C lernen mit BoS – Board of Symbols

Stephan Euler
TH Mittelhessen
Version 0.5 Dezember 2015

31. Dezember 2015

Inhaltsverzeichnis

Kapitel 1

Vorwort

1.0.1 Hinweise zum Lesen

Das Skript ist gedacht als Einführung in C für Programmieranfänger. Daher liegt der Schwerpunkt auf den wichtigsten Sprachelementen und einfachen, leicht nachvollziehbaren Beispielen. Daneben enthält es allerdings auch Abschnitte mit spezielleren Themen. Diese Abschnitte dienen eher als Referenz und können beim ersten Lesen gut übersprungen werden. Als Markierung dient das △-Zeichen. Enthält eine Überschrift dieses Zeichen, so dient der Abschnitt mehr der Vollständigkeit als dem Verständnis beim Einstieg. Mehrere △-Zeichenen symbolisieren einen noch stärkeren Referenz-Charakter.

Im Laufe der Zeit entstanden mehrere Versionen der Sprache C. Dieser Skript bezieht sich im Wesentlichen auf die Version C90.

Kapitel 2

Einführung

Frage 2.1. Wozu brauchen wir dieses BoS?

C ist eine alte Programmiersprache – ganz bestimmt älter als die meisten Leser. Der Ursprung liegt in den Jahren 1969 bis 1973 [Rit93]. Zu dieser Zeit waren grafische Benutzeroberflächen noch kein Thema. Daher fehlt in den C-Standard-Bibliotheken die grafische Ein-/Ausgabe. Natürlich ist es möglich, mit C grafische Ein- und Ausgabe zu realiseren. Aber es ist nicht gerade das Thema, mit dem Einsteiger anfangen sollten. Andererseits ist die Ausgabe in einem Textfenster für viele nicht besonders motivierend. Hier soll BoS helfen. Mit seiner Hilfe kann man aus einem C-Programm heraus leicht grafische Ausgaben erzeugen.

2.1 Voraussetzungen

Die Anwendung ist in Java geschrieben. Zur Ausführung benötigt man die Java-Laufzeitumgebung (*Java Runtime Environment*, kurz JRE), die auf den allermeisten Rechnern bereits vorhanden ist. Java ist weitgehend unabhängig vom Betriebssystem, so dass es hier keine Probleme geben sollte. Um die selbst geschriebenen C-Programme auszuführen, ist außerdem eine Entwicklungsumgebung für C nötig. Dabei gibt es verschiedene Möglichkeiten, wobei allerdings auch das jeweilige Betriebssystem eine Rolle spielt. Informationen zu diesen eher technischen Fragen sind im Moodle-Kurs und auf der Webseite zusammen gestellt.

2.2 Erste Schritte

Die Java-Anwendung befindet sich als Archiv zusammen gepackt in der Datei `jserver.jar`. In den meisten Fällen startet ein Doppelklick auf diese Datei die Anwendung. Alternativ kann man den Befehl

```
java -jar jserver.jar
```

in der Konsole (Eingabeaufforderung) eingeben. Wenn alles richtig eingerichtet ist, sollte ein Fenster wie in Bild 2.1 erscheinen. Wir sehen ein 10×10 großes Spielfeld. Jedes einzelne Feld ist mit einem Kreis bedeckt und in jedem Kreis steht eine Zahl. Dies ist die fortlaufende Nummer, unter dem wir das Feld ansprechen können. Sollten die Nummern nicht erscheinen, so kann man sie über das Menü *Formen* $\hookrightarrow$ *Numerierung Ein/Aus* erscheinen lassen. Die Nummerierung beginnt links unten mit 0 und läuft dann nach rechts und oben bis zu 99.

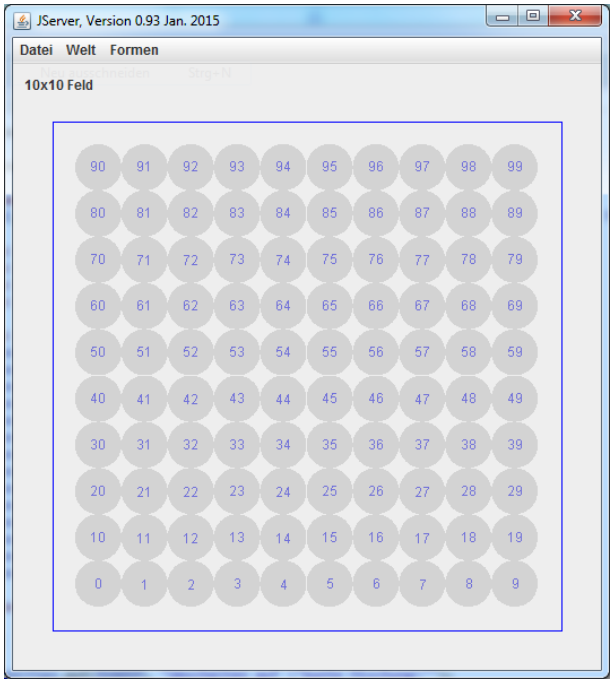


Abbildung 2.1: JServer

Frage 2.2. Warum zählen wir nicht ab 1?

Im täglichen Leben beginnen wir mit der 1. Außer der im Kölner Karneval besungenen *Kaygasse Numero 0* kenne ich zum Beispiel keine Straße, deren Hausnummern mit 0 beginnt. Aber in den Programmiersprachen der C-Familie ist die Zählung ab 0 üblich (*Zero-based numbering*). Dieser Vorgabe folgt die Nummerierung der Felder. Ursprünglich gab es durchaus gute technische Gründe für die 0-Nummerierung. Beim Herunterzählen konnte der Vergleich auf 0 durch einen direkten Prozessorbefehl ausgeführt werden: *jump on zero* oder ähnlich. Hintergründe dazu findet man z. B. im Artikel von Edsger W. Dijkstra [Dij82]. Mittlerweile spielt dies keine Rolle mehr, aber die 0 hat ihre Startposition behauptet.

Über die jeweilige Nummer kann man ein Symbol ansprechen und verändern. Wir beginnen mit einem einfachen Einfärben. Dazu wählen wir zunächst den

Menüpunkt *Welt* ↪ *Fenster für Code Eingabe* und sollten ein Fenster ähnlich wie 2.2 sehen. Das Fenster besteht aus drei Bereichen:

- oben: Eingabe der C-Anweisungen
- Mitte: Steuerelemente
- unten: Meldungen beim Ausführen

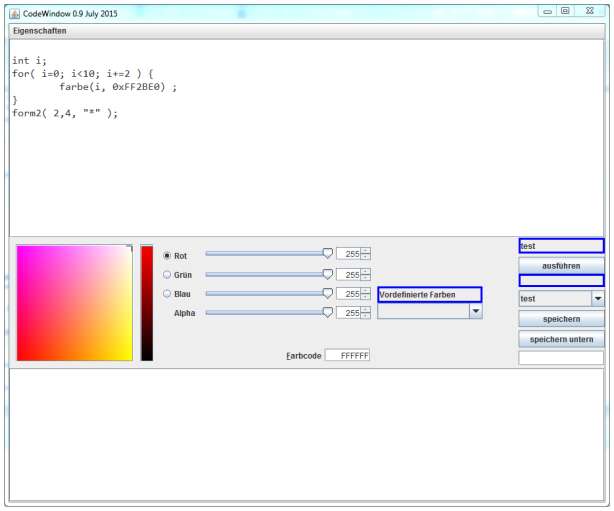


Abbildung 2.2: Eingabe für C-Code

Ins Eingabefenster schreiben wir unsere erste Anweisung:

```
farbe( 12, 0xff );
```

Dies ist ein kleiner Ausschnitt aus einem C-Programm. Wir bezeichnen dies auch als Programmcode.

Dabei ist `farbe` der Name einer Funktion. Funktionen erkennt man an den runden Klammern um die Argumente. Das Konzept kennen wir aus der Mathematik. Als Beispiel steht $sin(\pi/2)$ für die Berechnung der Sinus-Funktion. Die Funktion bekommt einen Wert – $\pi/2$ – und berechnet daraus ein Ergebnis. Wir wissen dabei nicht, wie die Berechnung ausgeführt wird. Aber wir können beispielsweise am Taschenrechner die Funktion ausführen und erhalten das gesuchte Ergebnis. Diesem Grundprinzip – wir übergeben Werte an eine Funktion, diese führt Berechnungen aus und gibt ein Result zurück – folgen auch die Funktionen in C. Wir können Funktionen als Bausteine verwenden, ohne dass wir uns Gedanken über deren innere Abläufe machen müssen. Funktionen in C sind aber nicht das gleiche wie Funktionen in der Mathematik. Es gibt grundlegende Unterschiede, um die wir uns aber bei unserer Programmierung nicht kümmern müssen.

Zurück zu unserer Eingabe. Wir rufen eine Funktion auf und übergeben ihr zwei Werte als Argumente. Mehrere Argumente werden durch Komma getrennt.

In diesem Fall sind wir nur an der ausgeführten Aktion interessiert. Eventuell gibt `farbe` auch ein Ergebnis zurück, aber wir ignorieren es einfach. Abgeschlossen wird die Anweisung durch das Semikolon `;`. Die beiden Argumente geben die Position und die Farbe an. Das erste ist klar, wir wollen das 12. Symbol färben. Beim zweiten Argument – dem Farbwert – ist es etwas komplizierter.

Wir verwenden den RGB-Farbraum, bei dem eine Farbe durch das additive Mischen der drei Grundfarben Rot, Grün und Blau gebildet wird. Für jede der drei Grundfarben wird den Anteil als Zahl zwischen 0 und 255 angegeben. Die Obergrenze 255 ist gerade $2^8 - 1$, so dass wir sie als Binärzahl mit 8 Einsen schreiben würden. Dementsprechend sind für jede Grundfarben 8 Stellen (bits) reserviert. Der Gesamtwert ergibt sich dann durch hintereinander Schreiben der Binärzahlen.

Anstelle der recht umständlichen Binärdarstellung verwendet man eher die Darstellung im 16-er System als Hexadezimalzahl. Dort steht jede Ziffer (0 bis 9 und dann A, B, C, D, E, und F, groß oder klein geschrieben) für 4 bits. Der Anteil jeder Grundfarbe wird dann mit zwei Ziffern angegeben. In C erkennen wir Hexadezimalzahlen am Anfang `0x`. In unserem Beispiel war der Farbwert `0xff`. Das letzte Byte ist gefüllt und demnach haben wir einen maximalen Anteil Blau und keine Anteile in den beiden anderen Grundfarben. Dementsprechend sollten die Darstellung wie in Bild 2.3 erscheinen. Durch die eingeschaltete Nummerierung ist die Farbe etwas blass, so dass man die Schrift noch lesen kann. Die Darstellung als Hexadezimalzahl ist nach etwas Eingewöhnung übersichtlich, wir hätten aber auch mit

```
farbe( 12, 255 );
```

den Wert als Dezimalzahl angeben können. Bei neueren Versionen von C kann man sogar die Binärzahl

```
farbe( 12, 0b11111111 );
```

direkt schreiben.

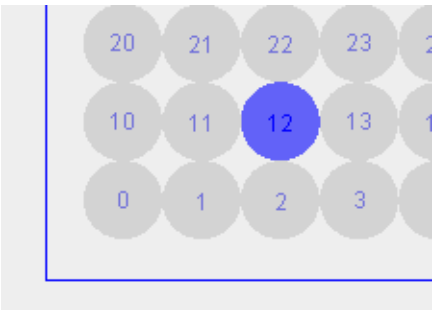


Abbildung 2.3: Erstes Beispiel mit Farbe

Frage 2.3. Muss ich jetzt das RGB-Modell verstehen und Bits und Bytes herum schubsen?

Nein, BoS bietet zwei Hilfen zur Farbeingabe. Zunächst gibt es einen Farbwähler, so wie man ihn aus anderen Anwendungen kennt. Dort kann man entweder über die Schieberegler die Anteile der Grundfarben einstellen oder per Klick in den Farbbereich eine Farbe direkt auswählen. Der zugehörige RGB-Wert wird dann in einem Feld angezeigt, von wo wir ihn mittels Copy-Paste holen und einfügen können (`0x` davor nicht vergessen). Weiterhin gibt es eine Auswahl von vordefinierten Farben. Wählt man eine davon aus, wird der Name direkt an der aktuellen Zeigerposition eingefügt. Wir hätten unser Beispiel also auch als

```
farbe( 12, BLUE );
```

schreiben können.

Frage 2.4. Kann ich eigene Farben definieren?

Ja, das ist sogar recht einfach. Die Farben stehen in einer Datei `colors.h`. Zum Beispiel definiert dort die Zeile

```
#define BLUE  0x0000FF
```

unser Blau. Man kann einfach nach diesem Muster weitere Zeilen einfügen. Wir könnten beispielsweise schreiben

```
#define BLAU  0x0000FF
#define MEINE_FARBE  0x123456
```

um zwei weitere Definitionen anzuhängen. Die Namen sollten nach C-Konventionen mit Großbuchstaben geschrieben werden, allerdings ist das nicht Pflicht.

Frage 2.5. Was passiert wenn ich mich vertippe?

Der eingegeben Text wird zunächst zu einem vollständigen C-Programm ergänzt und in einer Datei (`commandgenerator.c`) gespeichert. Diese Datei wird von einem speziellen Programm analysiert und – vereinfacht gesprochen – in ein ausführbares Programm übersetzt. Unseren – von Menschen für Menschen geschriebenen – Text bezeichnet man daher als Quelltext (Quellcode, source code). Das Ergebnis ist Maschinencode – Code den der Prozessor direkt verstehen und ausführen kann.

Den Vorgang nennt man Kompilierung, den Übersetzer Compiler. Falls bei der Analyse Fehler gefunden werden, bricht der Compiler den Vorgang ab und meldet die gefundenen Fehler. Zum Beispiel bekommen wir bei einem vergessenen Semikolon am Ende der Anweisung die Meldung

```
commandgenerator.c: In function 'main':
commandgenerator.c:54:1: error: expected ';' before '}' token
 }
 ^
compile failed
```

Die Meldungen unterscheiden sich bei verschiedenen Compilern im Detail etwas. Diese stammt von GNU C-Compiler. Der Compiler von Visual Studio meldet den gleichen Fehler in der Form

```
commandgenerator.c(54) : error C2143: Syntaxfehler: Es fehlt ';' vor '}'
```

Die Meldungen sind mehr oder weniger hilfreich. Kommen viele Meldungen so sollte man sich zunächst um den ersten Fehler kümmern. Häufig stammen die weiteren Meldungen von Folgefehlern, die magisch verschwinden sobald der erste Fehler beseitigt ist. Erst wenn das Programm aus Sicht des Compilers fehlerfrei ist, wird ein ausführbares Programm erzeugt. Dieses wird in unserer Umgebung sofort ausgeführt. Der Compiler weiß aber nichts über die BoS Anwendung. Fehler, die die Logik von BoS betreffen, kann er daher nicht finden. So ist aus Sicht des Compilers

```
farbe( 100000, BLUE );
```

vollkommen in Ordnung. Der Compiler hat die Information, dass `farbe` zwei Zahlen als Argumente benötigt. Die Bedeutung der Zahlen und mögliche Einschränkungen im Wertebereich kennt er nicht. Dass es wohl nicht so viele Felder geben wird, muss die Anwendung selbst behandeln. In diesem Beispiel erscheint die Fehlermeldung

```
>>100000 0xff: ERROR - 100000 out of Range
```

Dabei handelt es sich aber nicht um eine Compiler-Meldung, sondern der Fehler wird erst während der Ausführung gemeldet.

Frage 2.6. Okay, die Farben sind damit klar. Was kann ich noch ändern?

Als nächstes ändern wir die Symbole. Dazu gibt es eine Funktion `form(i, s)`. Das erste Argument ist wieder die Feldnummer, das zweite gibt den gewünschten Symbolnamen (eigentlich eher Kurznamen) an. Der Name ist jetzt anders als die Farbe keine Zahl sondern ein kurzer Texte. Texte werden in doppelte Anführungszeichen geschrieben. Betrachten wir die Erweiterung

```
farbe(12, BLUE);
form( 12, "s" );
form( 2, "d" );
form( 11, "*" );
```

mit dem Ergebnis 2.4. Hier werden auf drei Feldern die Kreise durch andere Symbole ersetzt, einem Quadrat, einer Raute und einem Stern. Folgende Symbole sind möglich:

Typ	Kürzel	Details
Kreis	`c`	circle, Standard-Symbol beim Start
Quadrat	`s`	square
Raute	`d`	diamond
Stern	`*`	
Plus	`+`	
Linie	`|`	vertikal
Linie	`-`	horizontal
Linie	`/`	schräg
Linie	`\`	schräg (siehe Hinweis im Text)
Dreieck	`tld`	triangle left down, Ecke links unten
Dreieck	`trd`	triangle right down, Ecke rechts unten
Dreieck	`tlu`	triangle left up, Ecke links oben
Dreieck	`tru`	triangle right down, Ecke rechts unten
zufällig	`r`	random, Symbol wird zufällig gewählt
leer	`none`	Feld ist leer

Der Schrägstrich \ (*Backslash*, Rückstrich) hat in C eine besondere Bedeutung. Er leitet normalerweise ein Sonderzeichen wie `\n` für einen Zeilenzúmbruch ein. Um eine Schrägstrich selbst zu erhalten, muss er geschützt als `\\` eingeben werden. Zwei Kürzel haben eine besondere Wirkung. Bei `r` wird ein zufälliges Symbol ausgewählt während bei `none` ein eventuell vorhandenes Symbol entfernt wird und das Feld leer bleibt. Insbesondere bei diesen beiden Fällen möchte man manchmal alle Symbole gleichzeitig ändern. Dies kann man jederzeit über das Menü *Formen* erreichen. Alternativ steht die Funktion `formen()` zur Verfügung. Beim Aufruf gibt man lediglich das Symbolkürzel an. Als Beispiel füllt

```
formen( "r" );
```

das gesamte Spielfeld zufällig mit Symbolen.

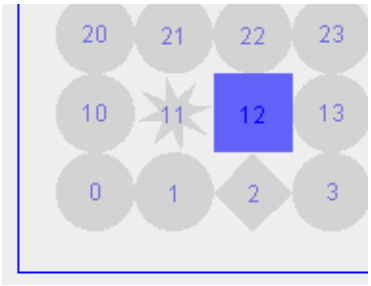
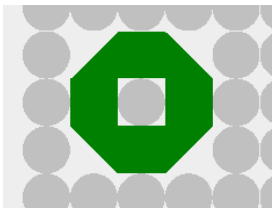


Abbildung 2.4: Wechsel der Symbole

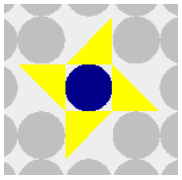
Nachdem wir jetzt Farbe und Symbol jedes Feldes ändern können, ist die Zeit für einige Übungen gekommen.

Übung 2.1. Einfache Muster
Schreiben Sie Anweisungen, um die folgenden Muster zu erzeugen:

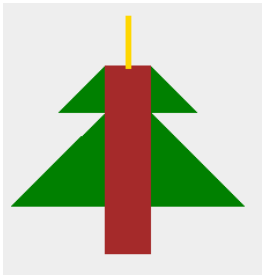
1. Stadion



2. Windrad



3. Baum



Frage 2.7. Kann ich mehrere Symbole auf einem Feld kombinieren, z. B. zwei Dreiecke mit unterschiedlichen Farben?

Nein, man kann sich das wie bei *Schach* oder *Mensch ärger dich nicht* vorstellen. Auf jedem Feld ist nur für ein Symbol Platz. Setzt man ein neues Symbol, so wird das alte vom Brett genommen. Allerdings wäre es schade, wenn jedes Feld nur jeweils eine Farbe hätte. Daher gibt es die Möglichkeit, zumindest eine zweite Farbe als Hintergrund anzugeben. Das ist das gleiche Vorgehen wie bei der Hauptfarbe, die entsprechende Funktion heißt `hintergrund` . Eine kleines Beispiel dazu: der Code-Abschnitt

```
form( 11, "tld" );
farbe(11, BLUE);
hintergrund(11, YELLOW);
form( 13, "*" );
farbe(13, BLUE);
hintergrund(13, YELLOW);
```

erzeugt das Muster in Bild 2.5. Wie das erste Feld zeigt, kann man auf diesem Weg zwei nebeneinander liegende Dreiecke nachbilden.

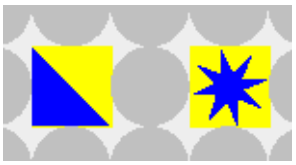


Abbildung 2.5: Zwei Felder mit gelbem Hintergrund

Frage 2.8. Wie kann ich den Code abspeichern?

Die Code-Schnipsel werden in der Datei `codes.xml` gespeichert. Als Format wird XML (*Extensible Markup Language*) verwendet. Dabei werden neben dem eigentlichen Code noch einige Informationen wie Erstellungs- und Änderungsdatum oder Name des Autors oder der Autorin (kann man im Menü Eigenschaften eintragen) abgelegt. Beim Speichern vergibt man einen Namen für einen Code-Schnipsel, über den man ihn später auch wieder laden kann.

Zum Abschluss dieses Kapitels noch ein kurzer Blick auf einige Hilfsfunktionen. Die folgenden Funktionen

Name	Funktionsweise
`groesse(int x, int y)`	neue Größe für das Brett
`flaeche(int f)`	Hintergrundfarbe
`rahmen(int f)`	Rahmenfarbe
`farben(int f)`	Farbe aller Symbole
`loeschen()`	löscht (fast) alle Farb-Einstellungen

sind manchmal nützlich, um Größe und Aussehen einzustellen. Das `int` vor den Argumenten legt fest, dass es sich um ganze Zahlen handeln muss – dazu mehr im nächsten Kapitel. Ein Aufruf der Funktion `loeschen` zu Beginn entfernt eventuell vorhandene Einstellung aus vorher gehenden Ausführungen. Bei einer Änderung der Brettgröße werden ebenfalls alle Symbole zurück gesetzt.

2.3 Übungen

Übung 2.2. Buchstabe
Schreiben Sie Anweisungen für eine schöne Darstellung des Anfangsbuchstabens Ihres Namens.

Kapitel 3

Variablen und ganze Zahlen

Bisher haben wir Feldindex und Farbe immer direkt als Zahl eingegeben. Das wird schnell mühsam und ist auch schwer zu ändern oder zu erweitern. In diesem und dem folgenden Kapitel werden wir sehen, wie auch aufwändigere Muster durch entsprechende Möglichkeiten der Sprache C einfach generiert werden können.

3.1 Variablen

Beginnen wir mit dem einfachen Beispiel

```
farbe( 11, 255 );
farbe( 12, 255 );
farbe( 13, 255 );
farbe( 14, 255 );
```

in dem vier aufeinander folgende Felder Blau gefärbt werden. Das funktioniert, ist aber nicht besonders elegant. Wollen wir das Muster verschieben oder die Farbe ändern, so müssen wir alle Zeilen ändern. Besser wäre es, die Informationen an einer Stelle zu bündeln, so dass wir bei Änderungen uns auch nur um eine Stelle kümmern müssen. Beginnen wir mit dem Index und führen eine so genannte Variable ein:

```
int i = 11;
```

In dieser kleinen Zeile stecken einige Neuigkeiten. Vor dem =-Zeichen steht ein Datentyp – `int` – und der Name `i` unserer ersten Variablen. Stellen wir uns das vor wie einen Platz in einem Lagerregal oder ein Schließfach am Bahnhof. Wir wollen in dieses Fach etwas ablegen und später auch wieder abholen. Vielleicht wollen wir später auch etwas anderes in dieses Fach legen. Damit wir unser Fach später wieder finden, geben wir ihm einen Namen. In C und verwandten Programmiersprachen müssen wir außerdem festlegen, welche Art von Daten in dieses Fach gelegt werden soll. In unserem anschaulichen Beispiel Schließfach können wir auch zwischen verschiedenen Größen auswählen. Je nach dem ob wir nur eine kleine

Einkaufstüte oder einen sperrigen Koffer abstellen wollen, müssen wir ein Fach entsprechender Größe wählen.

C unterscheidet mehrere Typen von Daten. Unser `int` steht für Integer, der englischen Bezeichnung für ganze Zahlen. Wir legen also fest, dass unser `i` nur negative oder positive ganze Zahlen einschließlich der 0 enthalten kann.

Allgemein gilt in C, dass der Datentyp einer Variablen fest ist und sich zur Laufzeit nicht mehr ändert. Es handelt sich um eine explizite und statische Typisierung. Viele neuere Programmiersprachen – insbesondere interpretierte Sprachen wie PHP oder Javascript – verwenden demgegenüber eine dynamische und implizite Typisierung. Zur Laufzeit wird der Datentyp bei einer Zuweisung passend festgelegt. Dabei kann sich der Datentyp einer Variablen während der Laufzeit ändern. Beide Ansätze haben Vor- und Nachteile.

Das =-Zeichen ist keineswegs ein mathematisches Gleichheitszeichen sondern der so genannte Zuweisungsoperator. Der Wert auf der rechten Seite wird der Variablen auf der linken Seite zugewiesen. In unserem Fall steht dort nur eine Zahl. Im allgemeinen kann dort aber auch ein Ausdruck stehen, Beispiele dazu folgen gleich. Wichtig ist nur, dass der Ausdruck auf der rechten Seite der Zuweisung zum Typ der Variablen auf der linken Seite passt. Insgesamt haben wir mit der kleinen Zeile also folgendes erreicht:

- es gibt jetzt eine Variable mit dem Namen `i`
- die Variable ist vom Typ `int`
- sie wird mit dem Wert 11 belegt

Die ersten beiden Punkte bezeichnen wir als Definition der Variablen. Man kann auch Variablen definieren, ohne ihnen Werte zuzuweisen. Sie enthalten dann mehr oder weniger zufällige Anfangswerte. Umgekehrt geht es nicht. In C muss jede Variable vor ihrer Verwendung definiert werden. Der Compiler überprüft dies und meldet Verstöße in der Form

```
error: 'j' undeclared (first use in this function)
```

Nach diesen Vorbereitungen können wir jetzt endlich unsere Variable einsetzen und unser Beispiel ändern zu

```
int i = 11;

farbe( i, 255 );
farbe( i + 1, 255 );
farbe( i + 2, 255 );
```

Beim ersten Aufruf von `farbe` verwenden wir direkt unsere Variable, bei den beiden folgenden addieren wir jeweils die Verschiebung dazu. Die absolute Position steht jetzt nur noch an einer Stelle. Unser Mini-Muster kann also durch eine einzige Änderung verschoben werden.

Frage 3.1. Besonders variabel ist `i` aber in diesem Fall nicht.

Das stimmt, in dieser Form nutzen wir `i` wie eine Konstante. Wir können den Code-Abschnitt allerdings wie folgt umschreiben:

```
int i = 11;

farbe( i, 255 );
i = i + 1;
farbe( i, 255 );
i = i + 1;
farbe( i, 255 );
```

Jetzt ändert `i` tatsächlich den Inhalt. Nach jedem Aufruf von `farbe` wird in der Zuweisung

- der Wert in `i` geholt
- dazu eine `1` addiert
- das Ergebnis dann wiederum nach `i` geschrieben.

In diesem Fall hat die Variable `i` in der Anweisung `i = i + 1;` zwei verschiedene Bedeutungen. Auf der rechten Seite der Anweisung ist der Inhalt der Speicherzelle gemeint. Dieser Wert wird zu 1 addiert. Das Ergebnis wird dann in die mit `i` bezeichnete Speicherzelle geschrieben. Man spricht auch von R-Wert und L-Wert (rvalue, lvalue), wenn man die Bedeutung auf der rechten bzw. linken Seite meint. Mit den englischen Begriffen kann man sich unter den Namen auch *read value* und *location value* vorstellen. Eine Konstante kann nur als R-Wert benutzt werden. Eine Anweisung in der Art $7 = 3 + 4$; ist nicht sinnvoll und auch gar nicht erlaubt, der Compiler meldet

```
error: lvalue required as left operand of assignment
```

Um konsequent zu sein, führen wir eine zweite Variable für die Farbe ein:

```
int i = 11;
int linienFarbe = 255;

farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
```

In diesem Fall ändert sich der Wert nicht. Trotzdem lohnt es sich, den Zahlenwert in eine Variable zu speichern und damit über ihren Namen ansprechen zu können. Der Code wird lesbarer und Änderungen werden einfacher. Die Wiederholung

der gleichen Zeile ist noch unschön. Im nächsten Kapitel werden wir auch dies vermeiden lernen.

