt, welches über eine Schnittstelle (auch als Proxy bezeichnet) eine Verbindung zum *Internet Service Provider* (ISP) herstellt. Das Modem hat eine vom Provider zugeteilte und unter Vorratsdatenspeicherung temporär registrierte dynamische IP-Adresse, während der Provider eine statische besitzt. Der Provider ermittelt über einen *Domain Name Server* (DNS) die IP-Adresse der angewählten Seite. Anschließend wird die Anfrage an den Server der Seite verschickt, welcher daraufhin die gewünschten Daten an das anfragende Gerät in Form kleiner Datenpakete versendet. Mit einer Kapselung gelangen die Pakete über den Provider und das Modem zum Endgerät, welches die Anfrage abgesendet hat.

Bemerkungen 8.2. (i) Aufgrund von Weiterentwicklungen der Lichtsignaltechnologie werden nur ca. 3% der Kapazitäten der verfügbaren Glasfaserkabel genutzt.
(ii) In Deutschland verursacht der Stromverbrauch des Internets durch Endgeräte und Knotenpunkte etwa 3% des Gesamtstromverbrauchs. Weltweit ist die Kohlendioxid-Produktion durch das Internet vergleichbar mit der des gesamten internationalen Flugverkehrs.
(iii) Das Gesamtaufkommen an versendeten Daten im Internet beträgt pro Tag ca. 2 Exabyte beziehungsweise $2 \cdot 10^9$ Gigabyte. Die Hälfte davon entfällt auf das Abrufen von Videos.

8.2. IT-Sicherheit

Unter Sicherheit von Anwendungen in der Informationstechnologie wird der geeignete Umgang mit Daten verstanden, der sicherstellt, dass Risiken wie wirtschaftliche Schäden oder Bedrohungen vermieden werden. Insbesondere sind dabei Datensicherheit, -sicherung und -schutz zu beachten, das heißt die Vermeidung von Manipulation, Verlust und Verletzung der Vertraulichkeit. Entsprechende Ziele werden mit den Begriffen Vertraulichkeit, Integrität und Verfügbarkeit zusammengefasst:

- Das Ziel der *Vertraulichkeit* stellt sicher, dass zu jedem Zeitpunkt nur autorisierte Personen Zugriff auf die verarbeiteten Daten haben.
- Das Ziel der *Integrität* fordert, dass Änderungen an Daten stets nachvollziehbar sind.
- Das Ziel der *Verfügbarkeit* garantiert, dass auf Daten stets innerhalb eines vorgegebenen Zeitraums zugegriffen werden kann.

Die konkrete Formulierung und Bewertung anwendungsabhängiger Schutzziele erfolgt im Rahmen einer Risikoanalyse. Ein IT-System gilt als sicher, wenn der Aufwand eines Eindringens höher ist als der daraus erzielte Nutzen und die Gefahr von Verlusten durch technische Fehler relativ zum Aufwand der Wiederherstellung der Daten gering ist. Eine absolute Sicherheit ist in den meisten Fällen nicht gerechtfertigt, da dies die Arbeitsfähigkeit stark einschränkt und unangemessen hohe Kosten verursacht. Mögliche Angriffe auf ein System erfolgen meist durch Viren, Identitätsdiebstahl und physischen Einbruch. Als Angriffe gelten auch solche, die durch höhere Gewalt wie Blitzeinschlag verursacht werden. Maßnahmen zur Gewährung der Ziele der IT-Sicherheit sind die räumliche Trennung von Daten, das Einführen von Zugriffskontrollen, die Verwendung von Nutzungsrechten, die regelmäßige Aktualisierung verwendeter Software, die Erstellung von Sicherheitskopien sowie die Verwendung von Antiviren-Software und Firewalls. Gefährdungen der IT-Sicherheit können auch durch Programmierfehler verursacht werden. Gesetzlich untersagt ist jegliche Manipulation fremder Daten sowie das Ausspähen geschützter, das heißt verschlüsselter Daten.

8.3. Datenverschlüsselung

Ein wesentlicher Bestandteil der IT-Sicherheit ist die Verschlüsselung von Daten. Die Entwicklung und Bewertung entsprechender Verfahren wird auch als *Kryptografie* bezeichnet. Entsprechende Ideen existieren seit Jahrtausenden und basierten lange Zeit auf der vertraulichen Vereinbarung eines gemeinsamen geheimen Schlüssels. Dazu war jedes Mal ein vertrauenswürdiger Kurier oder ein persönliches Treffen erforderlich. Vor einigen Jahrzehnten haben sich durch mathematische Theorien und die Verfügbarkeit leistungsfähiger Computer Möglichkeiten ergeben, die diese Schwachstelle vermeiden. Der geheime Schlüssel wird durch offene Kommunikation zwischen Absender und Empfänger mittels sogenannter *Public-Key-Verfahren* generiert. Zentraler Bestandteil dieser Verfahren ist die praktische Irreversibilität gewisser mathematischer Operationen wie dem Multiplizieren von Primzahlen. Wir folgen in diesem Abschnitt den Ausführungen des Buchs *Mathematik sehen und verstehen* von D. Haftendorn (Springer 2016).