Frage 3.2. Ist `linienFarbe` dann vergleichbar mit den Farbnamen wie `BLUE`?

Nein, `linienFarbe` ist eine Variable mit einem Datentyp und der Möglichkeit, den Inhalt zu ändern. Die Farbnamen sind definierte Aliase, die nur zur besseren Lesbarkeit eingeführt wurden. Beim Übersetzen werden sie durch die hinterlegten Texte ersetzt.

3.1.1 Ausgabe mit printf

An dieser Stelle ist noch ein Einschub zur Ausgabe sinnvoll. In C steht mit `printf` (**print f**ormated) eine Funktion zur Verfügung, um Werte auszugeben. In unserer Anwendung erscheint die Ausgabe im unteren Fenster. Auch die eigenen Funktionen produzieren solche Ausgaben. Zur Unterscheidung beginnen diese immer mit der Zeichenfolge ». So ergeben die Zeilen

```
int i = 11;
int linienFarbe = 255;

printf("i: %d\n", i );
farbe( i, linienFarbe );
```

die Ausgabe

```
i: 11
>>11 0xff: okay
```

Die allgemeine Form von `printf` ist:

```
printf("Kontrollzeichenkette", Argument1, Argument2, ... )
```

Die durch Anführungszeichen begrenzte Zeichenkette (*Format String*) kann Text (Kommentar) enthalten, der direkt so ausgegeben wird. Weiterhin werden durch `%`-Zeichen Formatbeschreiber eingeleitet, mit denen festgelegt wird, wie die folgenden Argumente auszugeben sind. Im einfachen Fall folgt auf das `%` ein Buchstabe, der das Format angibt. Im Beispiel wurde mit `%d` angegeben, dass ein Integerwerte im Argument folgen wird. Die Bezeichnung `\n` steht für eine neue Zeile (*new line*). Ohne diese Angabe würde nach der Ausgabe kein Zeilenumbruch erfolgen.

Intern analysiert `printf` die Zeichenkette mit den Formatbeschreibern. Dabei wird Anzahl und Typ der weiteren Argumente bestimmt. Daher ist es wichtig, dass die übergebenen Argumente auch genau mit dieser Spezifikation übereinstimmen. Eventuelle Differenzen werden von den meisten Compilern nicht beanstandet. Erst während des Programmablaufs kommt es zu Fehlern.

Die Mischung von eigenen Ausgaben und Ausgaben der Steuer-Funktionen ist etwas heikel. Um Probleme zu vermeiden sollte man

- eigene Ausgaben immer mit einem Zeilenumbruch beenden
- niemals mit » beginnen lassen.

3.1.2 Variablennamen △

Der Name einer Variablen kann in C nach folgenden Regeln gebildet werden:

- Der Name beginnt mit einem Buchstaben.
- Anschließend folgt eine beliebige Folge von Buchstaben und Ziffern .
- Der Unterstrich `_` (*underscore*) kann wie ein Buchstabe eingesetzt werden.
- Groß- und Kleinschreibung wird unterschieden.
- in C reservierte Wörter (Schlüsselwörter) sind verboten (also nicht `int long;`)
- die Länge des Namens ist beliebig, jedoch unterscheiden viele Compiler nur die ersten 31 Zeichen oder sogar noch weniger; also nicht
  - `dies_Ist_Die_Variable_Nummer1`
  - `dies_Ist_Die_Variable_Nummer2`
- C Konvention: Namen von Variablen beginnen mit einem Kleinbuchstaben.

Neben den festen Vorgaben sollte man folgende allgemeine Hinweise beachten:

- Namen sollten passend und weitgehend selbsterklärend (aussagekräftig) sein (nicht `int eineintvariable;`).
- Namen von Integer Variablen fangen oft mit Buchstaben von `i` bis `n` an.
- Für kurzlebige Variablen können kurze Bezeichnungen wie `i`, `j`, `k` verwendet werden.
- Konsequent sein: wenn sich die Bedeutung einer Variable im Programm ändert, auch ihren Namen anpassen.
- Die Struktur innerhalb eines Namens wird mit Groß- / Kleinschreibung markiert:
  - `anzahlStudenten`
  - `strahlungsDauer`

Aufgrund der Ähnlichkeit mit Kamelhöckern spricht man von *CamelCase* („KamelSchrift“), genauer gesagt bei Beginn mit einem Kleinbuchstaben *lowerCamelCase*. In älteren Programmen findet man häufiger den Unterstrich als Trennzeichen (`brett_groesse`).

- Einheitliche Sprache (Englisch, Deutsch oder ...).
- Konsistenz! Ein einmal eingeführter Stil für Namen, Einrückungen, etc. sollte beibehalten werden.

Bei großen Software-Projekten werden oft Namenskonventionen verbindlich vorgegeben.

3.2 Datentyp Integer △

Der Grundtyp `int` ist – abhängig vom verwendeten Compiler – entweder 2 oder 4 Byte groß und enthält Zahlen in 2er-Komplement Darstellung. Der Wertebereich ist dann:

Typ	Speicherbedarf	Wertebereich
int	2 Byte	-32768 ... +32767
int	4 Byte	-2147483648 ... +2147483647

Konstanten für `int` können sowohl als Dezimalwerte als auch in den beiden anderen Zahlensystemen zur Basis 8 und 16 angegeben werden. Für Oktalzahlen fügt man eine führende 0 ein (Beispiel `0156`), bei Hexadezimalzahlen schreibt man 0x... (Beispiel `0xFF`). Das Vorzeichen wird durch - oder ein optionales + gekennzeichnet. Neben dem Basistyp `int` gibt es einige weitere Typen:

Typ	Speicherbedarf	Wertebereich
char	1 Byte	-128 ... 127
unsigned char	1 Byte	0 ...255
short int	2 Byte	-32768 ... +32767
unsigned short int	2 Byte	0 ... +65535
long int	4 Byte	-2147483648 ... +2147483647
unsigned long int	4 Byte	0 ... +4294967295

Statt `short int` und `long int` kann man auch kurz `short` beziehungsweise `long` schreiben. Die tatsächliche Größe von Datentypen kann mit dem Operator `sizeof` ermittelt werden. Der Ausdruck `sizeof( Typ )` liefert die Größe in Bytes zurück. So kann mit den Zeilen

```
printf("char: \t%d\n", sizeof( char ) );
printf("short: \t%d\n", sizeof( short ) );
printf("int: \t%d\n", sizeof( int ) );
printf("long: \t%d\n", sizeof( long ) );
```

die Größe der 4 Datentypen ausgegebene werden. Auf meinem Rechner erhält man

```
char:   1
short:  2
int:    4
long:   4
```

Der Operator kann auch auf Ausdrücke angewendet werden.

```
int i;

printf("int: \t%d\n", sizeof( i ) );
```

liefert ebenfalls den Wert 4. Wenn das Argument ein Ausdruck ist, kann die Klammer auch entfallen. Die Form

```
printf("int: \t%d\n", sizeof i );
```

ist ebenfalls korrekt. Schließlich stellt sich die Frage, was passiert bei Bereichsüberschreitungen? Betrachten wir dazu ein Beispiel:

```
long i = 2147483647; // groesste long Zahl
printf("i: %d\ni+1: %d\n", i, i + 1);
```

Die Ausgabe lautet:

```
i: 2147483647
i+1: -2147483648
```

Indem wir eine 1 zu der größten positiven Zahl addiert haben, sind wir zu der kleinsten negativen Zahl gelangt. Wie in Bild 3.1 dargestellt, kann man sich die Zahlen im Kreis angeordnet vorstellen.

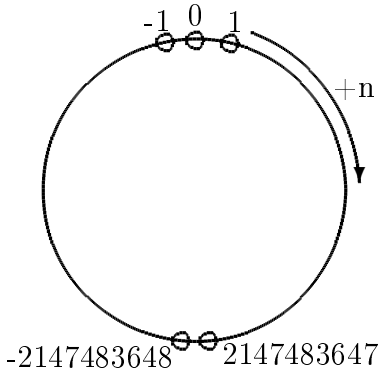


Abbildung 3.1: Anordnung von Integerzahlen

3.2.1 Rechnen mit Integerwerten

C unterstützt die Grundrechenarten mit den Operatoren +, -, * und /. Daneben gibt es noch den Modulo-Operator %, der den Rest bei der Division ergibt. Für die Operatoren gilt die übliche Hierarchie. Bei gleichberechtigten Operatoren wird der Ausdruck von links nach rechts abgearbeitet. Bei der Division muss man berücksichtigen, dass das Ergebnis wieder ein Integer Wert ist und damit ein eventueller Rest verloren geht. Dadurch spielt bei komplexeren Ausdrücken unter Umständen die Reihenfolge eine Rolle. Bei ungeschickter Reihenfolge kann es passieren, dass Zwischenergebnisse nur mit eingeschränkter Genauigkeit berechnet werden. In Folge des damit verbundene Fehlers führt die Auswertung nicht zu dem gewünschten Ergebnis.

Ausdruck	Ergebnis
`32 / 5 * 5`	`30`
`32 * 5 / 5`	`32`
`21 % 6`	`3 (21 - 3 * 6 )`
`28 % 7`	`0`
`21 / 6`	`3`
`23 / 6`	`3`

Man kann die Reihenfolge durch Klammern `()` verändern. Bei der Auswertung werden zunächst die Ausdrücke in Klammern berechnet. Klammern können geschachtelt werden. Die Berechnung beginnt dann bei der innersten Klammer. Dann wird die Klammer durch das Resultat ersetzt und die Berechnung fortgesetzt.

Übung 3.1. Welchen Wert ergeben die folgenden Ausdrücke?

`2 * 5 + 6 * 2`	
`2 * ( 5 + 6 * 2)`	
`2 * ( ( 5 + 6 ) * 2)`	

3.2.2 Inkrement und Dekrement Operator

In C gibt es zwei spezielle Operatoren `++` und `--`, die eine Variable um 1 erhöhen oder vermindern. Diese Operatoren verändern den Operanden und erfordern daher einen lvalue. Sie können beispielsweise nicht auf Konstanten angewandt werden (`++5` ist nicht erlaubt). Der Ausdruck mit dem Operator ist selbst wieder ein rvalue (`++i = ...;` geht nicht). Ungewöhnlich ist, dass die Operatoren vor (präfix) oder nach (postfix) dem Operanden stehen können. Im ersten Fall wird die Variable verändert, bevor sie weiter verwendet wird. Im zweiten Fall wird erst der Wert benutzt, dann verändert.

Mit `n = 7` wird durch `i = ++n;` die Variable `i` auf `8` gesetzt, bei `i = n++;` auf `7`. In beiden Fällen hat `n` anschließend den Wert `8`. Man kann die Operatoren anwenden, ohne die Variable weiter zu benutzen, d. h. `++n;` oder `n--;` sind mögliche Anweisungen und gebräuchliche Abkürzungen für `n = n + 1;` bzw. `n += 1;` (siehe nächster Abschnitt) oder `n = n - 1;` In diesem Fall spielt die Unterscheidung zwischen präfix und postfix keine Rolle. Ansonsten muss man bei komplizierteren Ausdrücken mit unbeabsichtigten Nebeneffekten rechnen. In jedem Fall leidet die Lesbarkeit des Programms. Gefährlich sind Querbezüge in der Art

```
i = 4;
i = i++ * 5;
```

In solchen Fällen ist es besser, eine Zeile mehr zu schreiben und damit klar darzustellen, was gemeint ist. Selbst wenn das Programm tatsächlich das ausführt, was die Programmiererin oder der Programmierer wollte, ist die kompakte Schreibweise schwerer zu verstehen. Es ist auch nicht zu erwarten, dass das entstehende Programm schneller läuft. In aller Regel wird der Compiler selbst durch Optimierung den effizientesten Code generieren.

3.2.3 Vereinfachte Zuweisung

Häufig hat man Zuweisungen in der Art

```
aktienImDepot = aktienImDepot + kauf;
```

d. h. die Variable auf der linken Seite wird auch als erster Operand auf der rechten Seite benutzt:

$$\text{expr1} = \text{expr1 op expr2}$$

C bietet dafür bei den meisten Operatoren mit zwei Argumenten die kompakte Schreibweise

$$\text{expr1 op= expr2}$$

(ohne Leerzeichen zwischen op und =) an. Die wahrscheinlich am häufigsten in der Praxis auftretende Form ist die Verbindung mit der Addition oder Subtraktion: `i += 5;` Hauptvorteil ist die bessere Übersichtlichkeit. Insbesondere wenn der Ausdruck auf der rechten Seite komplex wird, ist in dieser Schreibweise sofort die Bedeutung ersichtlich. Die Schreibweise entspricht der Intention: `i` soll um 5 erhöht werden. Die Berechnung des Ausdrucks auf der rechten Seite hat dabei Vorrang. Das Beispiel

```
i *= j + 5;
```

wird als

```
i = i * (j + 5);
```

ausgewertet.

3.2.4 Bit-Operatoren △

Tief im Inneren des Rechners werden alle Daten binär dargestellt. Jede Speicherzelle enthält eine Reihe von Nullen und Einsen, den einzelnen Bits. Eine Bedeutung erhalten dieses Bits erst durch den Datentyp. Damit legen wir fest, ob das Bitmuster eine Zahl, ein Zeichen oder was auch immer ist. Gelegentlich ist es notwendig oder zumindest hilfreich, das Bitmuster direkt zu bearbeiten. Dazu stellt C eine Reihe von Bit-Operatoren bereit.

Ein Bit-Operator betrachtet demnach den Operanden als Folge von Bits. Diese Bitfolge kann manipuliert werden, indem beispielsweise alle Werte um eine gegebene Anzahl von Positionen geschoben werden. Bei der Verknüpfung zweier Operanden werden alle Bits Position für Position miteinander bearbeitet. Die Bit-Operatoren kann man auf ganzzahlige Daten anwenden. Im Einzelnen bietet C folgende Bit-Operatoren:

Operator	Funktion	Anwendung
`~`	bitweises NICHT	`~`ausdruck
`&`	bitweises UND	ausdruck1 `&` ausdruck2
`|`	bitweises ODER	ausdruck1 `|` ausdruck2
`^`	bitweises EXOR	ausdruck1 `^` ausdruck2
`<<`	schieben nach links (shift)	ausdruck1 `<<` ausdruck2
`>>`	schieben nach rechts (shift)	ausdruck1 `>>` ausdruck2

Das bitweise NICHT invertiert jedes Bit des Operanden. Die drei Operationen UND, ODER und EXOR verknüpfen entsprechend die einzelnen Bits. Im folgenden Beispiel werden zwei Bytes (Datentyp `char`) mit UND verknüpft:

	Binär								Hex.	Dez.
	0	0	1	1	0	0	1	1	33	51
`&`	1	0	1	0	0	1	1	0	A6	166
	0	0	1	0	0	0	1	0	22	34

UND wird oft benutzt um Bits auszuschneiden, während mit ODER Bits gezielt gesetzt werden können. Dazu nutzt man folgenden Beziehungen

$$\begin{aligned} b \text{ UND } 1 &= b \\ b \text{ UND } 0 &= 0 \\ b \text{ ODER } 1 &= 1 \\ b \text{ ODER } 0 &= b \end{aligned}$$

Wendet man *UND Maske* auf einen Wert an, so bleiben demnach an Stellen mit 1 in der Maske die Werte erhalten, an den anderen Stellen werden sie 0. Umgekehrt führt bei *ODER Maske* eine 1 in der Maske zu einem 1 im Ergebnis während bei einer 0 der Wert übernommen wird. In C lässt sich das in der Art

```
i = i & 0xF; // löscht alle außer den letzten vier Bits
i = i | 0xF0; // setzt vier Bits
```

schreiben. Die Shift-Operatoren verschieben den linken Ausdruck um die im rechten Ausdruck angegebene Anzahl von Stellen. Dabei wird beim Schieben nach links stets mit 0 aufgefüllt. Schiebt man nach rechts, so werden bei positiven Zahlen oder dem Datentypen `unsigned` in gleicher Weise die höherwertigen Stellen auf 0 gesetzt. Das Verhalten bei negativen Werten – d. h. Füllen mit 0 oder 1 – ist implementationsabhängig. Einige Beispiele werden im folgenden Code-Abschnitt gezeigt:

```
unsigned char i=15; // Bitmuster 0000 1111

printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);
// Schieben nach links, mit 0 auffuellen
i = i << 1; // Bitmuster 0001 1110
printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// Schieben nach rechts
i = i >> 2; // Bitmuster 0000 0111
printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// ODER mit 0111 0000
i = i | 0x70; // Bitmuster 0111 0111
printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// UND mit 0011 1111
i = i & 0x3f; // Bitmuster 0011 0111
printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);

// Umkehren aller Stellen
i = ~i; // Bitmuster 1100 1000
printf("i= %4d (dez) %4o (oct) %4x (hex)\n", i, i, i);
```

ergibt:

```
i=   15 (dez)   17 (oct)    f (hex)
i=   30 (dez)   36 (oct)   1e (hex)
i=    7 (dez)    7 (oct)    7 (hex)
i=  119 (dez)  167 (oct)   77 (hex)
i=   55 (dez)   67 (oct)   37 (hex)
i=  200 (dez)  310 (oct)   c8 (hex)
```

Die beiden Schrägstriche `//` leiten in C einen Kommentar ein. Der Compiler ignoriert alles ab diesen Zeichen bis zum Zeilenende. Auch mehrzeilige Kommen-

tare sind möglich. Sie werden mit /* eingeleitet und mit */ wieder beendet. Das folgende Beispiel dazu

```
/* ********************
   * erstes Beispiel *
   * s.e. August 2015 *
   ******************** */
```

könnte man gut als Kopf an den Anfang eines Programms schreiben.

3.3 Übungen

Übung 3.2. Welche Namen sind zulässige Bezeichner?

beta	ws01/02	___test___	dritte_loesung
ws2001_okt_08	ss2001-07-08	3eck	monDay
Uebung1.1	Uebung1_1	Uebung1A	Uebung1 A

Kapitel 4

Abläufe

In unserem letzten Beispiel hatten wir mit den Zeilen

```
int i = 11;
int linienFarbe = 255;

farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
i = i + 1;
farbe( i, linienFarbe );
```

drei Punkte hintereinander gezeichnet. Das funktioniert, ist aber etwas umständlich. Wollten wir z. B. nicht nur drei sondern 30 Punkte malen, wäre die Wiederholung der gleichen Zeilen mühsam und unübersichtlich. Glücklicherweise bietet C für solche Fälle Konstruktionen, um mehrfache Wiederholungen kompakt zu formulieren. In diesem Kapitel werden wir die verschiedenen Möglichkeiten kennen lernen und dabei auch logische Ausdrücke als Basis der Entscheidung betrachten.

4.1 Zählschleife

Um eine Anweisung – oder auch mehrere Anweisungen – mit einem Zähler mehrfach zu wiederholen, benutzt man eine so genannte Zählschleife. In C ist dies konkret die for-Schleife. Unser Beispiel lässt sich dann als

```
int i;
int linienFarbe = 255;

for( i=11; i<14; i=i+1 ) {
   farbe( i, linienFarbe );
}
```

schreiben. Die Konstruktion besteht aus dem Kopf mit den Informationen zur Ausführung sowie dem Schleifenrumpf (auch als Schleifenkörper bezeichnet) mit den zu wiederholenden Anweisungen. Der Schleifenrumpf kann eine einzige Anweisung sein oder ein Block in geschweiften Klammern. Es ist aber eine gute Idee, auch bei einer einzelnen Anweisung die Klammern zu schreiben. Dadurch wird die Struktur deutlicher erkennbar. Nun zum Kopf: er besteht aus dem Schlüsselwort `for` und dann in runden Klammern drei Teile, getrennt durch `;`-Zeichen. Diese drei Teile haben folgende Bedeutung:

Initialisierung	`i=11`	wird vor dem ersten Durchlauf ausgeführt
Test	`i<14`	wird vor jedem Durchlauf geprüft Abbruch, falls nicht erfüllt
Fortsetzung	`i=i+1`	wird am Ende jedes Durchlaufs ausgeführt

In unserem Beispiel wird `i` zunächst auf den Startwert 11 gesetzt. Dann wird – wie vor jedem weiteren Durchgang geprüft, ob `i` noch kleiner als 14 ist. Mehr zu Vergleichen folgt später in diesem Kapitel. Falls ja, wird die Anweisung ausgeführt. Danach wird `i` um 1 erhöht und es geht wieder zurück zur Abfrage. Falls nein, wird die Schleife abgebrochen und das Programm geht mit der ersten Anweisung hinter dem Schleifenrumpf weiter.

Frage 4.1. Welchen Wert hat `i` nach der Schleife?

Die Variable behält einfach den letzten Wert. In unserem Fall wäre dies 14.

Frage 4.2. Prüft der Compiler, ob die Bedingungen in der Schleifen-Definition sinnvoll sind?

Nein, der Compiler kontrolliert nur die formale Struktur. Moderne Compiler erkennen allerdings bei der Optimierung Schleifen, die nie ausgeführt werden und lassen sie dann einfach weg. Aber dies gilt nicht als Fehler. So akzeptiert der Compiler sowohl

```
for( i=21; i<14; i=i+1 ) {
```

als auch

```
for( i=0; i<14; i=i ) {
```

Im ersten Fall passiert gar nichts. Falls die Bedingung am Anfang nicht erfüllt ist, wird der Rumpf überhaupt nicht ausgeführt. Man spricht daher auch von einer abweisenden Schleife. Im zweiten Fall bleibt das Programm endlos in der Schleife.

Frage 4.3. Darf ich die Zählervariable innerhalb des Schleifenrumpfs ändern?

Ja, die Zählervariable hat keinen besonderen Schutz. Allerdings muss man aufpassen, dass der Code gut lesbar und verständlich bleibt. Die Beispiele verwenden die Standardform einer Zähl-Schleifen. Allerdings können in den drei Komponenten von `for` beliebige Anweisungen stehen. Man könnte auch das Beispiel als

```
int i;
int linienFarbe = 255;

for( i=11; i<14; farbe( i++, linienFarbe ) );
```

schreiben. Derartige Konstruktionen sind aber schwerer zu lesen und sollten daher nicht genutzt werden. Einzelne Komponenten können leer bleiben. Eine leere Kontrollabfrage gilt als immer wahr. Die Schreibweise `for( ;; )` ist eine gängige Konstruktion für Endlosschleifen.

4.1.1 Geschachtelte Schleifen

Schleifen können ineinander verschachtelt werden. Bei jedem Durchlauf der äußeren Schleife wird die innere Schleife einmal komplett ausgeführt. Als Beispiel laufen in

```
int i, j;

for( i=2; i<6; i++ ) {
   for( j=2; j<6; j++ ) {
        farbe2( i, j, BLUE );
   }
}
```

die beiden Schleifen jeweils von 2 bis 5. Die Variablen `i` und `j` nehmen dabei nacheinander alle möglichen Kombinationen dieser Werte an:

$$(i, j) = (2, 2), (2, 3), (2, 4), (2, 5), (3, 2), (3, 3), \ldots, (5, 5)$$

Im Ergebnis wird ein 4×4-Quadrat gezeichnet (Bild 4.1). Die Schleifen können

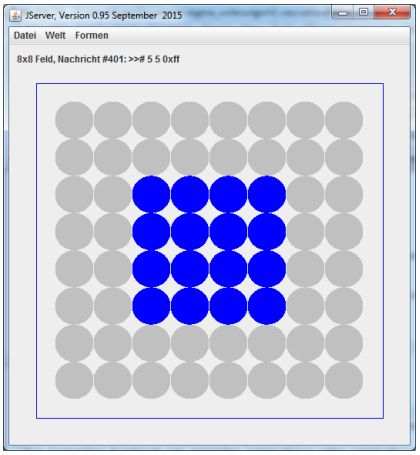


Abbildung 4.1: Quadrat durch verschachtelte for-Schleifen

unterschiedlich lang sein. Es ist auch durchaus möglich, in der inneren Schleife den Zähler der äußeren Schleife zu verwenden. So durchlaufen in

```
int i, j;

for( i=0; i<8; i++ ) {
   for( j=0; j<=i; j++ ) {
        farbe2( i, j, BLUE );
   }
}
```

die Variablen die Wertepaare

$$(i, j) = (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), \ldots, (7,7)$$

und ein Dreieck wird gezeichnet (Bild 4.2)

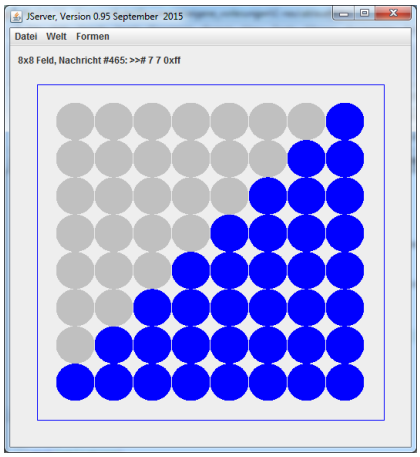


Abbildung 4.2: Dreieck durch verschachtelte for-Schleifen

Übung 4.1. Doppelte Schleife
Was bewirkt folgende Konstruktion:

```
int i, j;

for( i=0, j=0; i<8; i++, j++ ) {
       farbe2( i, j, BLUE );
}
```

4.2 Logische Ausdrücke

Die Entscheidung über einen weiteren Schleifendurchlauf wurde mit dem Ausdruck `i<14` getroffen. Das ist ein einfaches Beispiel für einen logischen Ausdruck.

Tabelle 4.1: Vergleichsoperatoren in C

`==`	gleich
`!=`	nicht gleich
`>`	größer
`>=`	größer gleich
`<`	kleiner
`<=`	kleiner gleich

Tabelle 4.2: Logische Operatoren in C

`!`	Nicht
`&&`	Und
`||`	Oder

Ein logischer Ausdruck ist entweder *Wahr* oder *Falsch*. In C gibt es keinen echten Datentyp für logische Variablen wie in einigen anderen Programmiersprachen. Beispielsweise ist in Java der Typ `boolean`[1] vorhanden. Entsprechende Variablen können nur die Werte `true` oder `false` annehmen. Es gibt allerdings wie wir sehen werden in C Vergleichsoperatoren und logische Operatoren, mit denen logische Ausdrücke gebildet werden können.

Tabelle 4.1 enthält eine vollständige Liste der Vergleichsoperatoren. Damit können Ausdrücke miteinander verglichen werden. Der numerische Wert des Resultat ist 1 falls die Bedingung erfüllt ist und 0 sonst. Mehrere mit diesen Vergleichsoperatoren gebildete logische Ausdrücke können durch logische Operatoren verknüpft werden.

Die logischen Operatoren sind in Tabelle 4.2 zusammen gestellt. So bezeichnet `&&` die Und-Verknüpfung zweier Ausdrücke und `||` die Oder-Verknüpfung. Weiterhin kann durch `!` (NICHT) ein Ausdruck negiert werden. Ist der Ausdruck 0 wird er durch den Operator `!` in 1 konvertiert, ansonsten (bei einem beliebigen von 0 verschiedenen Wert) ist das Resultat 0. Damit ist es möglich, die Rechenregeln der Booleschen Algebra anzuwenden, um beispielsweise Ausdrücke umzuformen.

Beispiel 4.1. Logische Operatoren.

```
n > 5 && n < 10
i == 2 || i == 4 || i == 6
```

Übung 4.2. Logische Operatoren.
Wie kann man in C die folgende Bedingungen abfragen? Getestet werden soll, ob

[1] benannt nach George Boole, engl. Mathematiker 1815-1864, gilt als Begründer der mathematischen Logik

	if( )
eine Variable `i` ein ganzzahliges Vielfaches von 5 ist	
eine Variable `x` im Bereich $[10, 20]$ liegt (einschließlich der Grenzen)	
eine Variable `i` kleiner als -10 oder größer als plus 10 ist	
ein Punkt mit den Koordinaten (x,y) innerhalb eines Kreises mit dem Mittelpunkt (0,0) und dem Radius 2 liegt?.	