Beispiele 8.3. (i) Bei der monoalphabetischen Verschlüsselung werden die Buchstaben des Alphabets zyklisch um eine feste, durch den vereinbarten Schlüssel festgelegte, Anzahl von Positionen verschoben. Mit 25 Tests oder effizienter durch Verwendung von Buchstabenhäufigkeiten können entsprechend verschlüsselte Texte jedoch lesbar gemacht werden. Etwas schwieriger ist dies bei polyalphabetischen Verschlüsselungen, die auf einem Schlüsselwort und einer Buchstabenaustauschtabelle basieren.
(ii) Sehr sicher ist das Verschlüsseln eines Texts, wenn dieser zunächst in eine Zahl beziehungsweise Ziffernfolge übersetzt wird, beispielsweise durch Identifikation von Buchstaben mit zweistelligen Zahlen und Hintereinanderfügen dieser Zahlen. Ist $m \in \mathbb{N}$ die zu übertragende Nachricht und $s \in \mathbb{N}$ ein Schlüssel selber Länge, so kann die verschlüsselte Nachricht $c \in \mathbb{N}$ selber Länge definiert werden durch die Ziffern

$$c_i = (m_i + s_i) \bmod 10.$$

Der Schlüssel kann beispielsweise als Teilfolge der Zahl π gewählt werden. Ohne den Schlüssel ist die Nachricht nicht zu entschlüsseln, da sich jede beliebige Zahl m' selber Länge mit einem geeigneten Schlüssel s' aus c erzeugen lässt.

Moderne Verfahren der Kryptografie nutzen Methoden der Mathematik, um das Problem der Vereinbarung eines gemeinsamen geheimen Schlüssels zu vermeiden. Wesentlich ist dabei die Verwendung von Primzahlen sowie der Primzahlfaktorisierung beliebiger Zahlen, das heißt die eindeutige Darstellung einer beliebigen Zahl $z \in \mathbb{N}$ als Produkt von Primzahlen $p_1, p_2, \ldots, p_k$

$$z = p_1^{\ell_1} p_2^{\ell_2} \ldots p_k^{\ell_k}.$$

Zwar lässt sich die bloße Existenz dieser Faktorisierung rigoros nachweisen, jedoch ist die praktische Bestimmung der Faktoren ein NP-schwieriges Problem, das heißt der Aufwand bekannter Verfahren zur Bestimmung der Faktoren wächst exponentiell mit der Anzahl der Stellen von z.

Beispiel 8.4. Zur Bestimmung der Primzahlfaktorisierung einer Zahl $z \in \mathbb{N}$ muss jede Primzahl $p \leq \sqrt{z}$ als Faktor getestet werden. Davon gibt es nach einer Formel von Gauß etwa $\sqrt{z}/\ln(\sqrt{z})$ viele. Besitzt z beispielsweise 300 Stellen, so ergeben sich ca. $3 \cdot 10^{147}$ Tests. Selbst bei Einsatz aller verfügbarer Rechner ist dies nur in Milliarden von Jahren realisierbar.

Rechnen modulo n. Zwei ganze Zahlen x und y werden als *gleich modulo* n bezeichnet, wenn ihre Differenz ein Vielfaches von n ist, das heißt wenn eine ganze Zahl q existiert, sodass

$$x = y + qn$$

gilt. In diesem Fall schreibt man $x \equiv_n y$; ist x in einer Gleichung $x \equiv_n y$ wählbar, so wählen wir x mit der Eigenschaft $0 \leq x \leq n-1$. Beim Multiplizieren können Faktoren durch gleiche Zahlen modulo n ersetzt werden, das heißt gilt $a \equiv_n \widetilde{a}$, so auch $ab \equiv_n \widetilde{a}b$. Das Potenzieren modulo n ist ein Spezialfall des Multiplizierens.

Beispiele 8.5. (i) Es gilt $5 \equiv_3 2$ und $4 \equiv_4 0$.
(ii) Es gilt $5 \cdot 5 \equiv_3 2 \cdot 5 \equiv_3 2 \cdot 2 \equiv_3 1$.
(iii) Es gilt $5^4 \equiv_{14} 25 \cdot 25 \equiv_{14} 11 \cdot 11 \equiv_{14} 9$.

Zwei Zahlen a, b heißen *invers modulo* n, wenn $ab \equiv_n 1$ gilt.