Wichtig ist die klare Unterscheidung zwischen dem Zuweisungsoperator `=` und Vergleichsoperator `==`. Viele Programmierfehler entstehen durch die Verwechslung beider Operatoren. Derartige Fehler kann der Compiler oft nicht entdecken. Die Ausdrücke ergeben zwar nicht das, was der Programmierer oder die Programmiererin wollte, sind aber formal vollkommen korrekt. Ähnliche Verwechslungsgefahr besteht zwischen den bitweisen und den logischen Operatoren. Wir haben gesehen, dass Wahr und Falsch durch 1 und 0 dargestellt werden. Damit können `int` Variablen auch als logische Variablen genutzt werden. Ein Ausdruck wie

```
int a, b, c;
...
  a && b && c || a && ! b && c
```

ist eine disjunktive Normalform in C. Man muss sich allerdings klar machen, dass die Variablen immer noch Integer Variablen sind und insbesondere alle von 0 verschiedene Werte als Wahr gelten. Betrachten wir folgendes Beispiel

```
int a = 1, b = -2;
a == b
a && b
```

dann ist der erste Ausdruck Falsch, da die beiden Werte verschieden sind. Andererseits sind aber beide Werte ungleich 0 und gelten damit als Wahr. Somit ist der zweite Ausdruck insgesamt Wahr.

Übung 4.3. Logische Ausdrücke.
Ist der folgende Programmcode korrekt? Was bewirkt die Zuweisung?

```
int i, j;
...
i = j < 10;
```

Frage 4.4. Warum wird der Ausdruck `-5 < i < 5` vom Compiler als gültiger C-Code akzeptiert, funktioniert aber nicht?

Der Complier interpretiert den Ausdruck von links nach rechts, wobei jeder Operator zwei Argumente hat. Er liest also `(-5 < i) < 5`, das ist vollkommen legal. Bei der Ausführung wird zunächst der Ausdruck in der Klammer berechnet. Der Vergleich kann wahr oder falsch ergeben, intern also 1 oder 0. Dieses Ergebnis wird dann eingesetzt und mit 5 verglichen. Egal ob 1 oder 0, beides ist kleiner als 5 und das Endergebnis ist wahr (1). Korrekt muss die Abfrage `-5 < i && i < 5` lauten.

4.3 if Abfrage

Die einfache Fallunterscheidung ist durch die `if-else` Anweisung realisiert. Die allgemeine Form ist:

```
if( ausdruck ) {
    anweisung1
} else {
    anweisung2
}
```

In der runden Klammer steht ein logischer Ausdruck. Abhängig von seinem Wert wird bei Resultat Wahr der erste Block ausgeführt, ansonsten der zweite. Der durch `else` eingeleitete Block ist optional und kann entfallen. Betrachten wir als Beispiel eine Schleife, die ein 8×8 Brett füllt. Abhängig vom Zähler werden die Felder gefüllt.

```
for( i=0; i<64; i++ ) {
   if( i < 10 ) {
       farbe(i, WHITE);
   } else {
       farbe(i, BLACK);
   }
}
```

Das Result zeigt Bild 4.3. Die ersten 10 Felder sind weiß, alle anderen schwarz. Wenn ein Block nur aus einer einzigen Anweisung besteht, kann man die geschweiften Klammern weglassen. Das Beispiel kann auch als

```
if( i < 10 ) farbe(i, WHITE);
else farbe(i, BLACK);
```

geschrieben werden. Die Form ohne Klammern sollte allerdings nur für einfache und übersichtliche Fälle benutzt werden. Die Blöcke können auch leer sein. So kann man die Berechung des Absolutbetrags als

```
if( x < 0 ) {
    x = -x;
```

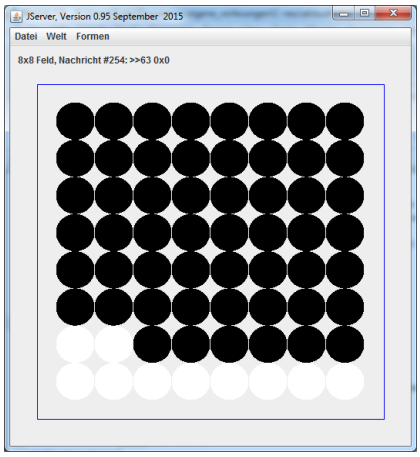


Abbildung 4.3: Ergebnis bei Abfrage `if( i < 10 )`

```
} else {
}
```

schreiben. Ein leerer Else-Block kann weggelassen werden. Aus dem Beispiel wird dann

```
if( x < 0 ) {
    x = -x;
}
```

4.3.1 Else-If

Häufig möchte man mehr als zwei Fälle unterscheiden. If-Abfragen lassen sich beliebig ineinander verschachteln. Allerdings wird die Gesamtstruktur dann schnell etwas unübersichtlich. Daher gibt es für die Fallunterscheidung eine spezielle Form mit `else if`:

```
if( ausdruck1 ) {
    anweisung1
} else if( ausdruck2 ) {
    anweisung2
} else if( ausdruck3 ) {
    anweisung3
} else {
    anweisung4
}
```

In dieser Form werden mehrere Überprüfungen geschachtelt nacheinander ausgeführt. Sobald eine Bedingung erfüllt ist, wird der zugehörige Block ausgeführt und die ganze Kette verlassen (selbst wenn weitere Bedingungen erfüllt wären).

Der letzte (optionale) `else` Block behandelt dann alle verbleibenden Fälle, die von keiner Bedingung abgedeckt sind. Wir erweitern unser Beispiel auf vier Fälle:

```
if( i < 10 ) {
    farbe(i, WHITE);
} else if( i < 20 ) {
    farbe(i, BLACK);
} else if( i < 30 ){
    farbe(i, BLUE);
} else {
    farbe(i, GREEN);
}
```

Wie Bild 4.4 zeigt, werden die ersten drei Zehnerblöcke auf die angegebenen Farben gesetzt, alle restlichen auf Grün.

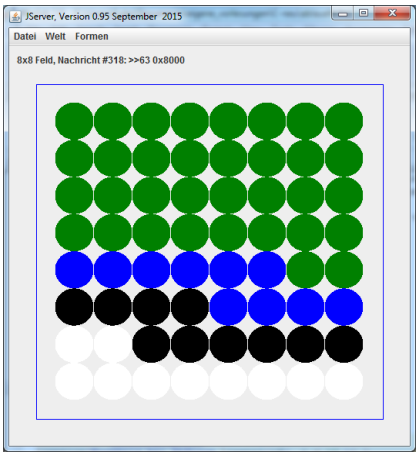


Abbildung 4.4: Ergebnis bei Else-If Abfrage

4.3.2 Fragezeichen-Operator △△

Die Auswahl zwischen zwei Alternativen kann kompakt mit dem ?-Operator geschrieben werden. Die allgemeine Syntax ist

ausdruck ? alternative_ja : alternative_nein

Ist der Ausdruck wahr, so wird die erste Alternative, ansonsten die zweite ausgeführt. Als Beispiel ist

```
p = ( x > 0 ) ? x : -x;
```

eine kompakte Alternative zu

```
if( x > 0 ) {
    p = x;
} else {
    p = -x;
}
```

um den Absolutbetrag zu von `x` berechnen. Die Verwendung des ?-Operators ist Geschmackssache. Man kann damit kompakten und eleganten Code schreiben, aber zumindest für Anfänger ist eine ausführliche Formulierung über if-else übersichtlicher.

4.4 Mehr zu Schleifen

Neben der for-Schleife gibt es in C noch zwei weitere Schleifentypen. Die wohl einfachste Schleife wird mit `while` eingeleitet:

```
while( ausdruck ) {
    anweisung
}
```

Der Ausdruck in der `while` Anweisung wird zunächst ausgewertet. Ergibt er Wahr, so werden die Anweisungen im Block ausgeführt (vorausgehende Bedingungsprüfung). Anschließend wird die Bedingung wieder überprüft und gegebenenfalls der Block erneut ausgeführt. Ein Beispiel dazu

```
i = 0;
while( i < 10 ) {
    farbe( i, RED );
    ++i;
}
```

Die Schleife wird durchlaufen, bis der Wert von `i` größer als 10 wird. Eine vereinfachte Schreibweise erhält man in vielen Fällen, wenn man ausnutzt, dass in C jeder Ausdruck einen Wert hat. Der einfachste Ausdruck – eine Konstante oder eine Variable – liefern als Ausdruck seinen Wert (rvalue). Damit kann man etwa schreiben:

```
i = 10;
while( i ) {
    farbe( i, RED );
    --i;
}
```

um die Zahlen 10 bis 1 auszugeben. Bei der dritten, relativ selten benutzte Form wird die Bedingung erst nach der Ausführung des Blocks geprüft. (nachfolgende Bedingungsprüfung). Die allgemeine Form ist

Tabelle 4.3: Übersicht Schleifen

Typ		C-Syntax
Wiederholung mit vorausgehender Prüfung (Kopfprüfung)	abweisend	`while(){}`
Wiederholung mit nachfolgender Prüfung (Fußprüfung)	nicht abweisend	`do{}while()`
Zählschleife	abweisend	`for(;;){}`

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

In diesem Fall wird die Schleife immer mindestens einmal durchlaufen (nicht abweisende Schleife). Erst am Ende der ersten Schleife wird die Bedingung zum ersten Mal überprüft. Dieser Art von Abfrage ist dann sinnvoll, wenn die Bedingung erst nach der Ausführung ausgewertet werden kann. Man kann diese Form verwenden, um die Korrektheit von Eingaben zu prüfen. In diesem Fall muss die Schleife einmal durchlaufen werden, um überhaupt einen Wert zu haben.

In Tabelle 4.3 sind die drei Grundtypen von Schleifen zusammen mit der C-Syntax zusammen gestellt. Grundsätzlich kann man noch die Wiederholung ohne Prüfung als eigenen Typ betrachten. Derartige Schleifen werden endlos wiederholt. Einsatzgebiet sind Prozesse, die ständig auf Ereignisse warten. Für Endlosschleifen gibt es in C keine gesonderte Konstruktion. Üblich sind die Formen `for(;;)` und `while( true )`.

4.4.1 Vorzeitiges Verlassen von Schleifen

In manchen Fällen ist es erforderlich, eine Schleife entweder ganz zu verlassen oder zumindest den aktuellen Durchgang abzubrechen. In C gibt es dazu die Anweisungen `break` und `continue`. Break beendet die Schleife komplett, der Programmablauf wird mit der ersten Anweisung hinter der Schleife fortgesetzt. Demgegenüber bleibt bei einem `continue` das Programm in der Schleife, nur der aktuelle Durchgang wird nicht zu Ende geführt. Statt dessen wird bei einer `while` oder `do` Schleife die Abbruchbedingung ausgewertet. Bei einer for Schleife wird das Programm mit der Ausführung des dritten Ausdrucks fortgesetzt.

```
int i, summe = 0;
for( i=0; ; i++ ) {
    if( ( summe += i ) > 1000 ) break;
}
printf("Summe %d bei i=%d erreicht\n", summe, i);
```

In diesem Programm wird in der Abfrage ausgenutzt, dass auch eine Zuweisung einen Ausdruck darstellt. Der Wert der Zuweisung ist genau der zugewiesene

Wert. Beispielsweise hat die Zuweisung

```
i = 4 * 5;
```

den Wert (rvalue) 20. Damit ist auch die Anweisung

```
if( i = 0 ) ...
```

syntaktisch korrekt, höchstwahrscheinlich aber inhaltlich falsch.

4.5 Die switch Anweisung

Wenn man eine Auswahl aus vielen einander sich gegenseitig ausschließenden Alternativen treffen will, werden `if ... else if` Strukturen recht unübersichtlich. Als Vereinfachung stellt C die Möglichkeit der `switch` (engl. Schalter) Anweisung zur Verfügung. Sie hat die Form

```
switch( ausdruck ) {
    case konst1:
        anweisung1
    case konst2:
        anweisung2
    default:
        anweisung3
}
```

In der Klammer steht ein Ausdruck, der einen ganzzahligen Wert liefert. Dieser wird dann mit den Konstanten an den `case` (engl. Fall) Marken verglichen. Bei Gleichheit wird das Programm an dieser Stelle fortgesetzt. Etwas gewöhnungsbedürftig ist die Tatsache, dass die Ausführung nicht automatisch durch das nächste `case` beendet wird. Soll – wie es in der Regel der Fall ist – die Bearbeitung der `switch` Anweisung nach einer Übereinstimmung verlassen werden, muss dies explizit durch ein `break` festgelegt werden.

Merkregel 4.1. Nur wenn nach einer `case`-Marke am Ende des Blocks ein `break` steht, wird die gesamte switch-Konstruktion verlassen.

Das optionale `default` „sammelt“ alle Fälle, in denen keine Übereinstimmung gefunden wurde.

Beispiel 4.2. Ausgabe des Wochentages.

```
int wochentag;
switch( wochentag ) {
    case 1: printf("Montag\n" ); break;
    case 2: printf("Dienstag\n" ); break;
    case 3: printf("Mittwoch\n" ); break;
```

```
    case 4: printf("Donnerstag\n" ); break;
    case 5: printf("Freitag\n" ); break;
    case 6: printf("Samstag\n" ); break;
    case 7: printf("Sonntag\n" ); break;
    default: printf("Kein gueltiger Wochentag\n");
}
```

Möchte man mehrere Fälle zusammenfassen, kann man die entsprechenden `case`-Marken direkt aufeinander folgen lassen. Man könnte etwa im obigen Beispiel schrieben

```
    case 6: case 7:
    printf("Wochenende\n" ); break;
```

4.6 Sprünge △△

Wesentlich für die flexible Bearbeitung von Befehlsfolgen ist die Möglichkeit, die Ausführung an einer Stelle abzubrechen und an einer anderen fortzusetzen. Die vorgestellten Elemente für Schleifen und Verzweigungen beruhen implizit auf derartigen Sprüngen in der Befehlsfolge. Aufgrund der großen Bedeutung von Sprüngen stellen Prozessoren in der Regel auch eine reiche Auswahl an Befehlen für absolute und relative, bedingte und unbedingte Sprünge zur Verfügung.

Entgegen dieser großen Bedeutung von Sprungbefehlen auf unterer Ebene, sollten sie in höheren Programmiersprachen weitgehend vermieden werden. Programme mit expliziten Sprungbefehlen werden schnell unübersichtlich und der Ablauf ist schwer nachvollziehbar. Im Prinzip können mit den bereits vorgestellten Elementen alle Abläufe ohne Sprünge realisiert werden Es gibt allerdings einige wenige Fälle, in denen Sprünge zu klareren Programmen führen. Insbesondere ist hier der Rücksprung aus mehrfach geschachtelten Schleifen zu nennen:

```
for( ...) {
    for( ...) {
        for( ...) {
            if( katastrophe ) goto fehlerBehandlung;
...
fehlerBehandlung:
    ...
```

Diese Art von Fehlerbehandlung kann sinnvoll sein, um verschiedene Fehlerfälle gemeinsam zu behandeln. Die Syntax für `goto` ist

```
goto label;
...
label: anweisung
```

Tabelle 4.4: Reihenfolge der Operatoren

`++, --`	Inkrement, Dekrement
`~`	bitweise NICHT
`!`	logisches NICHT
`+, -`	Vorzeichen
`*, /, %`	multiplikative Operatoren
`+, -`	Addition und Subtraktion
`<<, >>`	Bit shift
`<, <=, >=, >`	Vergleichsoperatoren
`==, !=`	Gleichheit, Ungleichheit
`&`	bitweises UND
`^`	bitweises XOR
`|`	bitweises ODER
`&&`	logisches UND
`||`	logisches ODER

Der Name der Zielmarke oder Sprungmarke (*label*, engl. Etikett) wird nach den Regeln für Variablen gebildet. Die Anweisung nach einer Sprungmarke kann auch leer sein (z. B. um an das Ende eines Blocks zu springen).

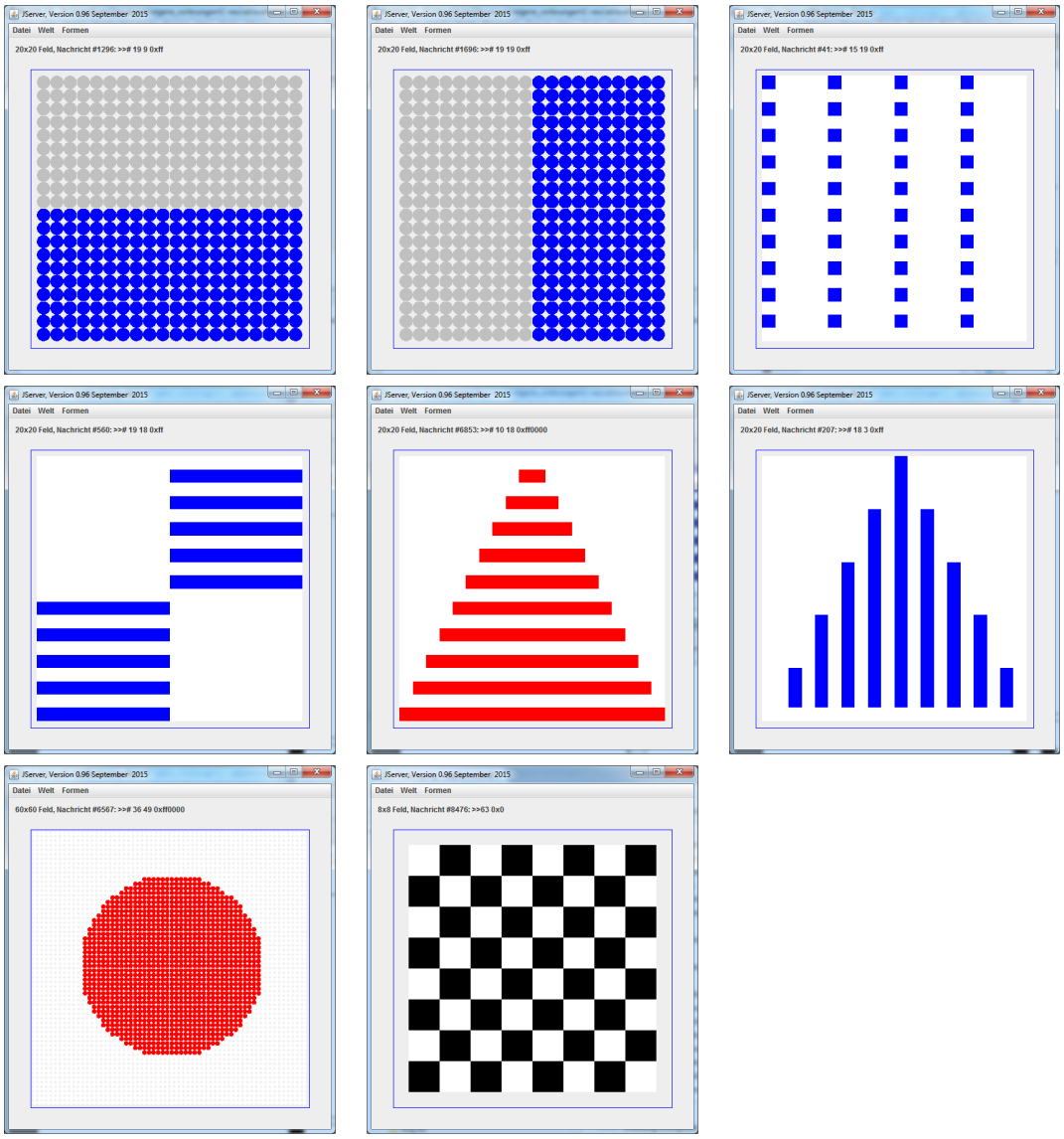
```
label: ;
```

4.7 Rangfolge der Operatoren

In Integer Ausdrücken können arithmetische und logische Operatoren, Vergleichsoperatoren, u. s. w. auftreten. Die Reihenfolge der bisher behandelten Operatoren ist in Tabelle 4.4 zusammengestellt. Je weiter oben ein Operator in der Tabelle steht, desto höher ist sein Rang oder man sagt, er bindet stärker. Am stärksten binden die Operatoren, die nur einen Operanden haben. Gleichberechtigte Operatoren werden von links nach rechts abgearbeitet. Schließlich kann man mit runden Klammern angeben, dass ein Teilausdruck zunächst als Ganzes bearbeitet wird.

4.8 Übungen

Übung 4.4. Muster
Schreiben Sie Anweisungen, um folgende Muster zu erzeugen:


JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #1296: >># 19 9 0xff
JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #1696: >># 19 19 0xff
JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #41: >># 15 19 0xff
JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #560: >># 19 18 0xff
JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #6853: >># 10 18 0xff0000
JServer, Version 0.96 September 2015
Datei Welt Formen
20x20 Feld, Nachricht #207: >># 18 3 0xff
JServer, Version 0.96 September 2015
Datei Welt Formen
60x60 Feld, Nachricht #6567: >># 36 49 0xff0000
JServer, Version 0.96 September 2015
Datei Welt Formen
8x8 Feld, Nachricht #8476: >>63 0x0

Übung 4.5. Fibonacci-Zahlen
Die Fibonacci[2]-Zahlen sind durch die Startbedingung $i_1 = 1$, $i_2 = 1$ und die Rekursion $i_n = i_{n-2} + i_{n-1}$ definiert. Damit gilt

$$\begin{aligned} i_3 &= i_1 + i_2 = 1 + 1 = 2 \\ i_4 &= i_2 + i_3 = 1 + 2 = 3 \\ i_5 &= i_3 + i_4 = 2 + 3 = 5 \\ i_6 &= i_4 + i_5 = 3 + 5 = 8 \end{aligned}$$

und so weiter. Geben Sie alle Fibonacci Zahlen kleiner als 30000 aus. Welchem Wert nähert sich der Quotient i_{n+1}/i_n mit wachsendem n an?

Übung 4.6. Sie betreiben einen Internet-Handel. Bei jedem Kunden zählen Sie die Anzahl der Einkäufe. Abhängig von dieser Zahl gelten Kunden als

Neuling	bis 5 Einkäufe
Kunde	bis 50 Einkäufe
Stammkunde	bis 500 Einkäufe
Gold Kunde	ab 501 Einkäufe

Wie sieht eine `if else if` Abfrage aus, um in Abhängigkeit von der Anzahl der Käufe eines konkreten Kunden seinen Status auszugeben?

[2]Leonardo Pisano genannt Fibonacci, italienischer Mathematiker, ca. 1170-1250

Kapitel 5

Gleitkommazahlen

5.1 Gleitkomma- Darstellung

Bisher hatten wir Zahlen in der Integerdarstellung betrachtet. Diese Darstellung von ganzen Zahlen und das Rechnen damit ist exakt solange man

- im darstellbaren Zahlenbereich bleibt
- keine Nachkommstellen betrachtet (z. B. nach Division)

In vielen praktischen Anwendungen benötigt man aber eine flexiblere Repräsentation von Zahlen. Oft ist das Rechnen mit Nachkommastellen notwendig oder zumindest natürlich. Viele Angaben enthalten Nachkommastellen (Zinssatz 3,5%, 4,7 Liter auf 100 Km). Die erste Erweiterung ist die Einführung von Nachkommastellen. Bei der Festkommadarstellung gibt man die Anzahl der Vor- und Nachkommastellen fest vor. Als Nachteil bleibt dabei die eingeschränkte Dynamik des Zahlenbereichs. Daher wird diese Darstellung nur in wenigen Spezialanwendungen verwendet.

Auch im Alltag und noch mehr in der Technik haben wir das Problem der unterschiedlichen Bereiche. Bei Längen beispielsweise kann je nach Anwendung eine Angabe in mm (Schrauben im Baumarkt) oder in km (Urlaubsreise) sinnvoll sein. Der Trick dabei ist, dass die Länge mit einer Anzahl von signifikanten Stellen und der Größenordnung angegeben wird: 3,5 mm oder 650 km. Vollständige Genauigkeit 650.245.789 mm ist weder sinnvoll noch notwendig. Allgemein schreibt man eine Größe z als Produkt der Mantisse M und einer ganzzahligen Potenz p von 10:

$$z = M \cdot 10^p$$

etwa 3,5 10^{-3} m. Für Maßangaben gibt es Namen bzw. Vorsilben für die entsprechenden Potenzen von 10 (Kilo, Giga, Mega, Nano, etc). Zum Rechnen muss man Zahlen auf eine einheitliche Darstellung normieren:

$$3{,}5\text{mm} + 650\text{km} = 0{,}003.5\text{m} + 650.000\text{m} = 650.000{,}003.5\text{m}$$

Die Beispiele zeigen, wie man durch Verschieben des Kommas in der Mantisse den Exponenten verändert. Es gibt für eine gegebene Zahl (unendlich) viele gleichwertige Darstellungen:

$$123 = 12,3 \cdot 10 = 1,23 \cdot 10^2 = 1230 \cdot 10^{-1} = \ldots$$

Eine einheitliche Darstellung erreicht man mit der Vereinbarung, dass die Mantisse nur genau eine Vorkommastelle hat. Damit hat man eine eindeutige Abbildung zwischen der Zahl z und ihrer Darstellung durch Mantisse m und Exponent p:

$$z \Leftrightarrow (m, p)$$

Die Position des Kommas wird je nach Bedarf verschoben, man nennt daher dieses Format Gleitkommadarstellung (floating point) oder auch halblogarithmische Darstellung. Für die Verwendung in Computern geht man zum Dualsystem über, so dass p dann der Exponent zur Basis 2 ist. In der normalisierten Darstellung wird das Komma so gesetzt, dass nur eine Vorkommastelle bleibt. Die Mantisse dann hat im Dualsystem immer die Form

$$m = 1, \ldots$$

Da die führende 1 bei jeder Zahl steht, braucht man sie in der Realisierung nicht abzuspeichern. In der Praxis sind für m und p nur endlich viele Bits verfügbar. Die Größe von m bestimmt die Genauigkeit der Zahlendarstellung und die Größe von p legt den insgesamt abgedeckten Zahlenbereich fest. Aufgrund der endlichen Größe von m und p kann es zu folgenden Fehlern in der Repräsentation kommen:

- Die Zahl ist zu groß oder zu klein ($z \geq 2^{pmax+1},\ z \leq -2^{pmax+1}$).
- Die Zahl ist betragsmäßig zu klein ($|z| < 2^{-pmax}$ bei symmetrischem Zahlenbereich des Exponenten).
- Die Mantisse ist nicht groß genug, um die erforderliche Anzahl von Stellen zu repräsentieren (Rundungsfehler).

Beispiel 5.1. Geleitkommadarstellung
Betrachten wir folgende Zahlendarstellung im Dezimalsystem: $\pm x.xxx \cdot 10^{\pm ee}$. Geben Sie dazu folgende Werte an:

- Kleinste Zahl
- Größte Zahl
- Betragsmäßig kleinste Zahl $\neq 0$
- Abstand zwischen den beiden größten Zahlen

- Abstand zwischen den beiden betragsmäßig kleinsten Zahlen

Insbesondere die Rundungsfehler bedingen, dass im allgemeinen eine reelle Zahl nicht genau dargestellt werden kann. Berechnet man etwa 1/3 so ist das Resultat 0,33333... prinzipiell nicht exakt darstellbar. Für die Repräsentierung einer Gleitkommazahl benötigt man im Detail:

- Mantisse
- Vorzeichen der Mantisse
- Exponent
- Vorzeichen des Exponents

Weiterhin muss festgelegt werden, wie die insgesamt verfügbaren Bits auf Mantisse und Exponent aufgeteilt werden. Lange Zeit waren die Details der Implementierung von Gleitkommazahlen herstellerabhängig. Da dies zu Problemen beim Datenaustausch und auch bei der Kompatibilität von Programmen führen kann, wurde vor einigen Jahren ein Standard von Normierungsgremien des IEEE (Institute for Electrical and Electronics Engineers, www.ieee.org) verabschiedet. Dieser Standard IEEE 754 definiert 3 Formate:

short real	32 Bit	einfache Genauigkeit
long real	64 Bit	doppelte Genauigkeit
temporary real	80 Bit	erweiterte Genauigkeit

Als Beispiel betrachten wir das Format short real näher:

Bit 31		Bit 0
Vz	Characteristik c	Mantisse m
1 Bit	8 Bit	23 Bit

mit

Vz	Vorzeichen der Mantisse (0 positiv, 1 negativ)
Characteristik	Exponent + 127
Mantisse	Nachkommastellen

Im Gegensatz zu der bei Integer üblichen Darstellung mit dem 2er-Komplement wird die Mantisse mit Betrag und getrenntem Vorzeichen abgelegt (Vorzeichen-Betrag-Darstellung). In der Charakteristik wird der um 127 verschobene Exponent (biased exponent) eingetragen.

Beispiel 5.2. Konvertierung in den Standard IEEE 754
Die Zahl $-37,125_{10}$ soll in das Format short real gebracht werden.

1. Konvertierung in Binärdarstellung: $100101,001$ $(32 + 4 + 1 + 1/8)$

2. Normalisierung: $1,00101001 \cdot 2^5$
3. Mantisse: 00101001
4. Charakteristik: $5 + 127 = 132_{10} = 10000100_2$
5. Vorzeichen: 1 für negative Zahl

Bit 31		Bit 0
1	1000010 0	0010100 10000000 00000000
1 Bit	8 Bit	23 Bit

Die Werte 0 und 255 sind reserviert, um den Wert Null sowie einige Spezialfälle darstellen zu können. Die Null ist ein Sonderfall, da für sie keine normalisierte Darstellung mit einer Eins vor dem Komma möglich ist. Daher wurde vereinbart, dass die Null durch ein Löschen aller Bit in Charakteristik und Mantisse dargestellt wird. Im Detail gilt:

	Vz	Charakteristik	Mantisse
Nicht normalisiert	$\pm$	0	$\neq 0$
Null	$\pm$	0	0
Unendlich (Inf)	$\pm$	255	0
Keine Zahl (NaN)	$\pm$	255	$\neq 0$

Mit den beiden Werten Inf und NaN können Fehlerfälle abgefangen werden. Bei Bereichsüberschreitung wird das Result auf Inf gesetzt, während NaN durch „unerlaubte“ Operationen wie Division von 0 durch 0 oder Wurzel aus einer negativen Zahl entsteht. Damit besteht einerseits die Möglichkeit, solche Fälle zu erkennen. So steht beispielsweise in C die Funktion `_isnan` zur Verfügung, um auf NaN zu testen. Andererseits kann man auch mit den Werten weiter rechnen. Der Standard spezifiziert das Ergebnis von Operationen wie Inf + 10 = Inf. In manchen Algorithmen kann man damit alle Abfragen auf Sonderfälle vermeiden, die sonst den linearen Programmablauf stören würden. Eine ausführliche Darstellung der Thematik enthält der Artikel von Goldberg [Gol91].

Beispiel 5.3. `Inf` und `NaN`
Der C-Code

```
printf( "log( 0.) = %15g\n", log( 0.));
printf( "log(-1.) = %15g\n", log(-1.));
```

liefert das Resultat

```
log( 0.) =          -1.#INF
log(-1.) =          -1.#IND
```

Bei den beiden größeren Typen long real und temporary real wird sowohl die Genauigkeit erhöht als auch der Wertebereich erweitert. Rechnen mit Gleitkommazahlen bedeutet einen wesentlich größeren Aufwand als das Rechnen mit Integerzahlen. Speziell bei der Addition müssen zunächst die Mantissen und Exponenten verschoben werden, bevor die Mantissen addiert werden können. Das Ergebnis muss anschließend gegebenenfalls wieder in die Normalform gebracht werden.

Schnelles Rechen in Gleitkommadarstellung erfordert entsprechenden zusätzlichen Schaltungsaufwand. Früher wurden dafür spezielle Baustein als Co-Prozessoren eingesetzt. Heute ist eine entsprechende Einheit bei den leistungsfähigen Prozessoren bereits integriert. Die Angabe der möglichen Gleitkommaoperationen pro Sekunde (FLOPS: *floating point operations per second*) ist eine wichtige Kenngröße für die Leistungsfähigkeit eines Computers.

Übung 5.1. Gleitkommadarstellung
Betrachten Sie folgendes einfaches Format für Gleitkommazahlen:

- Bit 15: Vorzeichen des Exponenten (0 für positiv)
- Bit 14: Vorzeichen der Mantisse (0 für positiv)
- Bit 6-13 Betrag der Mantisse
- Bit 0-5 Betrag des Exponenten

Welchen Wert haben die folgenden Bitmuster:

0	0	0110 0000	00 0011
1	0	1010 0000	00 0100

Übung 5.2. Wertebereich im Standard IEEE 754
Welches ist jeweils die

- kleinste
- größte
- betragsmäßig kleinste

darstellbare Zahl im short real Format?

Übung 5.3. Konvertierung in den Standard IEEE 754
Wie wird die Zahl $14,625_{10}$ als Gleitkommazahl im Format real short dargestellt? Geben Sie das Resultat in Binär- und Hexadezimaldarstellung an.

Übung 5.4. Addition und Multiplikation
Berechnen Sie für die beiden Zahlen $7,5 \cdot 10^4$ und $6,34 \cdot 10^2$ Summe und Produkt. Geben Sie das Ergebnis jeweils in normierter Form mit einer Vorkommastelle an. Welche einzelnen Rechenschritte sind erforderlich?

Tabelle 5.1: Vergleich zwischen Integer- und Gleitkommazahlen

	Integer	Gleitkomma
Nachkommastellen	Nein	Ja
Genauigkeit	Ergebnisse sind exakt	Rundungsfehler
Bereich	eingeschränkt	groß
Überlauf	wird nicht gemeldet	Inf
Rechenaufwand	niedrig	groß
Speicherbedarf	8-32 Bit	32-80 Bit
	Bit-Operatoren, modulo	

Übung 5.5. Darstellung der Zahl 0
In IEEE 754 gibt es zwei Bitmuster für die Zahl 0, einmal mit positiven und einmal mit negativem Vorzeichen. Diese Werte kann man als +0 und -0 interpretieren. Wo kann man diese Unterscheidung sinnvoll einsetzen? Welche Probleme können sich aus der doppelten Darstellung ergeben?

5.1.1 Vergleich der Zahlenformate

Abschließend sind die wesentlichen Unterschiede in den Zahlendarstellungen in Tabelle 5.1 zusammen gestellt. Abhängig von der Anwendung sind die einzelnen Kriterien unterschiedlich zu gewichten. Beispielsweise spielt bei einer low-cost-Anwendung der Preis eine entscheidende Rolle, so dass auf eine eigene Einheit für Gleitkommaoperationen verzichtet wird. Dann ist es oft notwendig Berechnungen, für die Gleitkommazahlen besser geeignet wären, aus Performanzgründen trotzdem mit Integerzahlen durchzuführen.

5.2 Gleitkommazahlen in C

C kennt 3 Stufen von Gleitkommazahlen:

- float
- double
- long double

Garantiert wird, dass die einzelnen Typen zunehmend genauer oder wenigsten gleich genau sind. In der konkreten Implementierung können aber auch durchaus Typen zusammen fallen. Aus der Hilfe von Visual C++:

> Previous 16-bit versions of Microsoft C/C++ and Microsoft Visual C++ supported the long double, 80-bit precision data type. In Win32 programming, however, the long double data type maps to the double, 64-bit precision data type.

In dieser Umgebung entspricht `float` dem IEEE Format short real und sowohl `double` als auch `long double` entsprechen IEEE long real. Für die Wertebereiche gilt:

Type	Minimum positive value	Maximum value
float	1.175494351 E-38	3.402823466 E+38
double	2.2250738585072014 E-308	1.7976931348623158 E+308

Zwischen Gleitkomma-Werten gelten die Grundrechenarten mit den Operatoren +, -, * und / mit den schon behandelten Vorrangsregeln. Auch die Vergleichsoperatoren sind für Gleitkomma-Werte gültig. Der Modulo Operator `%` allerdings kann nicht auf Gleitkomma-Werte angewandt werden (Allerdings gibt es eine Funktion `fmod()`). Ebenso sind die Bitoperationen für Gleitkomma-Werte nicht definiert.

Die Genauigkeit beträgt etwa 7 Stellen bei `float` und 15 Stellen bei `double`. Bei der Eingabe von Konstanten benutzt man den Punkt anstelle des Kommas. Optional kann man einen Zehner-Exponenten angeben.

Beispiel 5.4. Gültige Gleitkommazahlen:

```
0.456
-177.999
99.988e17
-1e-10
.1e-10
```

Die Werte werden intern normalisiert, bei der Eingabe ist man frei in der Anzahl der Stellen vor dem Dezimalpunkt. Der Dezimalpunkt direkt vor dem Exponenten kann weggelassen werden. Standardmäßig interpretiert der Compiler diese Konstanten als `double`. Durch Anhängen der Endung `f` (`F`) werden sie zu `float` Werten.

5.2.1 Rechnen mit Gleitkommazahlen

Der große Vorzug von Gleitkommazahlen ist der große Wertebereich mit gleichbleibender relativer Genauigkeit. Im Gegensatz zu Integerwerten stellt für übliche Anwendungen der Wertebereich kein Problem dar. Über- oder Unterschreitungen sind eher selten. Allerdings muss man stets mit Rundungseffekten rechnen. Problematisch sind Vergleiche in der Art

```
if( x == 5 )
```

Wenn `x` das Ergebnis einer Rechnung ist, kann es durchaus sein, dass `x` aufgrund der Rundungsfehler den Wert 4.99999 hat. Die Abfrage würde dann nicht erfüllt sein. Besser ist, in einem solchen Fall

```
if( fabs( x - 5 ) < epsilon )
```

zu schrieben, wobei `fabs` den Absolutbetrag berechnet und `epsilon` die gewünschte oder geforderte Genauigkeit angibt. Mit der gleichen Vorsicht muss man for-Schleifen mit Gleitkommawerten betrachten. In dem Beispiel

```
for( x=0.; x<=1.; x+=0.01 )
```

ist nicht gewährleistet, dass der Wert `1` exakt getroffen wird, so dass eventuell (z.B. bei `x=1.0000001`) die Schleife eine Iteration zu früh abgebrochen wird.

Grundsätzlich ist die Abdeckung innerhalb des Wertbereichs – im Gegensatz zu den Integerzahlen – nicht vollständig. Zwischen benachbarten Gleitkommazahlen ist eine Lücke und die Breite der Lücke hängt von der Größe der Zahlen ab. Dies kann zu Effekten führen, die der Mathematik wiedersprechen. Das folgende Fragment C-Code dient zur Veranschaulichung dieses Verhaltens. In dem Programm wird zu einer Zahl der Wert 1 addiert, wobei die Zahl schrittweise um den Faktor 10 erhöht wird.

```
double test = 1.;
double testp1;

do{
    test *= 10.;
    testp1 = test + 1.;
    printf( "%10g %10g %5g \n", test, testp1, testp1-test);
} while( testp1 > test );
```

Die Ausführung liefert die Ausgabe:

```
        10         11     1
       100        101     1
      1000       1001     1
     10000      10001     1
    100000     100001     1
    1e+006     1e+006     1
    1e+007     1e+007     1
    1e+008     1e+008     1
    1e+009     1e+009     1
    1e+010     1e+010     1
    1e+011     1e+011     1
    1e+012     1e+012     1
    1e+013     1e+013     1
    1e+014     1e+014     1
    1e+015     1e+015     1
    1e+016     1e+016     0
```

Zunächst zeigt das Programm das erwartete Verhalten und berechnet die Differenz zu 1. Aber wenn der Wert 10^{16} erreicht ist, führt die Addition nicht mehr zu einer anderen Zahl und die Differenz zwischen 10^{16} und $10^{16}+1$ liefert den Wert 0. Dieser Einfluss der Rundungsfehler ist bei der Programmentwicklung zu berücksichtigen.

Beispiel 5.5. Rundungsfehler
Die folgende Anweisung in C

```
printf( "5. - sqrt(5)*sqrt(5) = %15g\n", 5. - sqrt(5.)*sqrt(5.));
```

ergibt

```
5. - sqrt(5)*sqrt(5) =   -8.88178e-016
```

Übung 5.6. Reihenfolge der Auswertung eines Ausdrucks
Sei $x = 10^{30}$, $y = -10^{30}$ und $z = 1$. Welches Resultat ergibt sich bei dem Rechnen in Gleitkomma-Arithmetik für die beiden Ausdrücke

- $(x + y) + z$
- $x + (y + z)$

5.2.2 Ein- und Ausgabe

Für die Formatbezeichner in `printf` gibt es folgende Möglichkeiten:

f	double	$[-]mmm.dd$
e,E	double	$[-]m.dddde \pm xx$ oder $[-]m.ddddE \pm xx$
g,G	double	`%e` bzw. `%E` wenn der Exponent kleiner als -4 ist oder größer gleich der Anzahl der Nachkommastellen, ansonsten `%f`

Die Anzahl der Nachkommastellen (*precision*) kann vorgegeben werden. Dazu benutzt man die Form `%w.pf` mit

w: Mindestbreite für die Ausgabe des Wertes
p: Genauigkeit, Anzahl der Nachkommastellen oder signifikanten Stellen bei `%g`

Ein `L` vor dem Formatzeichen spezifiziert `long double`.

Beispiel 5.6. Ausgabe von Gleitkommazahlen

```
double w1=1.e-8, w2=-0.2, w3=3.123456789e12;
printf( " 15f: %15f %15f %15f\n", w1, w2, w3);
printf( " 15e: %15e %15e %15e\n", w1, w2, w3);
printf( "15.3e: %15.3e %15.3e %15.3e\n", w1, w2, w3);
printf( " 15g: %15g %15g %15g\n", w1, w2, w3);
```

ergibt:

```
  15f:        0.000000       -0.200000 3123456789000.000000
  15e:   1.000000e-008  -2.000000e-001   3.123457e+012
15.3e:      1.000e-008     -2.000e-001      3.123e+012
  15g:           1e-008            -0.2    3.12346e+012
```

5.2.3 Mathematisch Funktionen

Eine ganze Reihe von mathematische Standardfunktionen sind als Bibliotheksfunktionen in C vorhanden und in `math.h` deklariert. Unter anderem gibt es:

- trigonometrische Funktionen:
  sin(x), cos(x), tan(x), Argument jeweils im Bogenmaß
- inverse trigonometrische Funktionen:
  asin(x), acos(x), atan(x)
- Potenzen und Logarithmen:
  exp(x), log(x), log10(x), sqrt(x), pow(x,y) (x^y)
- Runden:
  - ceil(x) kleinste ganze Zahl größer x
  - floor(x) größte ganze Zahl kleiner x
- Betrag:
  fabs(x) (wichtig: nicht mit int abs(int) verwechseln)

Alle Funktionen erwarten Argumente vom Typ `double` und der Rückgabewert ist ebenfalls `double`. Probleme entstehen, wenn

- das Argument nicht in dem Definitionsbereich liegt (log(-1)) (*domain error*)
- das Ergebnis nicht mehr darstellbar ist (pow(1000,1000)) (*range error*)

Die Funktionen geben in diesen Fällen definierte Sonderwerte (NaN, *Not a Number*) zurück, die auch bei der Ausgabe von `printf` dargestellt werden.

5.3 Umwandlung zwischen Datentypen

In einem Ausdruck können die Operanden verschiedenen Datentyp haben. Der Typ des Ergebnisses hängt dann von den beteiligten Datentypen ab. Auch bei einer Zuweisung hat der Ausdruck auf der rechten Seite manchmal einen anderen Ergebnistyp als die Variable auf der linken Seite. In solchen Fällen erfolgt eine Umwandlung des Typs vor der Rechnung bzw. der Zuweisung. Die Umwandlung

erfolgt dabei stets vom „kleineren“ zum „größeren“ Typ (erweiternde Konvertierung). Beispielsweise wird bei der Addition eines `int` und eines `long` Wertes zunächst der `int` Wert in einen `long` Wert gewandelt. Die Umwandlung erfolgt allerdings „Schritt für Schritt“. Selbst wenn etwa auf der linken Seite einer Zuweisung eine `double` Variable steht, werden nicht notwendigerweise alle Rechnung in `double` ausgeführt. Das folgende Beispiel demonstriert diesen Effekt:

Beispiel 5.7. Umwandlung.

```
double test;
test = 10 / 3;
printf( "Konvertierung 1, test = %g\n", test);
test = 10. / 3;
printf( "Konvertierung 2, test = %g\n", test);
```

Im ersten Fall wird die rechte Seite noch als Integer-Rechnung ausgeführt. Erst das Ergebnis wird umgewandelt. Im zweiten Fall ist der erste Operand `10.` bereits ein `double`, so dass vor der Division die `3` umgewandelt wird. Entsprechend liefert das Beispiel

```
Konvertierung 1, test = 3.0
Konvertierung 2, test = 3.3333333333333335
```

Die Argumente von Funktionen sind ebenfalls Ausdrücke. Wenn keine weiteren Informationen vorliegen, werden Argumente automatisch auf int und double erweitert. Dies auch der Grund, warum es für printf kein Format für float gibt. Ausdrücke vom Typ float werden automatisch zu double erweitert, bevor sie an die Funktion übergeben werden.

Bei der erweiternde Konvertierung bleibt die Information erhalten. Sie werden daher als sicher angesehen und wenn notwendig vom Compiler automatisch eingefügt. Demgegenüber sind Konvertierungen in die andere Richtung (einschränkende Konvertierung) unsicher. Eventuell verliert man Genauigkeit oder der Wert passt nicht mehr in den geringeren Darstellungsbereich. Die genauen Regeln sind in den Referenzen zu C dargestellt.

Daneben gibt es auch Konversionen zwischen Integer und Gleitkomma-Werten. Bei sehr großen Int Werten treten bereits Rundungsfehler bei der Umwandlung nach float auf. Umgekehrt gehen bei der Umwandlung von Gleitkommazahlen zu Integer die Nachkommastellen verloren. Die Werte werden nicht gerundet, sondern abgeschnitten. Bei entsprechender Einstellung warnt der Compiler vor eventuellen Datenverlusten durch Konvertierungen (Projekt - Einstellungen - Warnstufe Stufe 4). Bei der Eingabe

```
int i;
double w1;
i = w1;
```

bekommt man dann die Meldung

```
warning C4244: '=' :
Konvertierung von 'double' in 'int',
moeglicher Datenverlust
```

Man kann explizit eine Konvertierungen mittels des so genannten Type-Cast-Operators anfordern. Die allgemeine Form, um einen Ausdruck `a` in einen anderen Datentyp zu wandeln, ist

```
(Datentyp) a
```

Sofern möglich, wird der Ausdruck in den in Klammern angegebenen Typ konvertiert. Damit werden auch einschränkende Konvertierungen in der Art

```
int i = (int) 1.5;
```

vom Compiler akzeptiert. Mit der expliziten Aufforderung zur Typumwandlung ist der Programmierer bereit, die resultierenden Verluste an Genauigkeit zu akzeptieren.

5.4 Gleitkommazahlen und BoS

In unserer Anwendung BoS gibt es nur ein Funktionspaar, das eine Gleitkommazahl als Parameter erwartet. Mit den beiden Funktionen

```
symbolGroesse(int i, double r)
symbolGroesse2(int i, int j, double r)
```

kann die Größe der einzelnen Symbole verändert werden. Angegeben wird der Radius `r`, wobei der Wert 0,5 einem vollen Feld entspricht. Ein Beispiel für wachsende Symbole zeigt das folgende Code-Abschnitt

```
int i;
int anzahl = 20;

loeschen();
formen( "*" );
groesse( anzahl, 1);
for( i=0; i<anzahl; i++ ) {
    farbe( i, BLACK );
    symbolGroesse( i, 0.1 + 0.4 * i / anzahl );
}
```

und das resultierende Bild 5.1.

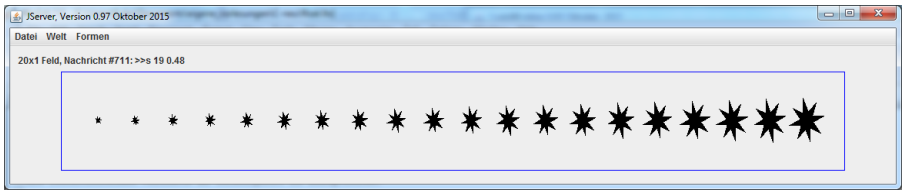


Abbildung 5.1: Symbole mit zunehmender Größe

5.5 Übungen

Übung 5.7. Schreiben Sie ein Programm, das die Lösungen der quadratischen Gleichung $x^2 + p \cdot x + q = 0$ berechnet. Dabei soll q den festen Wert 0,1 haben und p in Schritten von 0,1 von 0 bis 2 laufen. Prüfen Sie jeweils, ob eine reelle Lösung existiert. Falls ja, berechnen Sie die beiden Lösungen. Zur Kontrolle setzten Sie die gefundenen Werte in die Gleichung ein und geben das Ergebnis ebenfalls aus. Im Fall von komplexen Lösungen geben Sie einen entsprechenden Hinweis aus.

Übung 5.8. Nach einer alten Legende wünschte sich der Erfinder des Schachspiels vom König:

- 1 Reiskorn auf dem ersten Feld eines Schachbrettes
- 2 Reiskörner auf dem 2. Feld
- 4 Reiskörner auf dem 3. Feld
- u.s.w.

Geben Sie die Anzahl der Reiskörner mit wachsender Anzahl von Feldern aus. Berechnen Sie zusätzlich, wie viele LKWs (je 7,5 Tonnen Ladung) man für den Transport benötigt, wenn jedes Reiskorn 30 mg wiegt. Wenn weiterhin ein Reiskorn ein Volumen von etwa $3.5 \cdot 10^{-8} m^3$ hat, wie hoch wird dann ein Fußballfeld (70 auf 105 m) bedeckt?

Übung 5.9. Sie nehmen ein Darlehen über 100000 Euro auf. Der jährliche Zinssatz beträgt 4,5%. Jeden Monate zahlen Sie eine Rate von 500 Euro. Verfolgen Sie per Programm die Entwicklung des Darlehens. Berechnen Sie dazu jeden Monat die verbliebene Restschuld und geben diesen Wert jeweils am Ende eines Jahres aus. Wie lange dauert es, das Darlehen zu tilgen? Wie viele Zinsen werden bis zum Ende insgesamt gezahlt worden sein?

Übung 5.10. Muster
Lassen Sie die Funktion $sin(x) * cos(y)$ für Werte von x und y von 0 bis 2π darstellen.

- Rechnen Sie die Brettkoordinaten $0, 1, \ldots, N-1$ in entsprechende Werte aus dem Bereich $[0, 2\pi]$ um.

- Berechnen Sie dann für jedes Feld den entsprechenden Wert $sin(x) * cos(y)$.
- Negative Werte sollen rot, positive grün gefärbt werden.
- Die Funktion liefert Werte zwischen -1 und 1 zurück. Dementsprechend soll die Größe der Symbole gewählt werden. Beachten Sie, dass ein Radius von 0,5 das ganze Feld ausfüllt. Das Programm akzeptiert auch negative Werte.
- (freiwillig) Welche Muster ergeben sich bei anderen Funktionen wie $sin(x) * sin(y)$

Kapitel 6

Zeichendarstellung

6.1 ASCII

Neben dem Rechnen mit Zahlen ist die Verarbeitung von Texten aller Art eine der zentralen Aufgaben von Computern. Dazu ist es erforderlich, die verwendeten Zeichen – Buchstaben, Ziffern, Satzzeichen, etc. – durch Bitmuster im Speicher zu repräsentieren. Für insgesamt N Zeichen benötigt man $\log_2 N$ Bit Wortbreite. Im Prinzip ist die Zuordnung willkürlich. Wesentlich ist nur, dass für jedes Zeichen eindeutig ein Bitmuster vereinbart wird. Allerdings kann durch geschickte Festlegung der Zuordnung das Arbeiten mit Zeichen vereinfacht werden.

Für die Zuordnung zwischen Zeichen und Zahlenwerten existieren verschiedene Systeme. Besonders bei IBM Großrechnern ist das System EBCDIC (*Extended Binary Coded Decimal Interchange Code*) gebräuchlich. Demgegenüber ist bei PCs der ASCII (*American Standard Code for Information Interchange*) Zeichensatz üblich. Ursprünglich waren Zeichen im ASCII Satz nur 7 Bit groß. Damit können dann $2^7 = 128$ verschiedene Zeichen dargestellt werden. Dieser Zeichensatz enthält Buchstaben, Ziffern, Sonderzeichen und spezielle Steuerzeichen, die ursprünglich für Datenkommunikation eingeführt wurden.

Für den allgemeinen Einsatz zur Textverarbeitung wurde es erforderlich, auch nationale Sonderzeichen wie etwa Umlaute für das Deutsche darstellen zu können. Daher wurde ein erweiterter 8-Bit ASCII Zeichensatz mit nationalen Sonderzeichen und einfachen Graphikelementen eingeführt. Die Erweiterung ist allerdings nicht universell, sondern es gibt eine ganze Reihe von nationalen Varianten. Daher sollte man bei Programmen diese Zeichen nur mit Vorsicht einsetzen. Tabelle 6.1 zeigt im Überblick die 7-Bit ASCII Zeichen.

Man erkennt eine Einteilung in vier Blöcke mit je 32 Zeichen. Die ersten 32 Zeichen sind die schon erwähnten Steuerzeichen. Einige der Abkürzungen bedeuten:

BEL Bell (Piepszeichen)

HT Horizontaler Tab (Tabulator)

Tabelle 6.1: 7-Bit ASCII Zeichen

		0	1	2	3	4	5	6	7
Dez.	Binär	000...	001...	010...	011...	100...	101...	110...	111...
0	...0000	NUL	DLE	SP	0	@	P	‘	p
1	...0001	SOH	DC1	!	1	A	Q	a	q
2	...0010	STX	DC2	"	2	B	R	b	r
3	...0011	ETX	DC3	#	3	C	S	c	s
4	...0100	EOT	DC4	$	4	D	T	d	t
5	...0101	ENQ	NAK	%	5	E	U	e	u
6	...0110	ACK	SYN	&	6	F	V	f	v
7	...0111	BEL	ETB	’	7	G	W	g	w
8	...1000	BS	CAN	(	8	H	X	h	x
9	...1001	HT	EM	)	9	I	Y	i	y
10	...1010	LF	SUB	*	:	J	Z	j	z
11	...1011	VT	ESC	+	;	K	[	k	{
12	...1100	FF	FS	,	<	L	\	l	|
13	...1101	CR	GS	-	=	M	]	m	}
14	...1110	SO	RS	.	>	N	^	n	~
15	...1111	SI	US	/	?	O	_	o	DEL

LF Line Feed (Zeilenvorschub)

CR Carriage return (Zeilenanfang)

Die nächsten 32 Zeichen sind die Ziffern und Sonderzeichen. SP steht für das Leerzeichen (engl. space). Darauf folgen die Groß- und Kleinbuchstaben in alphabetischer Reihenfolge zusammen mit einigen weiteren Sonderzeichen. Die Buchstaben sind so angeordnet, dass der Abstand zwischen zusammen gehörenden Groß- und Kleinbuchstaben immer 32 beträgt. Damit unterscheiden sich die jeweiligen Bitmuster nur an einer Stelle und die Umwandlung zwischen beiden Formen ist einfach auszuführen.