Beispiel 8.6. Es gilt $3 \cdot 7 \equiv_{10} 1$.

Ohne Kenntnis der Zahl n ist die Bestimmung eines inversen Elements im Allgemeinen nicht möglich.

Bemerkungen 8.7. (i) Als Folgerung des Eulerschen Satzes ergibt sich der kleine Satz von Fermat, der besagt, dass für jede Primzahl p und jede Zahl $1 \leq a \leq p-1$ gilt

$$a^{p-1} \equiv_p 1.$$

(ii) Die *Powermod-Methode* eignet sich zur effizienten Berechnung großer Potenzen modulo n. Dabei wird der Exponent in Binärdarstellung geschrieben und eine geeignete Klammerung verwendet, so dass nur wenige Quadrate berechnet werden müssen:

$$a^\ell \equiv_n \left(\left(\ldots \left(a^{b_k} \right)_n^2 \ldots a^{b_2} \right)_n^2 a^{b_1} \right)_n^2 a^{b_0},$$

sofern $\ell = b_k 2^k + \ldots b_1 2 + b_0$ mit $b_i \in \{0, 1\}$ und mit der Notation $(x)_n = x \bmod n$.

Eine Erweiterung des Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen erlaubt für gegebene Zahlen a, b die Bestimmung zweier Zahlen s, t,

sodass gilt

$$1 = sa + tb$$

und insbesondere $1 \equiv_a tb$, das heißt t ist invers zu b modulo a.

Kryptografische Verfahren. Das Diffie-Hellman-Protokoll ist ein symmetrisches Verfahren zur Festlegung eines gemeinsamen geheimen Schlüssels, der mittels offener Kommunikation zwischen den beteiligten Parteien, die im Folgenden mit Anton und Berta bezeichnet werden, festlegt.

Algorithmus 8.8 (Diffie–Hellman-Protokoll)**.**
(1) Anton und Berta wählen offen eine Primzahl p und eine Zahl $1 < g < p$.
(2) Anton wählt eine geheime Zahl $1 < s < p$, berechnet $a \equiv_p g^s$ und teilt Berta die Zahl a mit.
(3) Berta wählt eine geheime Zahl $1 < r < p$, berechnet $b \equiv_p g^r$ und teilt Anton die Zahl b mit.
(4) Anton berechnet den geheimen Schlüssel $k \equiv_p b^s$.
(5) Berta berechnet den geheimen Schlüssel $k \equiv_p a^r$.

Anton und Berta sind nach Ausführung des Protokolls im Besitz desselben Schlüssels, denn es gilt

$$k_{\text{Anton}} \equiv_p b^s \equiv_p (g^r)^s \equiv_p (g^s)^r \equiv (a)^r \equiv_p k_{\text{Berta}}.$$

Ein Angreifer kann die Zahlen p, g und a, b abhören. Um jedoch an den Schlüssel k zu gelangen, muss er eine der Gleichungen

$$g^s \equiv_p a, \quad g^r \equiv_p b$$

nach s oder r lösen, was für große Zahlen jedoch praktisch unmöglich ist.

Beispiel 8.9. Die Gleichung $7^s \equiv_{23} 14$ kann durch Ausprobieren von $s = 1, 2, \ldots, 22$ gelöst werden und liefert die Lösung $s = 15$. Bei einer Primzahl p mit ℓ Stellen sind jedoch etwa 10^ℓ viele Zahlen zu testen. In der Praxis werden Primzahlen mit 300 Stellen verwendet.

Die einfach auszuwertende diskrete Exponentialabbildung $s \mapsto g^s \bmod p$ ist nach Sätzen der Algebra über zyklische Gruppen bijektiv, sofern g eine Primitivwurzel der Restklassengruppe modulo p ist. In diesem Fall heißt die Umkehrabbildung diskreter Logarithmus. Seine unregelmäßige Verteilung ist in Abbildung 8.3 illustriert und veranschaulicht, wieso die Berechnung des diskreten Logarithmus ein nach heutiger Kenntnis NP-schwieriges Problem ist. Dieser Sachverhalt und die Kommutativität der Potenz, das heißt die Identität

$$(g^s)^r = (g^r)^s,$$

sind die Basis des Diffie-Hellman-Protokolls. Eine effizente algorithmische Bestimmung von Primitivwurzeln ist zwar nicht bekannt, jedoch können die offenen Paare (p, g) je nach erforderlicher Schlüssellänge aus Tabellen entnommen werden. Die diskrete Exponentialabbildung ist ein Beispiel einer in der Kryptographie als *Einwegfunktion* bezeichnete Abbildung.