An dieser Stelle sei nochmals der Unterschied zwischen Ziffern und den Zahlen betont. Die Ziffer 2 wird durch das Bitmuster 110010_2 im Speicher repräsentiert. Interpretiert man dieses Bitmuster als Integerzahl, so ergibt sich der Wert 50_{10}. Viele Programmiersprachen erlauben es, eine solche Speicherzelle sowohl als Zeichen als auch als Zahl zu interpretieren. Damit kann man beispielsweise in C eine Konstruktion in der Art

```
i = ziffer - '0';
```

verwenden, um aus einer Ziffer den zugehörigen Integerwert zu berechnen. Die Subtraktion bezieht sich nicht auf den Zahlenwert, sondern auf die Position in der Zeichentabelle. Durch die Anordnung der Ziffern in der ASCII entspricht der

Abstand einer Ziffer zu der Ziffer 0 gerade ihrem Zahlenwert. Allerdings führt eine solche Konstruktion zu einer Abhängigkeit vom System. Es ist nicht garantiert, dass jede Zeichentabelle auf einem beliebigen Computersystem so aufgebaut ist, dass die Differenz den richtigen Wert ergibt. In C werden Zeichenkonstanten mit einfachen Hochkomma markiert. Das Beispiel

```
char c = 'a';
```

weist der Variablen `c` den Buchstaben „klein a“ zu. Für die Steuerzeichen gibt es so genannte Escape-Sequenzen als Abkürzungen. Beispielsweise ist `'\n'` das Zeichen für newline (neue Zeile). Andere gebräuchliche Sequenzen sind `'\t'` für einen Tabulator-Schritt und `'\''` um ein Hochkomma zu erhalten. In C ist der char Typ ein Untertyp von Integer. Damit kann man char Variablen auch wie Integervariablen benutzen und mit ihnen rechnen. So ist es durchaus möglich, char Variablen Zahlenwerte zuzuweisen. Nach der Anweisung

```
char c = 77;
```

hat c den Wert dezimal 77. Dem entspricht das Zeichen „groß M“. Genauso kann man Vergleiche mit char Variablen durchführen.

Beispiel 6.1. Abfrage auf Kleinbuchstabe.

```
if( c >= 'a' && c <= 'z' ) {
  printf( "c = %c ist ein Kleinbuchstabe\n", c);
}
```

Man könnte den Vergleich auch als

```
c >= 97 && c <= 122
```

schreiben, aber die erste Form ist wesentlich aussagekräftiger. Allerdings enthält auch die erste Form Annahmen über den benutzten Zeichensatz. Besser ist es, Standardfunktionen von C für derartige Vergleiche zu benutzen. Unter anderem stehen folgende Vergleiche zur Verfügung:

```
islower( c )
isupper( c )
isalpha( c )
isdigit( c )
```

Die Funktionen geben einen von 0 verschiedenen Wert (wahr) zurück, wenn die Bedingung erfüllt. Weiterhin kann man mit

```
c = toupper( c );
c = tolower( c );
```

eine Variable definiert auf Groß- oder Kleinbuchstaben umsetzen. Diese Funktionen sind in der Datei `ctype.h` deklariert.

6.1.1 switch Anweisungen mit char Variablen △

Eine besonders praktische Struktur kann man mit char Variablen und der switch Konstruktion aufbauen. Wenn man Befehle mit genau einem Zeichen benutzt, kann man mit auf diese Art übersichtlich die verschiedenen Aktionen darstellen.

Beispiel 6.2. switch Anweisung.

```
switch( command ) {
    case 'a':
        /* Aktion für Befehl a */
        break;
...
    case 'x':
        return;
    case 'h':
    case '?':
        /* Ausgabe von Hilfe */
        break;
    default:
        printf( "Unbekannter Befehl %c\n", command );
}
```

6.2 Zeichen und BoS

Mit dem Funktionenpaar

```
zeichen(int i, char c)
zeichen2(int i, int j, char c)
```

können Zeichen auf das Brett geschrieben werden. Die Zeichen werden jeweils zentriert auf das angegebene Feld gesetzt. Auch hierzu ein Beispiel: mit

```
char z;
int i;

groesse( 16, 5 );
for( i=0; i<80; i++ ) {
   z = 48 + i;
   zeichen(i, z );
}
```

werden die ASCII-Zeichen von 48 bis 127 dargestellt (Bild 6.1).

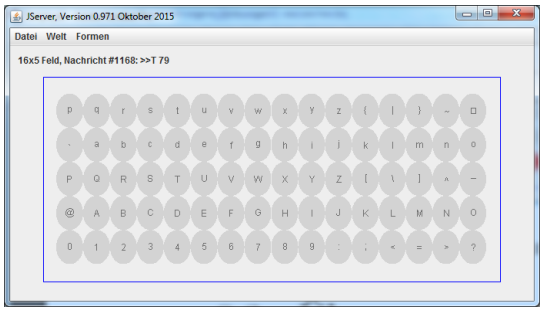


Abbildung 6.1: Einzelne Zeichen

6.3 Übungen

Übung 6.1. Erweitern Sie das Beispiel zur Darstellung der ASCII-Zeichen, in dem Sie die drei Gruppen Ziffern, Klein- und Großbuchstaben farbig markieren. Verwenden Sie zur Prüfung die Funktionen `isdigit( c )`, ...

Kapitel 7

Felder

In Kapitel 3 haben wir gelernt, dass wir über die Variablennamen Speicherplätze ansprechen können. Betrachten wir diesmal Häuser mit ihren Bewohnern als anschauliches Beispiel. Dann vergeben wir für jedes Haus einen Namen. Bild 7.1 zeigt ein entsprechendes Beispiel. Wir haben einen Platz oder eine Straße mit vier Häusern. Jedes Haus hat einen Namen und diese Namen dienen als Adresse.

Mit einigen wenigen Häusern funktioniert das gut. Wenn auf einem Paket angegeben ist, dass Sabine in Haus *Sonne* wohnt, kann der Kurier es gut zustellen. Schwierig wird es, wenn die Anzahl der Häuser wächst. Schon die Vergabe eindeutiger Namen ist nicht einfach und die Suche nach einem Haus kann langwierig werden. Die praktische Lösung ist, den Häusern keine Namen sondern fortlaufende Nummern zu geben. Man benötigt dann nur noch einen Namen für die Straße. Die Kombination Straßenname und Hausnummer ist dann eine gut zu findende Adresse. Bild 7.2 veranschaulicht dieses Konzept.

In der Tat benötigt man in der Programmierung häufig nicht nur eine einzelne Variable eines bestimmten Datentyps, sondern eine Reihe oder Anzahl von gleichartigen Variablen. Beispiele sind:

- die Matrikelnummern aller Hörer und Hörerinnen dieser Vorlesung
- alle Primzahlen kleiner 1000

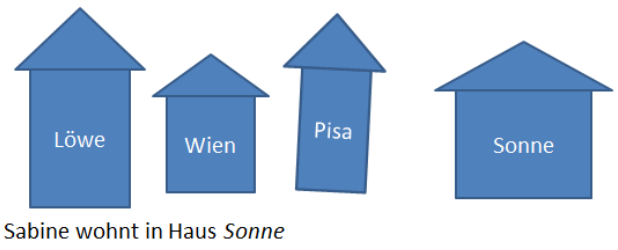


Abbildung 7.1: Häuser mit Namen

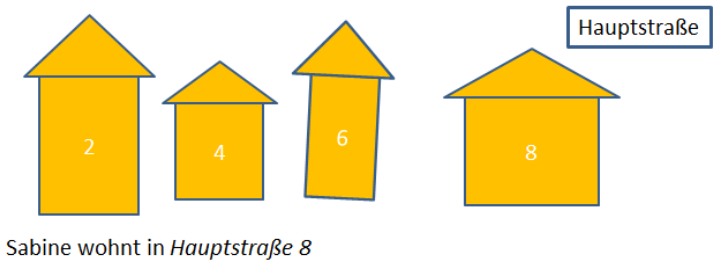


Abbildung 7.2: Häuser mit Namen

- die Zeichen I N F O R M A T I K im Wort Informatik

Charakteristisch ist, dass ein fester Datentyp mehrfach benötigt wird. In dem Beispiel der Matrikelnummern braucht man eine Struktur, die nacheinander alle Nummern (jeweils in einer int Variablen) enthält. Im Speicher hat man folgende Darstellung:

1. Matrikelnummer
2. Matrikelnummer
...
N-te Matrikelnummer

Die N aufeinander folgende Speicherzellen enthalten die Matrikelnummern der Studenten und Studentinnen. Die entsprechende Datenstruktur ist das Feld (engl. *array*). Da ein Feld aus den einfachen Datentypen aufgebaut ist, spricht man von einem strukturiertem Datentyp. Ein Feld ist charakterisiert durch:

- einen Namen (Bezeichner)
- einen Datentyp
- der vorgegebenen, festen Anzahl der Elemente, d. h. Variablen des Datentyps
- Indizes für die Elemente

In C erfolgt die Definition eines Feldes analog zur Definition von elementaren Variablen mit einer zusätzlichen Größenangabe in eckigen Klammern `[]`:

```
short matrikel[150];
float x[100];
char vorlesungsNamen[30];
static unsigned int test[33];
```

Wichtig ist, dass die Anzahl der Elemente fest ist und zur Compile-Zeit bekannt sein muss, da der Compiler entsprechend viele Speicherzellen reservieren muss. Möglich sind daher Definitionen wie

```
int ifeld[5*20];
```

da der Ausdruck vom Compiler eindeutig ausgewertet werden kann. Nicht erlaubt ist demgegenüber

```
int size = 10;
double x[size];
```

In diesem Fall betrachtet der Compiler `size` als nicht hinreichend fest und meldet:

```
error C2057: Konstanter Ausdruck erwartet
```

Man kann allerdings solche indirekte Größenangaben einbauen, indem man das zusätzliche Attribut `const` vor die Definition von `size` schreibt. Damit wird `size` zu einer Konstanten, deren Wert nicht mehr geändert werden kann. Der Code

```
const int size = 10;
double x[size];
```

ist erlaubt. Nach der Definition `double messwerte[30]` als Beispiel gilt:

- Es gibt ein Feld mit dem Namen `messwerte`.
- Das Feld besteht aus 30 Speicherzellen.
- Jede Speicherzelle hat Platz für einen Wert vom Typ `double`.

7.1 Initialisierung

Mit der Definition wird entsprechend viel Speicher reserviert. Standardmäßig sind alle Zellen entweder mit 0 initialisiert (globale oder static Definition, dazu später mehr in Kapitel 10) oder sie enthalten zufällige Werte. Man kann Felder initialisieren, indem man bei der Definition eine Liste mit Werten in geschweiften Klammern angibt.

Beispiel 7.1. Initialiserung eines Feldes

```
int gerade[7] = {2, 4, 6, 8, 10, 12, 14};
```

Man muss nicht alle Elemente initialisieren. In jedem Fall beginnt die Initialisierung aber mit dem ersten Element. Es gibt keine Möglichkeit, nur Elemente beispielsweise in der Mitte eines Feldes zu initialisieren. Lässt man die Größe des Feldes bei der Definition mit Initialisierung weg, so wird die Größe automatisch bestimmt. In dem Beispiel

Beispiel 7.2. Automatische Festlegung der Feldgröße

```
int a[] = {10, 20, 30 };
```

wird ein Feld mit 3 Elementen erzeugt. Wohl am häufigsten werden Felder mit `char` Werten benutzen. Auf diese Art kann man Zeichenketten aus einzelnen Zeichen zusammen setzen. Die Initialisierung mit einzelnen Zeichen hat dann die Form

Beispiel 7.3. Definition eines Feldes mit Zeichen

```
char informatik[] =
{ 'i', 'n', 'f', 'o', 'r', 'm', 'a', 't', 'i', 'k'};
```

Solche Zeichenketten sind die Grundlage von Textverarbeitung in C Programmen. Ein Problem ist dabei allerdings, dass man zunächst keine Information über die Länge der Zeichenkette hat. Insbesondere wenn die Zeichenkette in einem größeren Feld abgelegt ist, kann man nicht erkennen, wann das Ende erreicht ist. Es gibt daher in C die Vereinbarung, dass eine Zeichenkette mit einer NULL (`'\0'`, Wert 0) abgeschlossen wird. Bei der Analyse eines Feldes mit Zeichen markiert das erste NULL Zeichen das Ende. Leerzeichen oder Tabulatoren werden demgegenüber als Bestandteile der Zeichenkette behandelt. Solche abgeschlossenen Zeichenkette (Strings) können als ganzes in doppelten Anführungsstrichen angegeben werden. Die abschließende NULL wird automatisch angehängt. Ein entsprechendes Beispiel ist

```
char informatik2[] = {"informatik"};
```

Damit wird wieder ein Feld angelegt, das nacheinander die Zeichen `i n f` ... enthält. Allerdings ist das Feld in diesem Fall um ein Element – für die abschließende NULL – größer als bei der Definition in Beispiel 7.3. Die geschweifte Klammern können in diesem Fall auch der Einfachheit halber weggelassen werden.

7.2 Zugriff auf Elemente

In C bieten die Felder einen Namen für eine Reihe gleicher Elemente. Man kann nicht direkt mit Feldern rechnen. Eine Anweisung wie

```
int a[3], b[3];
a += b;
```

ist nicht möglich. Zum Rechnen muss man auf die einzelnen Elemente zugreifen. Zugriff erfolgt über den Index, d. h. der Position im Feld. Die Zählung der Elemente beginnt in C immer mit dem Index 0. Bei einem Feld mit 10 Elementen

haben die einzelnen Element die Indizes $0, 1, 2, \ldots, 9$. Ansprechen kann man ein Element über den Namen des Feldes und den Index in eckigen Klammern:

```
a[2]
```

ist das 3. Element (das Element mit Index 2) des Feldes `a`. Mit einem solchen Element kann man umgehen wie mit einer Variablen. Es kann sowohl in Ausdrücken eingesetzt werden als auch Ziel einer Zuweisung sein.

```
int a[5], b[5];
a[0] = 50 * b[2] + 30 * b[3];
```

Um alle Elemente eines Feldes zu bearbeiten, kann man beispielsweise `for`-Schleifen benutzen. So wird in

```
const int size = 10;
int feld[size];
int i;

for( i=0; i<size; i++ ) {
    feld[i] = i * i;
}
```

ein Feld mit den ersten 10 Quadratzahlen gefüllt. In C wird bei dem Zugriff auf ein Feld nicht überprüft, ob der Index im gültigen Bereich liegt. Das Programm

```
int n[3];
printf( "%d %d\n", n[-1], n[4] );
```

lässt sich kompilieren und auch ausführen. Bei der Ausführung werden Speicherzellen, die nicht zu dem Feld `n` gehören, ausgelesen. Abhängig davon, in welchen Speicherbereich diese Adresse fallen, werden entweder falsche Werte benutzt oder die Anwendung wird abgebrochen. Es liegt in der Verantwortung der Programmiererin oder des Programmierers dafür zu sorgen, dass die verwendeten Indizes im gültigen Bereich liegen.

7.3 Mehrdimensionale Felder

Felder sind nicht auf eine Dimension beschränkt. Bei der Definition eines Feldes können mehrerer Dimensionen unterschiedlicher Größe angegeben werden:

```
char tabelle[8][16];
```

definiert einen Bereich von 8 mal 16 char Werten. Solche Felder sind geeignet zur Speicherung von Bildern, 3-dimensionalen Objekten, etc. In jeder Dimension läuft der Index wieder von 0 bis N-1. Der Zugriff auf einzelne Elemente erfolgt entsprechend:

```
tabelle[3][2] = 'i';
```

Jeder Index steht dabei in einem eigenem Paar von Klammern. Intern wird aus den Indizes und den Größen in jeder Dimension die Adresse des Speicherplatzes für das ausgewählte Element berechnet.

7.4 Zeichenketten

Die mit `NULL` abgeschlossenen Zeichenketten erlauben die komfortable Zeichenverarbeitung in C. Sie sind zwar kein vollständiger Ersatz für einen eigenen Typ (z. B. String in JAVA), aber durch die spezielle Konvention vereinfacht sich die Benutzung wesentlich. Der Einsatz wird durch eine Reihe von Bibliotheksfunktionen (deklariert in `string.h`) unterstützt:

strcpy(s, ct)	Kopiert die Zeichenkette ct nach s
strcat(s, ct)	Hängt die Zeichenkette ct an s an (*concatenation*)
strcmp(cs, ct)	Vergleicht zwei Zeichenketten. Ergebnis: < 0 wenn cs $<$ ct 0 wenn cs $==$ ct > 0 wenn cs $>$ ct
strlen(cs)	Die Länge von cs (ohne NULL)

Mit der Funktion `strcmp` hat eine Abfrage auf Gleichheit die Form

```
if( strcmp( a, "test" ) == 0 ) ...
```

oder

```
if( ! strcmp( a, "test" ) ) ...
```

Die Tabelle enthält nur die wichtigsten Funktionen. Die vollständige Liste findet man in der Dokumentation zu C.

Beispiel 7.4. Verarbeitung von Zeichenketten

```
    char land1[] = "Hessen", land2[] = "Bremen";
    char name[200] = "LEER";

    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <LEER>, laenge = 4

    strcpy( name, land1);
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <Hessen>, laenge = 6

    strcat( name, land2);
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
```

```
>> name : <HessenBremen>, laenge = 12

    strcpy( name, land1);
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <Hessen>, laenge = 6

    strcat( name, " ");
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <Hessen >, laenge = 8

    strcat( name, land2);
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <Hessen Bremen>, laenge = 14

    strcpy( name, land1 + 1);
    printf("name: <%s>, laenge = %d\n", name, strlen(name));
>> name : <essen>, laenge = 5
```

Zeichenketten werden mit dem Formatkennzeichner `%s` durch `printf` ausgegeben. Dabei wird die ganze Kette bis zur abschließenden `NULL` inklusive eventueller Leerzeichen ausgegeben. Beim Einlesen mit `scanf` und `%s` hingegen wird bis zu dem nächsten Trennzeichen (Leerzeichen, Tab, Zeilenumbruch) gelesen. Eine komplette Zeile wird mit `gets( s )` eingelesen. Nach dem Aufruf enthält dann das Feld `s` die komplette Zeile, wobei das Zeichen für den Zeilenumbruch durch NULL ersetzt wird. Mit speziellen Varianten von `scanf` und `printf` kann man auch aus Zeichenketten lesen und in Zeichenketten schreiben. Die Funktionen heißen `sscanf` und `sprintf`. Sie haben als zusätzliches (erstes) Argument den Namen eines Feldes. Der Code

```
char a[100];
int i = 10;
sprintf( a, "i = %d", i);
```

schreibt den Text `i=10` in das Feld `a`.

7.5 Zusammenfassung Felder

`int a[5];`	Definition eines Feldes mit 5 Elementen
`short s[3] = {2, 4, 6};`	Definition und Initialisierung eines Feldes mit 3 Elementen
`float x[] ={1.1, 2.2, 3.3}`	Definition und Initialisierung eines Feldes mit 3 Elementen, automatische Größe
`x[0] = 2*x[1];`	Zugriff auf einzelne Elemente eines Feldes
`int b[80][40];`	Definition eines zweidimensionalen Feldes
`b[22][34] = 1;`	Zugriff auf einzelne Elemente eines zweidimensionalen Feldes

Für Zeichenketten gilt:

- Felder vom Datentyp `char`
- Ende der Zeichenkette durch NULL markiert
- Zeichenketten-Konstanten in Anführungszeichen
- Formatkennzeichnung `%s` in `printf` und `scanf`
- Einlesen einer ganzen Zeile mit `gets( feld )`

7.6 Unterschiede zu anderen Programmiersprachen

Das Konzept der Felder findet man auch in anderen Programmiersprachen. In Detail gibt es folgende Besonderheiten bei C:

- der Index beginnt immer bei 0
- der Zugriff ist nicht kontrolliert
- Felder haben eine feste Größe
- es gibt keine Operationen auf ganzen Feldern

7.7 Felder und BoS

Auch in diesem Kapitel lernen wir noch neue BoS-Funktionen kennen. Die beiden Funktionen

```
text(int i, char c[])
text2(int i, int j, char c[])
```

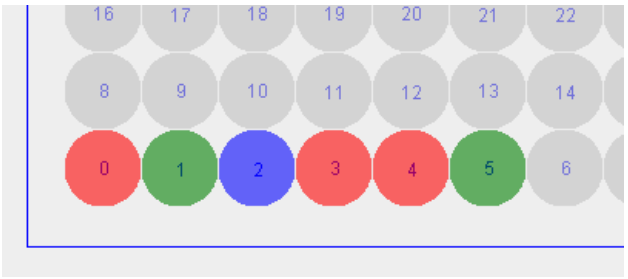


Abbildung 7.3: Farbenfolge in Feld

erwarten jeweils eine Zeichenkette (ein Feld von char) und schreiben sie zentriert in das angegebene Feld. Ansonsten werden Felder nicht in den Funktionen verwendet. Aber Felder sind hervorragend geeignet, um Anwendung für BoS einfacher und übersichtlicher zu schreiben. Wir können zum Beispiel eine Farbenfolge in einem Feld speichern und dann direkt die Felder setzen. Der Code-Abschnitt

```
int farben[] = { RED, GREEN, BLUE, RED, RED, GREEN };
int i;

for( i=0; i<6; i++ ) {
    farbe( i, farben[i] );
}
```

generiert das Muster in Bild 7.3. der Feldindex entspricht dabei der Nummerierung der Brettfelder.

Frage 7.1. In dem Beispiel steht die Feldgröße als Konstante 6 im Code. Dann muss man bei jeder Änderung der Feldgröße auch diesen Wert ändern. Kann man nicht die Feldgröße abfragen?

Das geht tatsächlich mit dem Operator `sizeof`. Wir können das Beispiel besser als

```
int farben[] = { RED, GREEN, BLUE, RED, RED, GREEN };
int anz = sizeof farben / sizeof farben[0];
int i;

for( i=0; i<anz; i++ ) {
    farbe( i, farben[i] );
}
```

schreiben. Die Variable `anz` enthält jetzt die Länge – d. h. die Anzahl der Elemente – von `farben`. Der Operator `sizeof` gibt die Größe in Bytes zurück. Daher müssen wir noch durch die Größe des ersten Elementes dividieren. In diesem Beispiel geben wir die Farben an und die Nummerierung läuft mit dem Feldindex. Wir

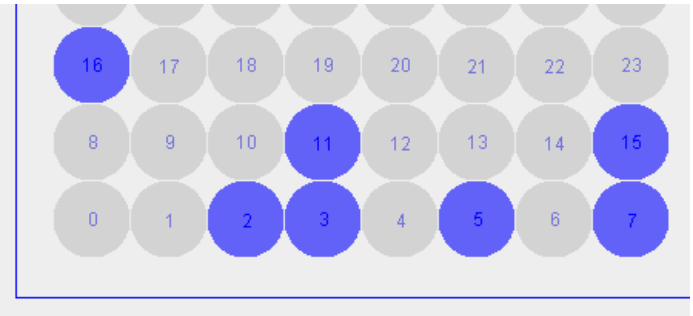


Abbildung 7.4: Felder in Blau

können dies auch umkehren: für eine gegebene Farbe speichern wir die Nummern in ein Feld. In

```
int blau[] = { 3, 5, 7, 11, 15, 16, 2 };
int anz = sizeof blau / sizeof blau[0];
int i;

for( i=0; i<anz; i++ ) {
    farbe( blau[i], BLUE );
}
```

geben wir vor, welche Brettfelder Blau gefärbt werden sollen (Bild 7.4). Die größte Flexibilität erreichen wir indem jeweils Feldnummer und Farbe paarweise abgespeichert werden. Wertepaare lassen sich äquivalent in zweidimensionalen Feldern abspeichern. Die entsprechende Variante ist

```
int felder[][2] = { {3, BLUE}, {5, RED}, {1, GREEN} };
int anz = sizeof felder / sizeof felder[0];
int i;

for( i=0; i<anz; i++ ) {
    farbe( felder[i][0], felder[i][1] );
}
```

mit drei Wertepaaren Feldnummer – Farbe (Bild 7.5).

Frage 7.2. `felder` ist ein zweidimensionales Feld, was bedeutet dann `sizeof felder[0]`?

Zweidimensionale Felder sind Felder von Felder. Das kann man sich vorstellen wie ein Rechteck, das aus mehreren Zeilen aufgebaut ist. Allgemein besteht ein n-dimensionales Feld aus (n-1)-dimensionalen Einheiten. Wenn man bei der Angabe eine Dimension weg lässt, so bezieht man die Einheit der niedrigeren Dimension. In unserem Fall ist `felder[0]` das erste eindimensionales Feld. Indem wir die Gesamtgröße durch die Größe dieses Feldes teilen, erhalten wir automatisch die Anzahl der eindimensionalen Felder. Über diese Anzahl läuft dann die for-Schleife.

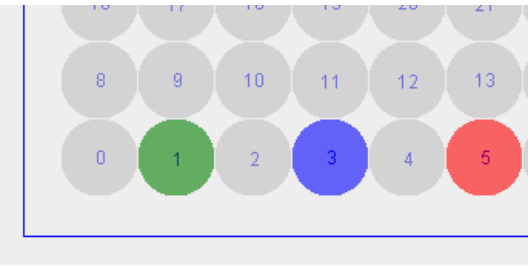


Abbildung 7.5: Felder und Farben

7.7.1 Beispiel Brettspiel

Felder sind besonders gut geeignet, um die Informationen für Spiele zu speichern. Dabei gibt es verschiedene Möglichkeiten:

- für jeden Spieler und jede Spielerin gibt es ein eigenes Feld mit den Positionen der Spielsteine.
- der Spieler-Index wird als zusätzliche Dimension in ein einziges Feld aufgenommen.
- ein Feld repräsentiert das Spielbrett und die Spielsteine werden dort eingetragen.

Welche Struktur am besten passt, lässt sich schwer allgemein beantworten. Beschränken wir uns auf Spiele mit nur zwei Spielern und quadratische Bretter wie zum Beispiel Dame, Vier gewinnt oder Go, so ist ein zweidimensionales Feld als Repräsentation des Spielbretts eine gute Lösung. Die Spielsteine werden dann – je nach Datentyp des Feldes – als Zahlenwerte oder als Zeichenketten in die einzelnen Zellen eingetragen. In Listing 7.1 ist der Aufbau eines 8×8-Spielfeldes mit der Anfangsstellung des Dame-Spiels gezeigt. Dabei stehen die Zahlen 1 und 2 für die Steine der beiden Spieler (noch ohne Dame). In zwei verschachtelten for-Schleifen wird das Spielfeld mit schwarz-weißen Hintergründen und den Spielsteinen dargestellt (Bild 7.6).

Frage 7.3. Warum sind die Positionen bei der Definition des Feldes `brett` gedreht?

Intern sind die N-dimensionalen Felder aus N-1-dimensionalen Feldern aufgebaut. Daher läuft der innerste (letzte) Index am schnellsten. Wir wollen aber ein xy-System benutzen. Daher müssen wir beim Anlegen die Dimensionen in umgekehrter Reihenfolge eingeben.

Frage 7.4. Wäre `int brett[N][N]` nicht eleganter und besserer Code?

Listing 7.1: Dame-Spiel

```
const int N=8;
int i, j;

int brett[8][8] = {
        {1, 0, 1, 0, 0, 0, 2, 0 },
        {0, 1, 0, 0, 0, 2, 0, 2 },
        {1, 0, 1, 0, 0, 0, 2, 0 },
        {0, 1, 0, 0, 0, 2, 0, 2 },
        {1, 0, 1, 0, 0, 0, 2, 0 },
        {0, 1, 0, 0, 0, 2, 0, 2 },
        {1, 0, 1, 0, 0, 0, 2, 0 },
        {0, 1, 0, 0, 0, 2, 0, 2 }
    };

for( i=0; i<N; i++ ) {
    for( j=0; j<N; j++ ) {
        if( brett[i][j] == 1 ) {
            farbe2( i, j, BLUE );
            symbolGroesse2( i, j, 0.4 );
        } else if( brett[i][j] == 2 ) {
            symbolGroesse2( i, j, 0.4 );
            farbe2( i, j, YELLOW );
        } else {
            form2( i, j, "none" );
        }
        if( (i + j ) % 2 == 0 ) {
            hintergrund2( i, j, 0xafafaf );
            } else {
                hintergrund2( i, j , WHITE );
            }
    }
}
```

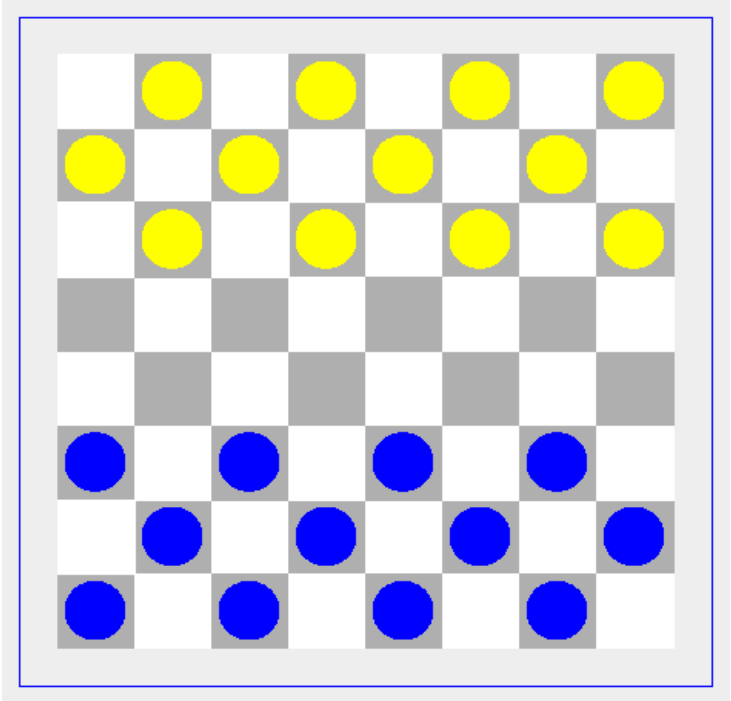


Abbildung 7.6: Ausgangsstellung für Dame Spiel

Ja, das wäre es aber der Compiler (oder zumindest manche) haben damit ein Problem (Fehlermeldung bei gcc `error: variable-sized object may not be initialized`). Man kann allerdings `N` statt als Variable mit

```
#define N 8
```

als Makro setzten. Dann wird vorm Kompilieren `N` überall durch 8 ersetzt.

Kapitel 8

Interaktivtät

Mit den bisherigen Kapiteln haben wir eine gute Grundlage für das Programmieren in C geschaffen. Jetzt ist ein guter Punkt, die eingeschränkte Code-Eingabe in BoS zu verlassen und zu einer richtigen Entwicklungsumgebungen zu wechseln. Für unsere Zwecke ist jede der gebräuchlichen Umgebungen ausreichend. Die weiteren Beispiele wurden mit Microsoft Visual Studio erstellt. Im wesentlichen lassen sie sich aber leicht auf andere Umgebungen übertragen. Auf spezifische Details wird hingewiesen.

Frage 8.1. Kein BoS mehr?

Doch, wir werden bald auf BoS zurück kommen, es dann aber nur noch als Benutzeroberfläche verwenden.

8.1 Erstes komplettes C-Programm

Bisher hat die BoS-Umgebung unsere Code-Schnipsel intern zu einem vollständigen C-Programm zusammen gebaut. Ein

```
printf( "Hallo Friedberg" );
```

genügt und die Ausgabe erschien im unteren Fenster. Demgegenüber zeigt Listing 8.1 ein entsprechendes Programm erstellt als Konsolanwendung in Visual Studio. Dabei bedeuten die einzelnen Teile

Zeilen 1 und 2 automatisch generierter Kommentar, das Programm hat den wenig originellen Namen `prog1`. Die Endung `cpp` steht für C++. Eigentlich ist Visual Studio für die Sprache C++ ausgelegt. Da C aber eine Untermenge von C++ ist, können wir alle Möglichkeiten von C++ ignorieren und C-Programme bearbeiten.

Zeile 4 Mit `include` werden so genannte Headerdateien (daher die Endung `h`) eingebunden. Die hier angegebene Datei `stdafx.h` ist eine Besonder-

Listing 8.1: Komplettes C-Programm

```
1  // Prog1.cpp : Definiert den Einstiegspunkt für die Konsolenanwendung.
2  //
3
4  #include "stdafx.h"
5
6
7  int _tmain(int argc, _TCHAR* argv[])
8  {
9      printf( "Hallo Friedberg" );
10
11     getchar();
12     return 0;
13 }
```

heit von Visual Studio. Sie selbst enthält die Standard-Dateien von C wie `stdio.h`.

Zeile 7 Ein C-Programm muss eine Funktion `main()` enthalten. Die Ausführung startet mit dieser Funktion. `_tmain` ist wiederum ein Visual Studio Besonderheit, die aber nur für die Codierung der Sonderzeichen wichtig ist. Man kann den Namen auch in `main` ändern. Die Funktion `main` erhält Argumente und gibt auch einen Zahlenwert zurück. Die Argumente kann man beim Aufruf des Programms in einer Konsole mitgegeben. Der Rückgabewerte soll informieren, ob das Programm korrekt endete. Dies kann man man auswerten, wenn das Programm in einem Skript ausgeführt wird.

Zeilen 8 und 13 die geschweiften Klammern begrenzen den Programmrumpf.

Zeile 9 endlich unsere Ausgabe

Zeile 11 startet man das Programm in Visual Studio, wird das zugehörige Fenster nach Programmende sofort geschlossen. Um dies zu verhindern, warten wir mit `getchar()` (dazu gleich mehr) auf eine Tastatureingabe. Erst nach einer beliebigen Tastatureingabe geht das Programm mit der nächsten Anweisung weiter.

Zeile 12 die Anweisung `return` beendet eine Funktion und kann einen Wert zurück geben. In unserem Fall bedeutet die 0 per Konvention *alles in Ordnung, kein Fehler*. Mit dem Rücksprung aus `main` endet auch die Ausführung des Programms.

Listing 8.2: printf und scanf

```
main() {
    int anzahlWochen;
    int anzahlStunden;

    printf( "Wochen: ");
    scanf("%d", &anzahlWochen );
    printf( "Stunden: ");
    scanf("%d", &anzahlStunden );
    printf("Gesamt Stunden = %d\n",
         anzahlWochen * anzahlStunden);
}
```

8.2 Eingabe

Das Gegenstück zu `printf` zum Einlesen ist `scanf` (**scan f**ormated). Die Funktion liest Werte von der Tastatur ein. Im Prinzip funktioniert es genau wie bei `printf`. Das erste Argument ist eine Zeichenkette mit Formatspezifikationen. Diese Zeichenketten sollte keinen Text enthalten. Sollte hier Text stehen, wird er nicht etwa ausgegeben sondern vielmehr in der Eingabe erwartet. Leerzeichen in der Formatspezifikation sind unkritisch und werden ignoriert. Da die genaue Funktionsweise von `scanf` nicht immer ganz offensichtlich ist, hat es sich bewährt, zur Sicherheit die eingelesenen Werte nochmals auszugeben. Dann lässt sich zumindest schnell erkennen, ob die Eingabe richtig funktioniert hat.

Nach der Formatspezifikation folgt eine Liste mit Variablen, in die die gelesenen Werte abgelegt werden. Der wesentliche Unterschied zu `printf` ist, dass jetzt im Verlauf von `scanf` der Inhalt dieser Variablen verändert wird. Daher – genaueres dazu später – muss vor jedem Variablennamen ein `&`-Zeichen gesetzt werden. Mit `scanf` ist es möglich, interaktive Programme zu schreiben. So kann man die festen Vorgaben durch flexible Eingaben ersetzen. Listing 8.2 zeigt ein einfaches Beispiel. Der Code ist in standardisierter Form ohne Visual Studio spezifische Besonderheiten. Zur Verwendung in Visual Studio muss auch `scanf` durch `scanf_s` (für *scanf safe*) ersetzt werden.

Bei der Ausführung werden die benötigten Werte abgefragt. Daraus wird dann der gesuchte Wert berechnet und schließlich ausgegeben. Wir haben hier ein ganz einfaches Beispiel des grundlegenden EVA-Prinzips:

- Eingabe
- Verarbeitung
- Ausgabe

Bei der Eingabe eines Wertes vom Typ `float` bzw. `double` kann man `e`, `f` oder `g` einsetzen. Dabei ist man nicht an das spezielle Format gebunden. Auch bei beispielsweise `%f` kann man den Wert mit Exponenten angeben. Wichtig ist, dass `scanf` standardmäßig nur einen `float` Wert, d. h. den kleinsten der Gleitkommatypen, erwartet. Will man eine Variable vom Typ `double` einlesen, so muss man unbedingt `%lf` angeben.

Mit dem Formatkennzeichner `%c` kann man mit `scanf` einzelne Zeichen eingeben. Bei der Eingabe wartet `scanf` auf ein Zeilenende (Eingabetaste). Danach werden alle eingegebenen Zeichen einzeln abgearbeitet – einschließlich des Zeilenendes. Man benötigt daher mehrere `scanf` Aktionen, um eine komplette Zeile einzulesen. Neben dieser formatierten Ein- und Ausgabe gibt es auch Funktionen für unformatierten Ein- und Ausgabe. Dabei wird die Zeichenkette nicht interpretiert sondern einfach Zeichen für Zeichen übertragen. Hierzu stehen folgende Funktionen (Makros) zur Verfügung:

- `getchar(void)` liest ein einzelnes Zeichen und übergibt es als Rückgabewert. Wie bei `scanf` muss die Eingabe mit `<RETURN>` beendet werden.

- `getch(void)` und `getche(void)` lesen ein einzelnes Zeichen direkt von der Tastatur und sind in `conio.h` deklariert. Jeder Tastendruck wird sofort verarbeitet. Bei `getche()` – entspricht `getch()` mit Echo – wird das Zeichen auch auf dem Bildschirm angezeigt. Für beide gibt es neuer Namen mit einem zusätzlichen Unterstrich: `_getch` und `_getche`. Eventuell kennt der Compiler die alten Namen nicht mehr.

- `putch( char c )` ist das Gegenstück zu `getchar()` für die Ausgabe eines einzelnen Zeichens auf den Bildschirm.

Das folgende Beispiel zeigt eine Schleife, in der jeweils ein Zeichen eingelesen und direkt wieder ausgegeben wird.

```
#include <stdio.h>
#include <conio.h>
main() {
    unsigned char c;
    printf( "Bitte Zeichen eingeben, x beendet die Abfrage\n");
    do {
        c = _getch();
        printf( "%2.2x %d %c\n", c, c, c );
    } while( c !='x' );
    printf( "Ende, Taste druecken zum Schliessen des Fensters\n");
    getchar();
    return 0;
}
```

Frage 8.2. Was passiert wenn man `scanf` in einem BoS-Schnipsel verwendet?

Das entsprechende Programm wird generiert und auch gestartet. Dann wartet der Prozess (die Ausführung des Programms) ewig auf eine Eingabe, die nicht erfolgen kann. Der Prozess muss dann per Befehl beendet werden.

8.3 Interaktion mit BoS

Wie versprochen können wir weiterhin BoS zur Darstellung nutzen. Die Funktionsnamen bleiben die gleichen, nur die interne Funktionsweise ändert sich. BoS arbeitet nun wie ein Web-Server. Er wartet auf Anfragen die über das Netzwerk kommen, führt die entsprechenden Aktionen aus und sendet eine Antwort zurück. Damit das funktioniert, muss man im C-Projekt die passenden Bibliotheken und Header-Dateien einbauen. Die technischen Details dazu hängen vom Betriebssystem und der Entwicklungsumgebung ab. Daher gehen wir an dieser Stelle nicht darauf ein, entsprechende Informationen sind in den Webseiten hinterlegt. Wenn alles gut vorbereitet ist, können wir unsere Code-Schnipsel recyclen und in die entsprechende Programm-Hülse einfügen. So wird mit

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    farbe( 12, RED );

    return 0;
}
```

Feld 12 rot gefärbt. Für die weiteren Tests wollen wir nicht ständig neue Projekte anlegen. Daher verlagern wir die Funktionalität in unsere erste eigene Funktion `test1()`. In `main` rufen wir nur einmal diese Funktion auf. Das Ergebnis ist dann

```
#include "stdafx.h"

void test1()( {
       farbe( 12, RED );
}

int _tmain(int argc, _TCHAR* argv[])
{
    test1();
    return 0;
}
```

Nach diesem Prinzip können wir weitere Funktionen einfügen und müssen dann nur den Namen in `main` anpassen. Im folgenden sind zur besseren Übersicht nur noch diese kleinen Funktionen angegeben. Die Ansteuerung über eine eigene Anwendung eröffnet neue Möglichkeiten. So kann man den zeitlichen Ablauf steuern. Mit der Standard-Funktion `Sleep()` gibt man an, dass die Ausführung für die angegebene Zeit (in Millisekunden) angehalten wird. Als Beispiel für eine einfache Animation werden in

```
void test2() {
    int i;
    for( i=0; i<40; i++ ) {
        farbe( i, BLUE );
        Sleep( 400 );
    }
}
```

die ersten 40 Felder Blau gefärbt, wobei nach jedem Feld eine Pause von 20 ms eingelegt wird. Insgesamt besteht damit die Anwendung aus zwei Teilen

1. dem Programm, das aus dem C-Code erzeugt wurde und in einem Konsolfenster läuft

2. der BoS-Anwendung

die miteinander über eine Netzwerk-Schnittstelle kommunizieren. Die Steuerung liegt beim Programm während BoS sich um die Darstellung kümmert. An beiden Enden kann man Benutzereingaben vorsehen. Zunächst kann das Programm über `scanf` oder die anderen Eingabemöglichkeiten Informationen abfragen. In einer entsprechenden Erweiterung

```
void test2a() {
    int i, anzahl;
    loeschen();
    printf("Wie viele Felder? ");
    scanf_s("%d", &anzahl );
    for( i=0; i<anzahl; i++ ) {
        farbe( i, BLUE );
        Sleep( 200 );
    }
}
```

wird die Anzahl der zu färbenden Felder abgefragt. Das funktioniert aber die Aufteilung Eingabe und Darstellung auf zwei Fenster ist nicht benutzerfreundlich. Daher gibt es in BoS auch die Möglichkeit für Eingaben. Dazu muss man im Menü *Welt* den Punkt *interaktiv* auswählen. Dann wird am unteren Rand ein Eingabefeld eingeblendet. Dort kann man beliebigen Text eingeben. Betätigt

man den Knopf *senden*, so wird der Inhalt in einen internen Speicher übernommen. Neben dem Knopf wird die Anzahl der bereits übernommenen Eingaben angezeigt. Mit der Funktion `abfragen()` kann das C-Programm diese Eingaben abholen. Ein minimales Beispiel ist

```
void test3() {
    for(;;) {
        char *a = abfragen();
        if( strlen( a ) > 0 ) {
            printf( "Nachricht: %s\n", a );
        } else {
            Sleep( 100 );
        }
    }
}
```

Die Funktion `abfragen()` gibt eine Zeichenkette zurück. Liegt keine neue Nachricht vor, so hat diese Zeichenkette die Länge 0. Im Beispiel wird bei Vorliegen einer neuen Nachricht diese ausgeben. Anschließend wartet das Programm für 100 ms mit der nächsten Anfrage. Diese Verfahren – in regelmäßigen Abständen nach neuen Nachrichten fragen – bezeichnet man allgemein als Polling. Es ist nicht besonders effizient, aber erlaubt eine recht lose Koppelung der beteiligten Anwendungen. Neben den Texteingaben reagiert BoS auch auf Mausklicks. Ein Klick auf ein Feld `n` wird als Nachricht `# n x y` eingetragen. Dabei steht `n` für den Zählindex während `x` und `y` Spalte und Zeile enthalten. Unser letztes Beispiel in diesem Abschnitt enthält eine zusätzlich Prüfung der Nachricht.

```
void test4() {
    int feld;
    for(;;) {
        char *a = abfragen();
        if( strlen( a ) > 0 ) {
            printf( "Nachricht: %s\n", a );
            if( a[0] == '#' ) {
                sscanf_s( a, "# %d", &feld );
                farbe( feld, 0xff00 );
            }
        } else {
            Sleep( 100 );
        }
    }
}
```

Handelt es sich um einen Klick, so wird mit `sscanf_s` /(zur Erinnerung: wie `scanf_s` aber aus einer Zeichenkette) der Feldindex gelesen und das entsprechen-

de Feld wird gefärbt.

8.3.1 Beispiel Brettspiel

Die Steuerung per Maus kann wie folgt in unser Dame-Spiel eingebaut werden:

```
int feld, ix, iy;
int spieler = 1;
for(;;) {
    char *a = abfragen();
    if( strlen( a ) > 0 ) {
        if( a[0] == '#' ) {
            sscanf_s( a, "# %d %d %d", &feld, &ix, &iy );
            if( brett[ix][iy] != 0 ) {
                form2( ix, iy, "none" );
                brett[ix][iy] = 0;
            } else {
                brett[ix][iy] = spieler;
                farbe2( ix, iy, BLUE );
                symbolGroesse2( ix, iy, 0.4 );
                form2( ix, iy , "c" );
            }

        }
    } else {
        Sleep( 100 );
    }
}
```

Das ist noch kein korrektes Spiel aber man kann bereits Steine entfernen oder einfügen.

Kapitel 9

Funktionen

Ein wesentlicher Gedanke der strukturierten Software-Entwicklung ist die Aufteilung einer komplexen Aufgabe in einzelne Teilaufgaben. Programmtechnisch entspricht dies der Aufteilung des Programms in mehrere Untereinheiten oder Module. In C ist die entsprechende Untereinheit eine Funktion. Eine Funktion ist ein eigenständiger Verarbeitungsblock mit definierten Eingangs- und Ausgangswerten. Vorteile von Funktionen sind:

- Berechnungen, die mehrfach benötigt werden, brauchen nur einmal programmiert zu werden
- Kapselung von Funktionalitäten (*black box* Prinzip, *information hiding*)
- verbesserte Testmöglichkeiten
- Wiederverwendbarkeit in anderen Projekten

In verschiedenen Programmiersprachen gibt es unterschiedliche Bezeichnungen und Konzepte für Module. Man kann grob unterscheiden:

- Funktionen verfügen über Argumente und Rückgabewerte
- Prozeduren oder Unterprogramme (*subroutines*) haben keinen Rückgabewert
- Methoden sind in objektorientierten Sprachen Funktionen, die zu einer Klasse gehören
- Makros sind Quellcode-Bausteine, bei denen vor dem Kompilieren ein gemeinsames, in der Regel längeres Stück Code an die entsprechend markierten Stellen eingefügt werden (Makro-Ersetzung).

Bei der Entwicklung von C wurde Wert auf einfachen und effiziente Einsatz von Funktionen gelegt. Größere Programme bestehen nahezu immer aus einer Anzahl

von (vielen) kleinen Funktionen. Diese Funktionen können auf mehrere Dateien verteilt sein. Mit Compiler und Linker werden sie zu einer gemeinsamen Anwendung zusammen gebaut.

9.1 Funktionsdefinition

Eine Funktion ist ein Block von Definitionen und Anweisungen mit einem Namen sowie Eingangswerten und einem Ausgangswert. Funktionen stehen auf der ersten Ebene, d. h. die Definitionen können nicht ineinander geschachtelt werden. Allerdings ist es sehr wohl möglich, dass eine Funktion eine weiter Funktion aufruft. Die allgemeine Form ist

```
rückgabe-typ   funktions_name( argumente ) {
    ...
}
```

Zunächst wird spezifiziert, welchen Typ von Variable die Funktion zurück gibt. Anschließend folgt der Name der Funktion und dann in runden Klammern die Liste der Eingangswerte (formale Argumente). Jedes Argument besteht aus dem Typ und einem Namen. Mehrere Argumente werden durch Komma getrennt. Die verkürzte Schreibweise wie etwa `(int i, j)` ist nicht erlaubt, jedes Argument benötigt eine eigene Typangabe: `(int i, int j)`.

Die eigentliche Funktion – Vereinbarungsteil und Ausführungsteil begrenzt durch geschweifte Klammern – kommt danach. Für die Rückgabe steht der Befehl `return ausdruck;` zur Verfügung. Bei Erreichen eines `return` Befehls wird die Bearbeitung der Funktion beendet. Der Ausdruck wird ausgewertet und an das aufrufende Programm zurück gegeben. Innerhalb einer Funktion können mehrere `return` Anweisungen stehen, um alternative Rücksprünge zu realisieren.

Beispiel 9.1. Berechnung der Fakultät:

```
int fakultaet( int wert ) {
    int result = 1;
     // Argument ausserhalb des Wertebereiches ?
    if( wert < 0 ) return -1;
    while( wert > 1 ) {
        result *= wert;
        --wert;
   }
   return result;
}
```

Das Beispiel berechnet für den Eingangswert `wert` die Fakultät und gibt das Ergebnis zurück. Um eine Funktion aufzurufen, wird sie über den Namen und mit

entsprechenden Argumenten angesprochen. Eine Funktion liefert einen Wert zurück. Das aufrufende Programm (Hauptprogramm) kann diesen Wert ignorieren oder in einem Ausdruck als rvalue weiter verwenden.

Beispiel 9.2. Verwendung der Methode Fakultät:

```
/* gibt den Wert z! aus */
printf("%d! = %d\n", z, fakultaet( z ) );
/* legal aber wenig wirksam */
fakultaet( 2 * 3 );
```

Bei dem Aufruf wird der Ausdruck in der Klammer ausgewertet und an die Funktion übergeben. Selbst wenn man direkt eine Variable als Argument angibt, wird nur der Wert bzw. Inhalt der Variablen übergeben (*call by value*). Daher erfolgen alle Änderungen in der Funktion an einem Argument nur an der lokalen Kopie. Im obigen Beispiel bleibt etwa der Wert der Variablen `z` unverändert.

Grundsätzlich hat einen Funktion keinen Zugriff auf die Variablen im Hauptprogramm. Genauso wenig kann man mit der Sprunganweisung `goto` zwischen Hauptprogramm und Funktion springen. Aufgrund der strikten Trennung gibt es auch keine Konflikte bei gleichen Namen für Variablen. Variablen innerhalb einer Funktion haben nur die Lebensdauer eines Funktionsaufrufs. Sie werden beim Aufruf angelegt und nach Ende der Funktion wieder gelöscht (*automatic variable*). Dieser Vorgang wird für jeden neuen Aufruf wiederholt. Möchte man, dass eine Variable „länger lebt“ und bei einem weiteren Aufruf noch ihren alten Wert hat, so muss man dies mit dem Kennwort `static` bei der Definition angeben. Statische Variablen bleiben nach Ende einer Funktion erhalten und behalten ihren Wert auch über mehrere Aufrufe.

```
void funktion1( ) {
    int v1 = 0; /* temporaere variable */
    static int v2 = 0; /* statische variable */

    ++v1; ++v2;
    printf( "v1 = %d v2 = %d\n", v1, v2);
}
main()
{
    int i;
    for( i=0; i<4; i++ ) funktion1();
}
```

ergibt

```
v1 = 1   v2 = 1
v1 = 1   v2 = 2
```

```
v1 = 1   v2 = 3
v1 = 1   v2 = 4
```

Das Beispiel zeigt zwei neue Elemente:

- die Argumentliste kann leer sein
- eine Funktion ohne Rückgabewert hat den Typ `void` (engl. leer, nichtig). Eine solche Funktion endet entweder mit einem `return` ohne Wert oder an der schließenden Blockklammer.

9.2 Deklaration

Wenn die Funktion in der Datei vor dem Aufruf steht, erkennt der Compiler selbständig den Typ und die Art und Anzahl der Argumente. Die Definition beinhaltet bereits die Deklaration. Andernfalls – oder falls die Funktion in einer anderen Datei steht – muss man die Funktion deklarieren. Die Syntax ist analog zur Syntax der Definition. Für die Funktion `fakultaet` beispielsweise schreibt man den Funktionsprototyp

```
int fakultaet( int i );
```

um dem Compiler mitzuteilen, dass es

- eine Funktion mit dem Namen `fakultaet` gibt
- der Rückgabewert vom Typ `int` ist
- ein Argument vom Typ `int` erwartet wird

Der Name der Variablen im Argument ist willkürlich und kann sogar weggelassen werden:

```
int fakultaet( int );
```

Wenn man allerdings Namen vergibt, dann müssen sie eindeutig sein, d. h. jeder Name darf nur einmal benutzt werden. Es gibt dabei keine Konflikte mit den Variablennamen im Programm. Sinnvoll ist es, aussagekräftige Namen zu verwenden, am besten die gleichen wie bei der Definition. Für eine Funktion ohne Argument schreibt man `void` anstelle der Argumentliste. Aus Gründen der Kompatibilität zu frühen C Versionen kann man Teile der Deklaration auch weglassen. Standardmäßig wird dann der Rückgabetyp `int` angenommen. Es ist aber in jedem Fall besser, den Typ und die Argumente explizit anzugeben. Man kann die Deklarationen relativ frei innerhalb des Programms platzieren. Üblich und übersichtlich ist es, die Deklarationen an den Anfang der Datei (nach den `#include` Anweisungen) zu schreiben.

9.2.1 Beispiel Berechnung von Potenzen

In dem Beispiel wird eine Funktion zur Berechnung von a^b implementiert. Die Funktion steht in einer eigenen Datei. Zum Testen wird ein Programm mit freier Eingabe der Argumente benutzt. Die eigentliche Anwendung erstellt die Wertetabelle für zwei Potenzreihen.
Datei power.c:

```
/* funktion zur Berechnung der Potenz a hoch b
 * Einschränkung: b >= 0
 */
int power( int base, int exponent ) {
    int i;
    int result = 1;
    for( i=0; i<exponent; i++ ) result *= base;
    return result;
}
```

Datei test.c:

```
/* Testprogramm für Funktion power */
#include <stdio.h>

int power( int base, int exponent );
main() {
    int i, j;

    for( ;; ) {
        printf("Basis, Exponent: ");
        scanf( "%d%d", &i, &j);
        printf("ergibt: %d\n", power(i,j) );
    }
}
```

Kompilieren und Linken mit dem Befehl, anschließend ausführen:

```
>cl test.c power.c
>test
```

Durch den Befehl werden beide Quelldateien kompiliert und zusammen „gelinkt“. Gibt man nur das Hauptprogramm an, erhält man folgende Fehlermeldung:

```
error LNK2001: Nichtaufgeloestes externes Symbol _power
fatal error LNK1120: 1 unaufgeloeste externe Verweise
```

Die Funktion `power` wird nicht gefunden und damit bricht der Linker ab. Die eigentliche Anwendung in `haupt.c`:

```
#include <stdio.h>

int power( int base, int exponent );
main() {
    int i, f1, f2;

    for( i=-10; i<=10; i++ ) {
        f1 = 3 * power(i, 3) - 5 * power(i, 2);
        f2 = power( i, 4 );
        printf( "i=%3d f1=%6d f2=%6d\n", i, f1, f2);
    }
}
```

9.3 Rekursion

In der Übung 4.5 wird mit den Fibonacci-Zahlen ein Beispiel für eine rekursive Definition gegeben. In der Bestimmungsgleichung

$$i_n = i_{n-2} + i_{n-1}$$

ist das n-te Element als Summe seiner beiden Vorgänger definiert. Um den Wert von i_n zu bestimmen ist es gemäß dieser Definition notwendig, die beiden Vorgänger zu bestimmen. Diese wiederum beruhen auf ihren eigenen Vorgängern. Auf diese Art und Weise ist es notwendig, immer weiter zurück zu gehen bis man schließlich auf einen der beiden Startwerte

$$i_1 = 1,\ i_2 = 1$$

trifft. Zur Lösung der Aufgaben wird man allerdings eher den umgekehrten Weg gehen. Ausgehend von den beiden Startwerten wird die Folge bis zu dem gesuchten Wert berechnet. Allerdings ist mit C durchaus auch möglich, die rekursive Bestimmungsgleichung direkt in einen rekursiven Programmablauf umzusetzen. Im folgenden wird das allgemeine Konzept vorgestellt.

Erfahrungsgemäß fällt der Einstieg in die Programmierung rekursiver Funktionen nicht ganz leicht. Betrachten wir daher zunächst als ganz einfaches Beispiel die Berechnung der Summe der ersten 50 Zahlen. Mit der Funktion $S(N)$ als Summe der ersten N Zahlen erhalten wir

$$\begin{aligned} S(1) &= 1 \\ S(2) &= S(1) + 2 \\ S(3) &= S(2) + 3 \\ &\vdots \quad\quad \vdots \\ S(50) &= S(49) + 50 \end{aligned}$$

Dies entspricht unmittelbar der Realisierung durch eine Schleife in der Art