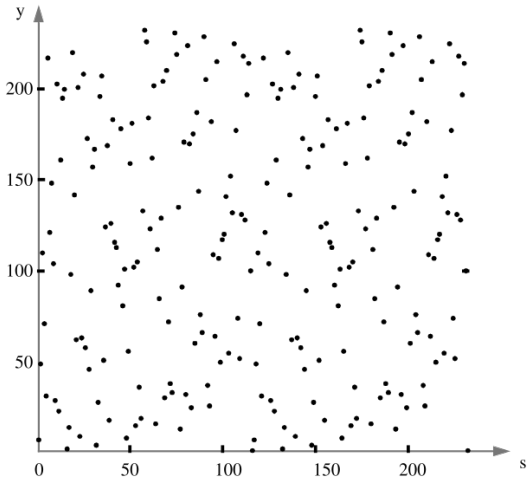

Abbildung 8.3. Unregelmäßige Verteilung der Werte der diskreten Exponentialfunktion $s \mapsto y = b^s \bmod p$, $1 \leq s \leq p-1$, mit $p = 233$ und $b = 7$.

Um gewisse Schwächen des Diffie-Hellman-Verfahren zu vermeiden, entwickelten Rivest, Shamir und Adleman das nach ihnen benannte RSA-Verfahren, welchem die praktische Irreversibilität der Primzahlfaktorisierung zugrunde liegt. Hierbei handelt es sich um ein asymmetrisches Verfahren, das heißt die Verschlüsselung erfolgt durch den Versender der Nachricht mit den vom Empfänger öffentlich bereitgestellten Informationen.

Algorithmus 8.10 (RSA-Protokoll, Schlüsselerzeugung)**.**
(1) Anton wählt Primzahlen p, q und bestimmt $n = pq$ sowie den Wert $\varphi(n) = (p-1)(q-1)$ der Eulerschen φ-Funktion.
(2) Anton wählt $1 < e < \varphi$ mit $\mathrm{ggT}(e, \varphi) = 1$ und bestimmt eine Zahl d mit $1 \leq d \leq \varphi$ und $de \equiv_\varphi 1$, die geheim gehalten wird. Die Zahlen p, q, φ werden gelöscht.
(3) Anton veröffentlicht den Schlüssel (n, e).

Will Berta eine verschlüsselte Nachricht m an Anton schicken, geht sie folgendermaßen vor.

Algorithmus 8.11 (RSA-Protokoll, Anwendung)**.**
(1) Berta lädt Antons öffentlichen Schlüssel (n, e).
(2) Berta bestimmt $c \equiv_n m^e$ und schickt c an Anton.
(3) Anton erhält c und bestimmt $m' \equiv_n c^d$.

Dass Anton die richtige Nachricht erhält, dass also $m = m'$ gilt, folgt aus dem Satz von Euler beziehungsweise dem kleinen Satz von Fermat: es gilt

$$m^{k\varphi(n)} \equiv_p m^{k(p-1)(q-1)} \equiv_p \left(m^{(p-1)}\right)^{k(q-1)} \equiv_p 1,$$
$$m^{k\varphi(n)} \equiv_q m^{k(p-1)(q-1)} \equiv_q \left(m^{(q-1)}\right)^{k(p-1)} \equiv_q 1,$$

und aus diesen beiden Gleichungen ergibt sich unmittelbar

$$m^{k\varphi(n)} \equiv_{pq} 1,$$

woraus unter Verwendung von $n = pq$ folgt, dass

$$m' \equiv_n c^d \equiv_n (m^e)^d \equiv_n m^{1+k\varphi(n)} \equiv_n m$$

gilt. Ein Angreifer müsste zur Entschlüsselung von c die Zahl d bestimmen, was jedoch ohne Kenntnis von $\varphi(n)$ beziehungsweise p und q nicht möglich ist. Mit dem RSA-Verfahren lassen sich auch digitale Signaturen erzeugen, die zum Zwecke der Datenintegrität verwendet werden können. Wenn Anton eine Nachricht veröffentlicht, fügt er seiner Nachricht die Signatur $\mathrm{sig} \equiv_n m^d$ hinzu. Ein Leser der Nachricht kann dann durch Berechnung von $\mathrm{test} \equiv_n \mathrm{sig}^e$ prüfen, ob tatsächlich $\mathrm{test} \equiv_n m$ gilt und er der Nachricht vertrauen kann. Gelegentlich wird m für diesen Zweck mittels einer Hash-Funktion H komprimiert, das heißt man betrachtet $\mathrm{sig} \equiv_n H(m)^d$ und überprüft, ob $\mathrm{test} \equiv_n \mathrm{sig}^e \equiv_n H(m)$ gilt. Durch Einbeziehung einer dritten Stelle kann zusätzlich eine Zertifizierung eines Schlüssels erfolgen.