```
int i, S=0;

for( i=1; i<=50; i++ ) {
    S = S + i;
}
printf("%d\n", S );
```

Der gesuchte Wert wird durch Addition aller Zahlen von 1 bis 50 berechnet. Formal lässt sich die Reihenfolge der Berechnung genauso gut umkehren. Wenn wir zunächst so tun, als wäre $S(49)$ bekannt, dann können wir einfach

$$S(50) = S(49) + 50$$

schreiben. Mit dem gleichen Argument kann $S(49)$ auf $S(48)$ zurück geführt werden. Insgesamt erhält man damit:

$$\begin{aligned} S(50) &= S(49) + 50 \\ S(49) &= S(48) + 49 \\ S(48) &= S(47) + 48 \\ &\vdots \quad\quad \vdots \\ S(2) &= S(1) + 2 \\ S(1) &= 1 \end{aligned}$$

Da die Berechnung vom Ende her beginnt, spricht man von Rekursion (lat. recurrere *zurücklaufen*). Demgegenüber wird das schrittweise Berechnen ab dem Anfang als Iteration (lat. iterare *wiederholen*) bezeichnet.

Wir haben gesehen, dass eine Funktion mit lokalen Kopien der Argumenten arbeitet und alle Variablen (außer den mit `static` deklarierten) temporärer Natur sind. Daher ist es möglich, dass eine Funktion sich selbst – oder besser gesagt eine neue Instanz von sich selbst – wieder aufruft. Bei jedem Aufruf werden neue lokale Variablen angelegt, so dass es zu keinen Zugriffskonflikten kommt. Rekursion lässt sich daher in C leicht realisieren. Dazu programmieren wir für die Addition der Zahlen eine Funktion, die entweder bei $n = 1$ den Wert 1 zurück gibt oder einen Rekursionsschritt ausführt:

```
int S( int n ) {
    if( n == 1 ) return 1;
    return S(n-1) + n;
}
```

Der gesuchte Wert kann dann einfach mit

```
printf("%d\n", S(50) );
```

ausgegeben werden. In dieser Form ist die Schleife verschwunden. Stattdessen steckt die Logik in der Methode. Die Methode ruft sich selbst mit einem um 1 kleineren Parameter immer wieder auf, bis irgendwann der Wert 1 erreicht ist. Dann wird rückwärts die Kette abgearbeitet.

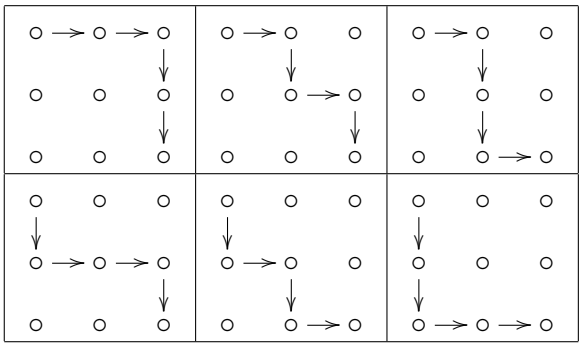
Diese Beispiel zeigt, wie ein Problem sowohl iterativ als auch rekursiv gelöst werden kann. In diesem Fall bietet die rekursive Lösung außer der eleganten Formulierung keinen Vorteil. Es ist sogar davon auszugehen, dass sie aufgrund der vielen Methodenaufrufe mehr Rechenzeit und Speicherplatz benötigt. Das klassische Beispiel für die Rekursion ist das Berechnen der Fakultät:

```
int fakultaet_r( int wert ) {
    if( wert == 0 || wert == 1) return 1;
    else return fakultaet_r(wert - 1) * wert;
}
```

Wenn das Argument 0 oder 1 ist, wird direkt $0! = 1! = 1$ zurückgegeben. Ansonsten wird der aktuelle Wert mit der Fakultät des um 1 kleineren Wertes gemäß $n! = n * (n-1)!$ berechnet. Beginnend mit einem positiven Wert wird diese Rekursion wiederholt, bis schließlich der immer wieder verminderte Wert auf 1 gefallen ist.

9.3.1 Beispiel Wegesuche

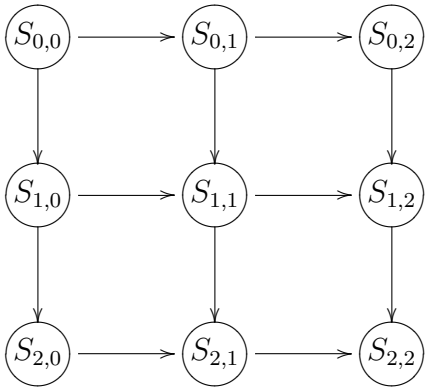
Betrachten wir ein komplexeres Beispiel. Angenommen, Sie wohnen in einer Stadt mit einem regelmäßigen, rechteckigen Straßenmuster (z. B. im Stadtteil Manhattan von New York). Wohnung und Arbeitsplatz liegen dabei auf den gegenüber liegenden Ecken eines Quadrates mit N Blocks. Sie möchten gerne die Entfernung auf möglichst vielen verschiedenen Wegen zurück legen – allerdings ohne Umwege. Nun stellt sich die Frage, wie viele derartige Wege es gibt. Etwas formaler gesprochen: vorgegeben ist ein Raster mit $N \times N$-Feldern. Wie viele mögliche Wege führen von einer Ecke zu der diagonal gegenüber liegenden Ecke? Die Wege sollen nur horizontal oder vertikal verlaufen und keine Umwege enthalten. Im Fall $N = 2$ gibt es insgesamt 6 Möglichkeiten:



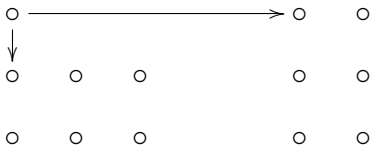
Allgemein gilt für die Anzahl N_W der Wege die Formel

$$N_W = \binom{2N}{N} = \frac{(2N)!}{N! \times N!}$$

Wir wollen allerdings die Werte per Programm berechnen lassen. Betrachten wir dazu den Fall eines Rasters mit 2×2-Feldern. Im folgenden Bild sind die Eckpunkte mit den abgehenden Möglichkeiten dargestellt.



Dabei bezeichne $S_{i,j}$ die Anzahl der Wege ab dem jeweiligen Punkt bis zum Endpunkt. Mit Ausnahme der Randpunkte gibt es an jedem Punkt zwei Verzweigungen. Der weitere Verlauf hängt nicht von dem bisherigen Weg ab. Daher kann man die weiteren Berechnungen unabhängig betrachten. Mit diesem Argument lässt sich das Hauptproblem – die Anzahl der Wege ab dem Startpunkt – in zwei kleinere Teilprobleme aufspalten. Man betrachtet jeden der beiden Wege ab dem Startpunkt für sich und berechnet die Anzahl der möglichen Wege in dem verbleibenden Rechteck. Folgendes Bild veranschaulicht diese Vorgehensweise:



Auf diese Art und Weise ist die Berechnung von $S_{0,0}$ auf die Summe der beiden Teile $S_{1,0}$ und $S_{0,1}$ vereinfacht. Diese Argument lässt sich verallgemeinern und es gilt für alle außer den Randpunkten die rekursive Beziehung

$$S_{i,j} = S_{i+1,j} + S_{i,j+1}$$

Für die Randpunkte ist die Situation einfach. Von jedem Randpunkt gibt es nur noch einen einzigen Weg – nach unten oder nach rechts, je nachdem ob man sich am rechten oder unteren Rand befindet. Damit kann man zusammenfassend schreiben:

$$S_{i,j} = \begin{cases} 1 & , \text{ wenn } i = n+1 \\ 1 & , \text{ wenn } j = n+1 \\ S_{i+1,j} + S_{i,j+1} & , \text{ sonst} \end{cases}$$

Diese Vorschrift lässt sich unmittelbar als C-Funktion schreiben:

```
#include "stdio.h"

// Anzahl der Wege ab dem Punkt i,j in einem n x n Feld
long S( int i, int j, int n ) {
    if( i == n+1 || j == n+1 ) {
        return 1;
    } else {
        return S( i+1, j, n ) + S(i, j+1, n );
    }
}

int main(int argc, char* argv[]) {
    printf("%d \n", S( 0, 0, 2 ) );
    return 0;
}
```

In `main` wird die Funktion mit dem Startpunkt $0,0$ aufgerufen. Intern werden rekursiv die Teillösungen berechnet und addiert. Diese Lösung liefert korrekte Ergebnisse. Allerdings beobachtet man ab etwa $n = 15$ extrem lange Rechenzeiten. Die Lösung ist zwar elegant aber nicht effizient. Das Problem ist, dass jede Teilsumme immer wieder neu berechnet wird. Der Punkt $1,1$ wird beispielsweise auf zwei Wegen erreicht. Daher wird auch die Funktion $S_{1,1}$ zweimal ausgewertet. Bei kleinem N spielt dies praktisch noch keine Rolle, aber mit wachsender Feldgröße führt diese Effekt schnell zu einem dramatischen Anstieg des Rechenaufwandes.

In Bild 9.1 ist die gemessen Rechenzeit bis $n = 16$ dargestellt. Für größere Werte von n liegt das Ergebnis nicht mehr im Bereich von `long`. In der halblogarithmischen Darstellung erkennt man deutlich den exponentiellen Anstieg der Rechenzeit. Bereits bei $n = 15$ benötigt mein Rechner mehr als 10 Sekunden.

Glücklicherweise gibt es einen einfachen Ausweg, um die Mehrarbeit zu vermeiden. Dazu wird lediglich das einmal gewonnene Wissens nicht vergessen sondern für die spätere Wiederverwendung aufgehoben. Der Code zeigt eine entsprechende Lösung:

```
#define MAX 50 // Maximale Größe
long feld[MAX][MAX];

// Anzahl der Wege ab dem Punkt i,j in einem n x n Feld
long S( int i, int j, int n ) {
    if( i == n+1 || j == n+1 ) {
        return 1;
    } else {
        if( feld[i][j] == 0 ) {
          feld[i][j] = S( i+1, j, n ) + S(i, j+1, n );
```

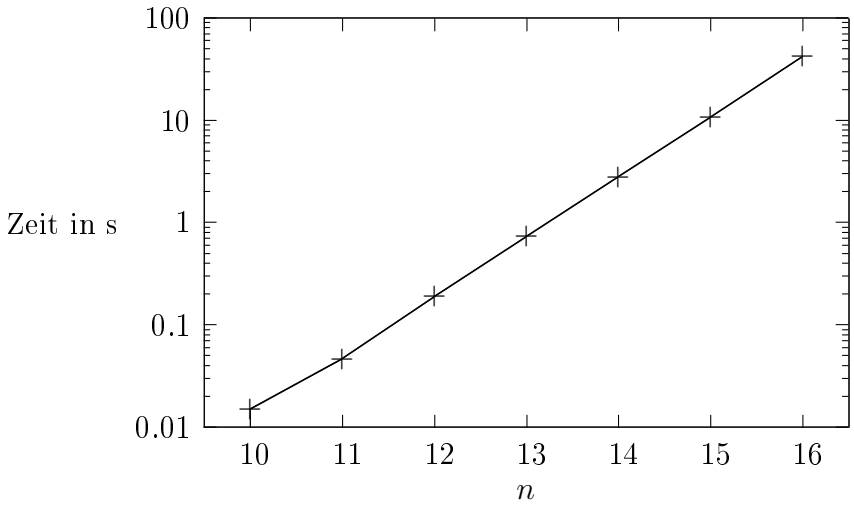


Abbildung 9.1: Gemessene Rechenzeit für rekursive Berechnung der Anzahl der Wege bei $n \times n$ Feldern.

```
        }
        return feld[i][j];
    }
}
```

Sobald ein Wert für $S_{i,j}$ feststeht, wird er in die entsprechende Zelle eines zweidimensionalen Feldes abgelegt. Wenn ein Wert für $S_{i,j}$ benötigt wird, wird zunächst nachgeschaut, ob die zugehörige Zelle bereits belegt ist. Falls ja, wird einfach dieser Wert verwendet. Nur falls die Zelle noch nicht belegt ist, wird eine neue Berechnung notwendig. Mit dieser Erweiterung liefert das Programm auch für größere Felder schnell die Lösung. Auf meinem Rechner liegt die Ausführungszeit unterhalb der Messgenauigkeit.

Das beschriebene Vorgehen ist ein Beispiel für ein allgemeines Verfahren zur Lösung derartiger Probleme, der so genannten *dynamischen Programmierung*. Der Ansatz eignet sich für Probleme, die sich in viele kleine Teilprobleme zerlegen lassen, die wiederum voneinander unabhängig gelöst werden können. Typischerweise werden die Teilergebnisse in Tabellen abgelegt, auf die dann später wieder zurückgegriffen werden kann. Die Speicherung der Rückgabewerten von Funktionen im speziellen nennt man Memoisation (engl. *memoization*).

9.3.2 Schleife oder Rekursion?

Rekursive Lösungen bieten sich immer dann an, wenn man ein komplexes Problem auf ein einfacheres Problem zurückführen kann. Man kann dann den Lö-

sungsansatz direkt als C-Code umsetzen. Die Lösung ist damit elegant und leicht verständlich. Ob die rekursive oder die immer auch mögliche iterative Lösung aufwandsgünstiger ist, muss im Einzelfall betrachtet werden.

Typisch ist diese Vorgehen für Probleme in der Art von Sudoku. Bei einigermaßen anspruchsvollen Sudoku kommt irgendwann der Punkt, wo es keine eindeutige Lösung für das nächste Feld gibt. Man schreibt dann versuchsweise eine der noch möglichen Zahlen in ein freies Feld und führt das Verfahren mit dieser Annahme weiter. Es gibt demnach eine Funktion, die ein teilweise gefülltes Sudoku erhält, einen weiteren Schritt ausführt und dann sich selbst weiter aufruft bis das Problem gelöst ist oder ein Widerspruch auftritt. Ergibt sich ein Widerspruch, so wird dieser Zweig abgebrochen und die nächste Hypothese ausprobiert. Bei der Übergabe für den nächsten Schritt kann entweder eine Kopie des aktuellen Standes verwendet werden oder die Änderung, die sich als falsch erwiesen hat, wird wieder rückgängig gemacht.

9.4 Anmerkungen

In den Übungen hatten wir bereits einige Standard-Funktionen eingesetzt. Im folgenden sind diese nochmals kurz zusammengefasst.

9.4.1 main

`main` ist selbst eine Funktion, die von einer Startprozedur aufgerufen wird. In jeder Anwendung gibt es genau ein `main`. Die vollständige Definition ist

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

d. h. `main` gibt einen int Wert zurück und hat 2 Argumente, deren Syntax wir noch nicht kennen. Nach der Rückkehr aus `main` werden eventuell belegte Ressourcen freigegeben und die Anwendung beendet.

9.4.2 printf

Für `printf` gilt

```
int printf( char *format, ... )
```

Die Definition wird der Datei `stdio.h` entnommen. Das erste Argument ist die Zeichenkette mit dem Format. Die drei Punkte zeigen an, dass danach eine variable Anzahl von Argumenten kommen kann. Der Rückgabewert gibt an, wie viele Zeichen geschrieben wurden.

9.4.3 scanf

`scanf` ist die Entsprechung von `printf` zum Einlesen von Werten.

```
int scanf( char *format, ... )
```

Der Rückgabewert ist die Anzahl der gelesenen Werte. Die Besonderheit ist, dass `scanf` offensichtlich die Variablen selbst verändern kann. Das `&` vor den Argumenten bewirkt eine andere Art der Argumentübergabe, mit der die Funktion Zugriff auf die Variable hat.

9.5 Übungen

Übung 9.1. Schiefer Wurf
Wirft man einen Körper mit der Geschwindigkeit v und dem Winkel α, so ergibt sich unter idealisierten Bedingungen mit g=9,81 m/s^2 die Wurfweite W gemäß

$$W = \frac{v^2 \sin 2\alpha}{g}$$

1. Schreiben Sie eine Funktion `schieferWurf`, die als Parameter v und α erhält und die daraus berechnete Wurfweite zurück gibt.

2. Verwenden Sie diese Funktion, um bei $v = 5m/s$ die Wurfweite in einer Schleife für $0,01 \leq \alpha < \pi/2$ in Schritten von 0,01 auszugeben.

Übung 9.2. Größter gemeinsamer Teiler
Die folgende Methode berechnet den größten gemeinsamen Teiler (GGT) zweier natürlicher Zahlen $a > b$:

```
int ggt( int a, int b ) {
    while( b > 0 ) {
        int rest = a % b; // Rest bestimmen
        a = b; // Werte tauschen
        b = rest;
    }
    return a;
}
```

Wie sieht eine rekursive Lösung aus?

Übung 9.3. Wie sieht eine rekursive Lösung zur Berechnung der Fibonacci Zahlen aus? Wie beurteilen Sie diese in Bezug auf den Rechenaufwand?

Übung 9.4. Schreiben Sie eine rekursive Lösung zur Berechnung der Quersumme einer Zahl.

Übung 9.5. Türme von Hanoi
Das folgende Programm löst die Aufgabe rekursiv[1].

[1] Beispiel nach Alfred Nussbaumer, `http://noebis.pi-noe.ac.at/javanuss/`

- Wie funktioniert das Programm?
- Wie viele Schritte benötigt man für einen Turm der Höhe n?

```
#include <stdio.h>

int schritte;

void lege(int n, char von, char nach, char zwischen) {
    if (n>0) {
        lege(n-1, von, zwischen, nach);
        printf("%d. Scheibe von %c nach %c\n", n, von ,nach);
        lege(n-1,zwischen, nach, von);
        schritte++;
    }
}

void main() {
    int maxzahl = 7;
    lege(maxzahl, 'a', 'c', 'b');
    printf("----------------------------------------\n");
    printf("Insgesamt %d Schritte\n", schritte );
}
```

Kapitel 10

Sichtbarkeit und Lebensdauer von Variablen

Im Zusammenhang mit Funktionen hatten wir gesehen, dass Variablen nur begrenzt sichtbar sein können und eine eingeschränkte Lebensdauer haben können. Im folgenden sollen diese Zusammenhänge und die in C angewandten Regeln systematisch vorgestellt werden.

10.1 Lokale Variablen

Bisher haben wir die Definitionen als Vereinbarungsteil an den Beginn einer Funktion gestellt. Allgemeiner kann jeder mit geschweiften Klammern begrenzter Block mit einem Vereinbarungsteil beginnen. Beispielsweise kann man in dem Block einer Schleife Variablen definieren:

```
for( ... ) {
    int variableInSchleife = 1;
    ...
}
```

Diese Regel gilt für jeden Block. Grundsätzlich gilt, dass an jeder Stelle an der eine Anweisung steht, auch ein Block stehen kann. Man spricht im Englischen von einem „compound statement“. Die Syntax ist:

```
{
    Vereinbarungsteil
    Ausführungsteil
}
```

In einem Block definierte Variablen sind nur innerhalb des Blocks sichtbar (lokale Variablen). Sie werden zu Beginn des Blocks angelegt und am Ende wieder entfernt. Soll eine Variable bestehen bleiben, muss dies bei der Definition mit dem

Schlüsselwort `static` angegeben werden. Allerdings sind innerhalb des Blocks auch alle Variablen des umgebenden Blocks sichtbar. Wird innerhalb des Blocks ein bereits außen vergebener Name nochmals definiert, überschreibt die innere Definition die äußere. Während der Ausführung des Blocks ist damit die äußere Variable verdeckt. Als Beispiel bewirkt

```
main() {
    int i = 1, j = 1; /* Variable auf der Ebene von main */

    printf( "i = %d j = %d\n", i, j );
    {
        int i = 3; /* Variable innerhalb eines Blocks */
                        /* aeusseres i wird ueberschrieben */
                   /* j ist nach wie vor sichtbar */
        printf( "i = %d j = %d \n", i, j );
    }
    printf( "i = %d j = %d \n", i, j );
}
```

die Ausgabe

```
i = 1  j = 1
i = 3  j = 1
i = 1  j = 1
```

Man sollte die Möglichkeit der Überdeckung sparsam einsetzen. Es dient nicht unbedingt der Klarheit, wenn zwei unterschiedliche Variablen den gleichen Namen tragen. Die Definition mit eventueller Initialisierung wird immer bei Beginn des Blocks ausgeführt, sofern nicht das Attribut `static` benutzt wird. Auch hierzu ein Beispiel:

```
for( i=0; i<3; i++ ) {
    int i1 = 0; /* i1 ist automatic */
    static int i2 = 0; /* i2 ist static */

    printf( "i1 = %d, i2 = %d\n", i1, i2 );
    ++i1; ++i2;
}
```

ergibt

```
i1 = 0, i2 = 0
i1 = 0, i2 = 1
i1 = 0, i2 = 2
```

Die automatische Variable `i1` wird immer wieder neu initialisiert, während die statische Variable `i2` ihren Wert über mehrere Ausführungen behält. Zusammenfassend gilt für lokale Variablen:

- lokale Variablen werden im Vereinbarungsteil am Anfang eines Blocks definiert
- ohne Initialisierung ist der Wert undefiniert
- standardmäßig sind lokale Variablen temporär, ihre Lebensdauer ist die Ausführungszeit des Blocks
- mit `static` kann man lokale Variablen dauerhaft erhalten
- innere Blöcke haben Zugriff auf die Variablen der äußeren Blöcke
- Variablen in einem Block überdecken äußere Variablen mit dem gleichen Namen

10.2 Globale Variablen

Neben den lokalen Variablen gibt es auch globale Variablen. Ähnlich wie Funktionen können globale Variablen in einer Datei definiert werden und in einer anderen benutzt werden. Daher erweitern wir zunächst die Unterscheidung zwischen Definition und Deklaration von Funktionen auf Variablen:

- Durch die Definition wird die Variable „angelegt“, d. h. es wird Speicherplatz reserviert und eventuell wird die Variable auch initialisiert. Jede Variable muss definiert werden und es darf nur eine einzige Definition geben.
- Durch die Deklaration wird die Variable angekündigt, d. h. der Kompiler bekommt die Information, dass es eine Variable mit dem angegebenen Namen und Typ gibt, die irgendwo auch definiert sein muss. Mit der Deklaration ist keine Aktion verbunden, es wird weder Speicher reserviert noch ein Wert zugewiesen. Daher können Deklarationen auch mehrfach vorkommen.

Globale Variablen werden außerhalb von Blöcken definiert. Mit

```
#include ...
int globalAnzahl;
...
main() {
```

wird eine globale Variable mit dem Namen `globalAnzahl` definiert. Die Definition beinhaltet auch die Deklaration. Globale Variablen werden mit dem Wert 0 initialisiert. Eine globale Variable ist in allen Funktionen und Blöcken der gesamten Anwendung ansprechbar. Globale Variablen können zur Kommunikation zwischen verschiedenen Funktionen benutzt werden. Wie im folgenden Beispiel gezeigt,

```
int count; /* eine globale Variable */
void incCount( void ); /* Funktion deklarieren */

main() {
    do {
        printf( "count = %d\n", count );
        incCount();
    } while( count < 5 );
}

/* Erhoehen des zaehlers in Funktion */
void incCount( void ) {
    ++count;
}
```

kann man globale Variablen zur Kommunikation oder zum Datenaustausch zwischen verschiedenen Funktionen nutzen. Wenn die zweite Funktion in einer anderen Datei steht, muss allerdings die Variable bekannt gemacht werden. Dazu gibt es die Deklaration mit dem Schlüsselwort `extern`:

```
extern int globalAnzahl;
```

ist die Deklaration für die oben eingeführte globale Variable. Mit dem Schlüsselwort `extern` wird dem Kompiler mitgeteilt, dass die entsprechende Variable irgendwo (extern) definiert ist und damit benutzt werden kann. Will man sicher stellen, dass eine globale Variable nur von den Funktionen in der Datei, in der sie definiert wird, angesprochen wird, kann die Sichtbarkeit eingeschränkt werden. Mit dem Schlüsselwort `static` vor der Definition wird die Sichtbarkeit auf die aktuelle Datei eingeschränkt.

```
static int globalAnzahl;
```

definiert eine globale Variable, die aber außerhalb der Datei nicht sichtbar ist. Damit vermeidet man eventuelle Konflikte, wenn in einer anderen Datei eine globale Variable mit dem gleichen Namen definiert werden würde. Für die globalen Variablen gilt zusammenfassend:

- globale Variablen werden außerhalb von Blöcken definiert
- ohne explizite Initialisierung haben sie den Anfangswert 0
- globale Variablen werden zu Beginn der Anwendung angelegt und bleiben bis zum Ende bestehen
- standardmäßig sind globale Variablen überall sichtbar
- Um globale Variablen aus einer anderen Datei zu benutzen, muss man sie deklarieren (Schlüsselwort `extern`)

- mit `static` kann man die Sichtbarkeit auf eine Datei beschränken
- lokale Variablen überdecken globale Variablen mit dem gleichen Namen

Globale Variablen bieten eine einfache und komfortable Methode des Datenaustauschs. Insbesondere wenn eine Funktion viele Eingangs- und Ausgangswerte hat, ist der Austausch über gemeinsame Variablen praktisch. Der große Nachteil ist, dass Informationen über das ganze Programm verteilt sind. Funktionen, die globale Variablen benutzen, können nicht so leicht in andere Anwendungen übernommen werden. Die Fehlersuche kann schwierig werden, da man das gesamte Programm durchsuchen muss. Daher sollte man den Einsatz globaler Variablen sorgfältig überlegen. Vorteilhaft sind globale Variablen, um Daten und Zugriffsfunktionen zusammen zu halten.

Beispiel 10.1. globale Variablen

```
// Datei aktien.c:
static int anzahl; /* Anzahl der Aktien */

/* Funktionen zum Kaufen und Verkaufen */
void kaufen( int zahl ) {
    anzahl += zahl;
}
void verkaufen( int zahl ) {
    anzahl -= zahl;
}

/* Rueckgabe des aktuellen Stands */
int wieviele( void ) {
    return anzahl;
}
```

Die globale Variable wird auf die Datei beschränkt. Andere Programme haben durch entsprechende Funktionen Zugriff zum Verändern oder Abfragen.

```
// Datei haupt.c:
void kaufen( int zahl );
void verkaufen( int zahl );
int wieviele( void );

void zeigeDepot( void ) {
    printf( "Anzahl der Aktien: %d\n", wieviele() );
}

main() {
```

```
    int aktien;
    zeigeDepot();
    for( ;; ) {
        printf( "Anzahl neuer Aktien: ");
        scanf( "%d", &aktien );
        kaufen( aktien );
        zeigeDepot();
    }
}
```

10.3 Andere Bezeichner

Neben den Variablen gibt es noch andere Bezeichner innerhalb von C Programmen. Wir haben bis jetzt Funktionen und Sprungmarken kennen gelernt. Diese Namen werden in unterschiedlichen Bereichen verwaltet (Namensräume, engl. *name spaces*). Eine Variable kann daher ohne Konflikt den gleichen Namen haben wie eine Funktion oder eine Sprungmarke. Für die verschiedenen Namensräume gelten ähnliche Regeln. Beispielsweise gibt es auch bei Funktionen die Spezifikation `static`, womit der Name der Funktion nur innerhalb der aktuellen Datei sichtbar ist. Allerdings sind Sprungmarken nicht nur in dem Block sondern in der ganzen Funktion sichtbar.

Kapitel 11

Spiele mit Zuständen

In Abschnitt 8.3 hatten wir bereits die Darstellung eines Dame-Spiel behandelt. Wir hatten auch gesehen, wie man beim Klicken auf ein Spielfeld dort einen Spielstein wegnehmen beziehungsweise hinsetzen kann. Nun wollen wir im nächsten Schritt eine rudimentäre Spiellogik einbauen: zwei SpielerInnen sollen abwechselnd ziehen können. Zug bedeutet dabei: man wählt einen eigenen Stein aus und setzt ihn auf ein freies Feld. Die Spielregeln von Dame werden dabei noch nicht berücksichtigt. Trotzdem suchen wir auch jetzt schon eine systematische und übersichtliche Lösung, die später leicht zu erweitern ist. Betrachten wir dazu einen Zeitpunkt während des Spiels. Dann stellen sich die Fragen

- in welcher Spielsituation befinden wir uns?
- welche Aktionen sind möglich?
- wohin (zu welchen anderen Spielsituationen) führen die Aktionen?

Beginnen wir mit der Anfangssituation:

- SpielerIn 1 ist am Zug
- SpielerIn 1 kann einen eigenen Stein zum ziehen auswählen
- dies führt zu *Zielfeld auswählen*

Übersichtlich kann man dies als Tabelle 11.1 schreiben.

Tabelle 11.1: Anfangszustand

Nr	Zustand	Klick	Folge
10	S1 am Zug	eigenen Stein	11
		anders	10

Tabelle 11.2: Zustände und Übergänge im Dame-Spiel

Nr	Zustand	Klick	Folge
10	S1 am Zug	eigenen Stein	11
		anders	10
11	Zug ausführen	freies Feld	20
		anders	11
20	S2 am Zug	eigenen Stein	21
		anders	20
21	Zug ausführen	freies Feld	10
		anders	21

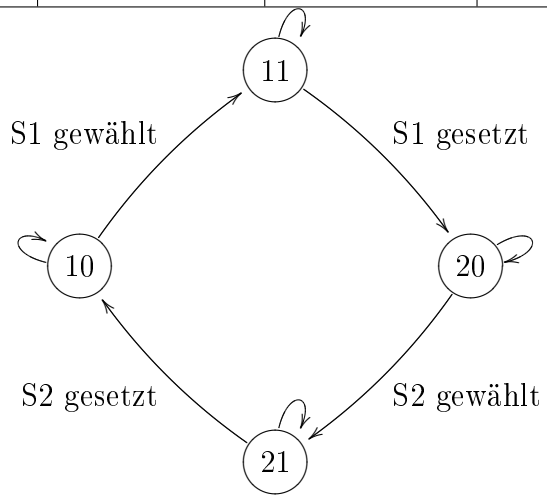


Abbildung 11.1: Zustandsdiagramm für Dame-Spiel

Wir verwenden dabei die allgemeinere Bezeichnung Zustand für eine Spielsituation und haben eine Nummerierung begonnen. Die Nummerierung der Zustände ist frei wählbar. Die 10 wurde gewählt um SpielerIn 1 (S1) und Anfang (0) zu symbolisieren. Die Tabelle besagt, dass man in Zustand 10 durch Anklicken eines eigenen Steines in einen noch zu beschreibenden Folgezustand 11 gelangt. Alle anderen Klicks werden ignoriert und man verbleibt im Zustand 10.

Im folgenden Zustand 11 soll ein freies Zielfeld ausgewählt werden. Damit ist der Zug beendet und SpielerIn 2 am Zug. Dafür sehen wir einen weiteren Zustand 20 vor. Mit noch einem Zustand, in dem SpielerIn 2 den Zug beendet, erhalten wir die Tabelle 11.2. Diese Zustandsübergangstabelle beschreibt alle möglichen Zustände und Übergänge zwischen den Zuständen. Alternativ kann man diese Information auch grafisch darstellen. Bild 11.1 zeigt eine solche Darstellung (Zustandsdiagramm). Die Zustände werden durch Kreise symbolisiert und Pfeile markieren die möglichen Übergänge.

Frage 11.1. Nach dieser Tabelle muss man einen ausgewählten Stein auch ziehen.

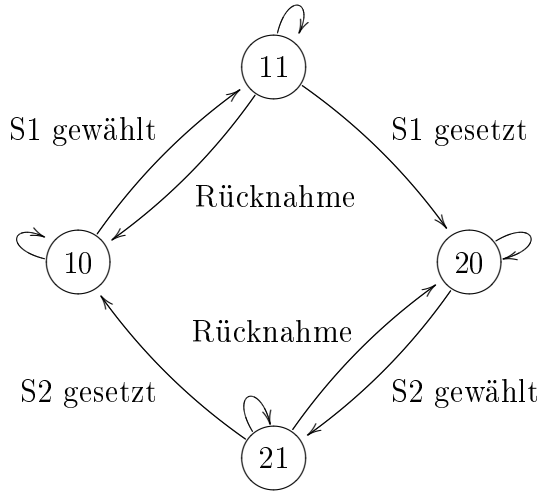


Abbildung 11.2: Zustandsdiagramm mit Rücknahme des berührten Steins

Wie könnte man dies ändern?

Unser Modell hält sich an die *berührt – geführt* Regel aus dem Turnierschach. Wir könnten es aber leicht anpassen, indem wir im Zustand 11 (und entsprechend 21) einen weiteren Fall *ausgewählte Figur nochmals anklicken* unterscheiden. Dies würde direkt wieder zurück zu Zustand 10 führen. Solche Fragen lassen sich leicht anhand der Übergangstabelle oder des Diagramms klären. Bild 11.2 zeigt ein entsprechend erweitertes Zustandsdiagramm.

Frage 11.2. Und wie kommen wir jetzt zum C-Code?

Die Umsetzung von Zustandsdiagramm oder Übergangstabelle in Programmcode ist recht einfach. Zumindest für den Aufbau der Struktur muss man dazu nicht einmal das Problem verstanden haben. Daher lässt sich die Umsetzung gut automatisieren und es gibt diverse Programme, um aus einem Zustandsmodell Code zu generieren. Ausgangspunkt ist eine Variable für den aktuellen Zustand. Setzen wir für unseren Fall mit

```
int zustand = 10;
```

den Startzustand. Dann warten wir in einer Schleife auf Aktionen. Sobald eine Aktion auftritt, gehen wir zum aktuellen Zustand, werten die Aktion aus und legen den Folgezustand fest. Für die Auswahl des Zustandes bietet sich eine Konstruktion mit if und else-if oder ein switch-Element an. Beides funktioniert gut, wobei das switch-Element etwas übersichtlicher ist. Für jeden Zustand steht ein case-Block. Wir erhalten dann folgende allgemeine Form

```
for(;;) {
    // auf Aktion warten
    switch( zustand ) {
```

```
    case 10:
        Aktion auswerten
        Folgezustand setzen
        break;
    case 11:
        Aktion auswerten
        Folgezustand setzen
        break;
        ...
    }
}
```

Auf diese Art und Weise erhalten wir ein Grundgerüst, in das wir nur noch die Auswertung der Aktionen integrieren müssen. den vollständigen Code zeigt Listing 11.1. Die Details:

- die Anwendung fragt mit `abfragen()` nach neuen Nachrichten. Falls keine vorliegen (`strlen( a )` liefert 0), warte sie für 100 ms.
- neue Nachrichten werden mit `printf` ausgegeben und es wird geprüft, ob es sich um ein Klick handelt (die Nachricht mit `#` beginnt).
- dann werden die Koordinaten aus der Nachricht gelesen und in `ix` und `iy` geschrieben
- nach der Auswahl eines Steins werden dessen Koordinaten in `xalt` und `yalt` gepeichert
- ein ausgewählter Stein wird vergrößert (Wert in `radG`)
- die Ausführung eines Zugs ist zur besseren Übersicht in eine eigene Funktion ausgelagert (11.2)
- zur Sicherheit wurde eine `default`-Anweisung eingebaut, um bei unbekannten Zuständen eine Meldung auszugeben (sollte nicht vorkommen)
- mit der Methode statusText() werden Informationen zum Spielverlauf in der Statuszeile von BoS (oberhalb des Spielbretts) angezeigt

Frage 11.3. Was bedeutet `%*d` in `sscanf_s( a, "# %*d %d %d", &ix, &iy );`?

Mit dem Stern markieren wir, dass ein Wert zwar gelesen werden soll, aber nicht in einer Variablen abgespeichert wird. In diesem Fall wird die Nummer des angewählten Spielfeldes übersprungen und nur die Werte für Spalte und Zeile werden eingetragen.

Listing 11.1: Umsetzung der Übergangstabelle

```
int zustand = 10; // Startzustand
int ix, iy; int xalt, yalt;
statusText( "SpielerIn 1 am Zug" );
for(;;) {
    char *a = abfragen();
    if( strlen( a ) > 0 ) {
        printf( "Nachricht: %s\n", a );
        if( a[0] == '#' ) {
            sscanf_s( a, "# %*d %d %d", &ix, &iy );
            switch(zustand ) {
            case 10:
                if( brett[ix][iy] == 1 ) {
                    symbolGroesse2( ix, iy, radG );
                    xalt = ix; yalt = iy;
                    zustand = 11; }
                break;
            case 11:
                if( ix == xalt && iy == yalt ) {
                    symbolGroesse2( ix, iy, rad );
                    zustand = 10;
                } else if( brett[ix][iy] == 0 ) {
                    zug( xalt, yalt, ix, iy, 1 );
                    statusText( "SpielerIn 2 am Zug" );
                    zustand = 20; }
                break;
            case 20:
                if( brett[ix][iy] == 2 ) {
                    symbolGroesse2( ix, iy, radG );
                    xalt = ix; yalt = iy;
                    zustand = 21; }
                break;
            case 21:
                if( ix == xalt && iy == yalt ) {
                    symbolGroesse2( ix, iy, rad );
                    zustand = 20;
                } else if( brett[ix][iy] == 0 ) {
                    zug( xalt, yalt, ix, iy, 2 );
                    statusText( "SpielerIn 1 am Zug" );
                    zustand = 10; }
                break;
            default:
                printf("Ungueltiger Zustand %d\n", zustand );
            }
        }
    } else { Sleep( 100 ); }
}
```

Listing 11.2: Funktion zug

```
void zug( int xalt, int yalt, int ix, int iy, int spieler ) {
       brett[ix][iy] = brett[xalt][yalt];
       brett[xalt][yalt] = 0;
       form2( xalt, yalt, "none" );
       symbolGroesse2( xalt, yalt, rad );
       farbe2( ix, iy, farben[spieler] );
       form2( ix, iy, "c" );
       symbolGroesse2( ix, iy, rad );
}
```

Kapitel 12

Zeiger

Im bisherigen Teil haben wir gesehen, wie mit dem Konzept von Variablen Werte im Speicher abgelegt und bearbeitet werden können. Eine Variable besteht dabei aus:

- Name
- Datentyp
- Wert

Durch den Datentyp ist auch die Größe des benötigten Speichers bestimmt. Eine Erweiterung stellen Felder dar, bei denen eine Anzahl von Werten – sei es als lineares Feld oder mit einer mehrfachen Indizierung als mehrdimensionales Feld — zusammen gehören. In jedem Fall erfolgt der Zugriff auf eine Variable über den Namen — gegebenenfalls mit zusätzlichen Indizes. Die Adresse im Speicher spielt keine Rolle. Es ist gerade eine der Hauptaufgabe des Compilers, den Speicher zu organisieren und für die Variablen ausreichend Platz zu reservieren. Es gibt jedoch auch Fälle, in denen direkter Zugriff auf eine Speicherzelle notwendig ist. Wichtige Beispiele sind:

- Ändern von Variablen im Hauptprogramm aus Funktionen
- Anlegen von neuen Objekten zu Laufzeit

In C gibt es für derartige Anwendungen Zeiger (*pointer*). Eine Zeigervariable enthält die Adresse eines Objektes im Speicher. Um das Objekt benutzen zu können, muss auch der Datentyp bekannt sein, d. h. eine Zeigervariable muss auch mit einen Datentyp verknüpft sein. Eine Zeigervariable wird durch die Angabe des Datentyps, des Namens und einen zusätzlichen Stern vor dem Namen definiert.

```
int *p; /* p ist ein Zeiger auf ein int Objekt */
double *pd /* pd ist ein Zeiger auf ein double Objekt */
```

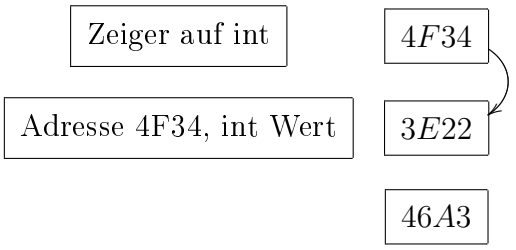


Abbildung 12.1: Zeiger auf int-Wert

In einer Anweisung können mehrere Zeiger vom gleichen Typ definiert werden. Dabei muss der Stern vor jedem Namen wiederholt werden.

```
long *pl1, *pl2;
/* pl1 und pl2 sind Zeiger auf long Objekte */
```

Die Definitionen von Variablen und Zeigervariablen können sogar gemischt werden.

```
float wert, *zwert;
/* wert ist ein float Objekt und
   zwert ist ein Zeiger auf ein float Objekt */
```

Eine Zeigervariable enthält die Adresse auf ein Objekt des entsprechenden Typs. Abbildung 12.1 zeigt ein Beispiel für die Verbindung zwischen Zeigerobjekt und Objekt. Der absolute Wert der Zeigervariable selbst ist nicht wichtig. Nützlich wird eine Zeigervariable durch zwei Möglichkeiten:

- über den Zeiger kann indirekt der Wert des Objektes angesprochen werden
- der Zeiger kann auf ein anderes Objekt des gleichen Typs verändert werden

Den Inhalt des Objektes, auf das der Zeiger deutet, erhält man über den Dereferenzierungsoperator `*`. Wie bei der Definition wird der Stern vor den Namen gesetzt. Nach der Definition `int *p;` kann mit `*p` auf den Inhalt des Objektes zugegriffen werden. Man kann beispielsweise schreiben

```
*p = ( 5 * *p ) / 3;
```

Der Ablauf ist:

- Hole den Inhalt der Zelle, auf die `p` zeigt
- Berechne damit den Ausdruck auf der rechten Seite
- Schreibe das Resultat in die Zelle, auf die `p` zeigt

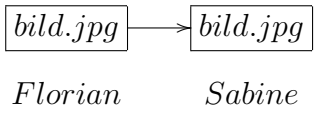


Abbildung 12.2: Kopieren einer Bilddatei

Die Kombination `*Name` — Dereferenzierungsoperator und Zeiger -– entspricht syntaktisch einer Variablen. Damit ist es möglich, indirekt auf ein Objekt zuzugreifen, ohne dass man den zugehörigen Variablennamen kennen muss. Insbesondere kann man damit auch Variablen ansprechen, deren Namen im aktuellen Modul überhaupt nicht bekannt sind. Mit dieser Technik können Funktionen Werte im Hauptprogramm ändern. Definieren wir eine Funktion, die als Argument einen Zeiger hat:

```
void rechne( int *p ) {
    *p = ( 5 * *p ) / 3;
}
```

Dann wird in der Funktion der Wert in der Zelle, auf die der Zeiger deutet, verändert. Die Funktion erhält nicht eine Kopie eines Wertes sondern einen Verweis auf seine Position. Daher bezeichnet man diesen Mechanismus als *call by reference*.

Damit gibt es neben der uns schon bekannten Übergabe des Wertes (*call by value*) eine zweite Möglichkeit, Informationen an eine Funktion zu übermitteln. Man kann sich die prinzipiellen Unterschiede gut an einem praktischen Beispiel klar machen. Angenommen Florian hat eine Bilddatei `bild.jpg` und möchte das Bild an Sabine weitergeben. Eine Möglichkeit ist, ihr die Datei zu kopieren. Wie in Bild 12.2 dargestellt, erhält Sabine eine Kopie der Datei. Dies entspricht der Übergabe mittels call by value. Das Verfahren ist aufwändig bezüglich Übertragung und Speicherbedarf. Das komplette Bild wird übertragen und anschließend wird doppelter Speicherplatz belegt. Die beiden Versionen existieren von nun an getrennt. Ändert zum Beispiel Sabine etwas an dem Bild, so hat dies keinerlei Auswirkungen auf Florians Bild.

Die Übergabe als call by reference kann man sich wie folgt vorstellen: Florians Bild liegt irgendwo im Internet. Anstelle der Datei schickt er Sabine nur den Verweis (Link) auf diese Stelle. Bild 12.3 zeigt den Ablauf. In diesem Fall wird nur ein wenig Text für den Verweis übertragen. Florian und Sabine verwenden anschließend das gleiche Bild. Ändert Sabine etwas an dem Bild, so sieht Florian ebenfalls die Auswirkung.

Beide Übergabeverfahren haben Vor- und Nachteile. Je nach Anwendung kann das eine oder das andere besser geeignet sein. Wichtig ist, dass man sich die Unterschiede klar macht und dann die Konsequenzen in der weiteren Verarbeitung berücksichtigt.

Die beim Aufruf übergebene Zeigervariable selbst ist wiederum nur eine Kopie der Zeigervariablen im aufrufenden Programm. Eventuelle Änderungen an der

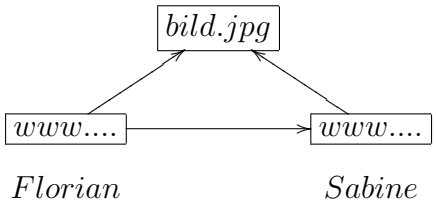


Abbildung 12.3: Übergabe eines Verweises

Zeigervariablen sind nur in der Funktion wirksam. In dem Hauptprogramm wird noch eine Möglichkeit benötigt, die Adresse eines Objektes anzugeben. Dazu stellt C den Adressoperator `&` zur Verfügung. Für eine Variable `Name` ergibt dann `&Name` deren Adresse. Im folgenden Beispiel wird eine Variable `i` sowie ein darauf verweisender Zeiger definiert. Anschließend wird einmal über den Variablennamen und dann über den Zeiger der Inhalt geändert:

```
int i; /* eine int Variable */
int *p = &i; /* ein Zeiger auf die Variable i */

i = 10; /* i hat jetzt den Wert 10 */
*p = 12; /* i hat jetzt den Wert 12 */
```

Betrachten wir ein ausführlicheres Beispiel:

```
/* Funktion zum Tauschen zweier Variablen */
void tausche( int *a, int *b ) {
    int tmp;

    tmp = *b;
    *b = *a;
    *a = tmp;
}

void main( void )
{
    int i = 1, j = 2;
    int *p1 = &i; /* p1 zeigt auf i */
    int *p2 = &j; /* p2 zeigt auf j */

    printf( "i = %d, j = %d \n", i, j );
    tausche( p1, p2 );
    printf( "i = %d, j = %d \n", i, j );
}
```

In dem Hauptprogramm werden zwei Variablen definiert. Anschließend werden zwei Zeiger auf diese Variablen gesetzt. Die beiden Zeiger werden dann als Argumente an eine Funktion übergeben. In der Funktion werden über die Zeiger die Werte der beiden Variablen getauscht. Es ist nicht notwendig, explizite Zeigervariablen zu benutzen. Man kann auch schreiben:

```
tausche( &i, &j );
```

Diese Konstruktion hatten wir bereits bei der Funktion `scanf` benutzt. `scanf` ist ein typisches Beispiel für eine Funktion, die den Inhalt von Variablen verändert. Allerdings müssen die Argumente auch tatsächlich Zeiger sein und es liegt in der Verantwortung des Programmierers (bzw. des Compilers), sicher zu stellen, dass auch in der Tat Zeiger übergeben werden. Es ist einer der häufigsten Fehler in C Programmen, an `scanf` die Variablen selbst statt deren Adresse zu übergeben. Das Programm interpretiert dann den Inhalt der Variable als Speicheradresse und schreibt den gelesenen Wert an diese falsche Stelle.

12.1 Zeiger und Felder

Felder sind zusammenhängende Speicherbereiche, die mehrere Objekte des gleichen Typs hintereinander enthalten. Betrachten wir ein Feld mit 5 Elementen:

```
int feld[5];
```

Das Feld enthält die einzelnen Elemente `feld[0]`, `feld[1]`, ..., `feld[4]`. Ein Zeiger kann auf ein einzelnes Element gesetzt werden:

```
int *p;
p = &feld[0]; /* p deutet auf das erste Element in feld */
```

Innerhalb des Feldes kann der Zeiger durch Addition oder Subtraktion verschoben werden.

```
p = p + 1; /* p deutet auf feld[1] */
++p; /* p deutet auf feld[2] */
--p; /* p deutet auf feld[1] */
p += 2; /* p deutet auf feld[3] */
```

Die Berechnung einer neuen Adresse funktioniert nur richtig, wenn die Größe eines Objektes im Speicher berücksichtigt wird. Die Operation `++p` hat als Beispiel einen unterschiedlichen Effekt für short und long Variablen. Der Compiler berücksichtigt die Größe der Objekte und berechnet die neue Adresse damit richtig. Allerdings wird die Einhaltung der Feldgrenzen nicht geprüft. Felder und Zeiger sind in C eng verwandt. Der Name des Feldes – in unserem Beispiel `feld` – ist äquivalent zu einem Zeiger, d. h. `feld` ist ein Zeiger auf den Beginn des Feldes. Damit kann man auch schreiben

```
p = feld; /* p deutet auf das erste Element in feld */
```

Das zweite Element ist durch

```
feld[1]
```

oder

```
*(p+1)
```

gegeben. Da aber C Feldernamen und Zeiger gleich behandelt, sind auch folgende Ausdrücke legal:

```
*(feld+1)
p[1]
```

Intern wandelt C einen Ausdruck in der Form `a[i]` in `*(a+i)` um. Indizierung eines Feldes oder Addition eines Offsets zu einem Zeiger sind gleichwertig. Da die Addition eine kommutative Operation ist, ist sogar die folgende (nicht empfehlenswerte) Schreibweise syntaktisch korrekt:

```
1[feld]
```

Auch als Argumente für Funktionsaufrufe sind Feldernamen und Zeiger austauschbar wie folgender Code zeigt

```
void mystrcpy( char *a, char *b ) {
...
}

void main( void )
{
    char z1[] = "Test Zeichenkette";
    char z2[40];

    mystrcpy( z2, z1 );
}
```

Die Funktion `mystrcpy` ist zwar mit Zeigern als Argumenten definiert, kann aber auch mit Feldnamen aufgerufen werden. Man kann daher den Feldnamen als Zeiger auf den zugehörigen Speicher verstehen. Er ist aber konstant und kann nicht im Programm auf einen anderen Bereich geändert werden.

12.2 Zeigerarithmetik

In den Beispielen wurden bereits Zeiger verändert, im wesentlichen um die einzelnen Elemente in Feldern zu adressieren. Insbesondere gilt für die Adressierung, dass automatisch die Adressberechnung an die Größe der Objekte angepasst wird. Für einen Zeiger `p` von einen bestimmten Typ und eine Integerzahl `n` ergibt `p + n` genau die Adresse des n–ten Objektes nach dem Objekt, auf das `p` zeigt. Wichtig

ist der Unterschied zwischen der Zeigervariablen selbst und dem Wert auf den sie deutet. Die beiden Anweisungen

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

bewirken vollkommen unterschiedliche Effekte. Im ersten Fall wird der Zeiger erhöht, d. h. `p` deutet anschließend auf das nächste Objekt im Speicher. Im zweiten Fall wird der Inhalt des Objektes um eins erhöht, der Zeiger bleibt auf dem Objekt stehen. Wie bereits erwähnt, kann der Ausdruck `*p` überall dort stehen, wo auch der Name der Variablen, auf die `p` deutet, stehen könnte. Demgegenüber sind mit Zeigern nur folgende Operationen möglich:

- Zuweisung eines Zeigers zu einem anderen Zeiger des gleichen Typs. Beide Zeiger deuten anschließend auf die gleiche Speicherposition.
- Addition oder Subtraktion einer Integer Größe. Dies beinhaltet auch die Inkrement bzw. Dekrement Operatoren `++` und `--`.
- Subtraktion eines anderen Zeigers des gleichen Typs. Wenn beide Zeiger auf Elemente eines Feldes deuten, ergibt die Subtraktion die Anzahl der Elemente dazwischen.
- Vergleich zweier Zeiger des gleichen Typs. Die Vergleichsoperationen $==$, $<$, $>$, etc. können auf Zeiger angewandt werden. Sinnvolle Resultate ergeben sich allerdings nur, wenn beide Zeiger in das gleiche Feld zeigen (oder an die Position unmittelbar nach dem letzten Element).
- Zuweisung von 0 zu einem Zeiger
- Vergleich eines Zeigers mit 0
- Umwandlung zwischen Integerwerten und Zeigern ist möglich, aber implementierungsabhängig.
- Der Wert eines Zeigers kann in printf mit `%p` ausgegeben werden,

Die `0` (NULL) wird häufig benutzt um anzuzeigen, dass ein Zeiger nicht initialisiert ist oder eine Funktion keine sinnvolles Resultat liefern kann. Andere Verknüpfungen sind mit Zeigervariablen nicht erlaubt. Außerdem können Zeiger unterschiedlichen Typs nicht ohne explizites cast einander zugewiesen werden. (Fehlermeldung: *Die Typen, auf die verwiesen wird, sind nicht verwandt; die Konvertierung erfordert einen reinterpret cast-Operator oder eine Typumwandlung im C- oder Funktionsformat*). Die genauen Regeln für die Umwandlung sind Implementierungsabhängig. Sicher ist die Umwandlung von und nach Zeigern vom Typ `void *`.

12.3 Zeiger und Zeichenketten

Zeiger vereinfachen den Umgang mit konstanten Zeichenketten. C erlaubt die direkte Angabe von Zeichenkonstanten mit einem Zeiger in der Form

```
char *name = "Text";
```

Damit wird entsprechend viel Speicher bereit gestellt, um den Text – inklusive der abschließenden '\0' – aufzunehmen. Im Gegensatz zum Initialisieren eines entsprechend großen Feldes wird die Zeichenkette aber als konstant, d. h. konstante Länge und unveränderbarer Inhalt, angesehen. Es bleibt dem Compiler überlassen, eine spezielle Verwaltung für solche konstanten Zeichenketten einzuführen.

Beispiel 12.1. Initialisierung von Zeichenketten

```
char f0[] = "Friedberg"; /* ein Feld mit 10 Zeichen */
char *f1 = "Friedberg"; /* ein Zeiger auf eine Zeichenkette */

printf( "%s %c %s %c \n", f0, f0[0], f1, f1[0] );
f0[0] = 'f';
/* Änderung in Feld, okay */

printf( "%s %c %s %c \n", f0, f0[0], f1, f1[0] );
f1[0] = 'f';
/* Änderung in konstantem Text, Access Violation */
```

In beiden Fällen wird Speicher für den Text bereit gestellt. Im Fall des Feldes ist dies normaler Speicher, dessen Inhalt auch im Programm verändert werden kann. Im zweiten Fall liegt der Text in einem speziellen, geschützten Speicherbereich. Der Versuch, den Inhalt zu ändern führt zu einem Programmabbruch. Als weitere Besonderheit kann der konstante Text mehrfach verwendet werden. Ergänzt man den obigen Programmteil um die Zeilen

```
    char *f2 = "Friedberg";
...
printf( "f0 = %p, f1 = %p, f2 = %p \n", f0, f1, f2 );
```

erhält man das Resultat

```
f0 = 0012FEF4, f1 = 00421028, f2 = 00421028
```

f1 und f2 deuten auf die gleiche Adresse, die aber nicht mit der in f0 übereinstimmt. Die Zeiger selbst sind im Allgemeinen nicht konstant und können durchaus auf andere Texte oder Felder verändert werden. Im Beispiel

```
char *f3 = "Wetterau";
    ...
f1 = f3; /* f1 deutet jetzt auch auf den Text Wetterau */
```

wird kein Text kopiert, lediglich der Zeiger wird „umgehängt“. In vielen Fällen wird die Umwandlung von Text in Zeigern implizit vorgenommen. Als Beispiel erwartet die Funktion `printf` als erstes Argument einen Zeiger.

```
int printf( const char *format [, argument]... );
```

Der direkte eingetragene Text wird automatisch in einen solchen Zeiger umgesetzt. Die Angabe `const char *` bedeutet, dass der Zeiger auf einen konstanten Text deutet.

12.3.1 Felder von Zeichen

Die enge Verwandtschaft zwischen Zeigern und Feldern einerseits und die Realisierung von Zeichenketten als Feldern von Zeichen ermöglicht die elegante Bearbeitung von Zeichenketten mittels Zeigern. In einem der Beispiele wurde die Funktion zum Kopieren einer Zeichenkette in eine zweite eingeführt:

```
void mystrcpy( char *a, char *b ) {
...
}
```

Mit Feldern erhält man beispielsweise folgende Implementierung:

```
void mystrcpy( char *a, char *b ) {
    int i = 0;
    while( ( a[i]=b[i] ) != '\0' ) i++;
}
```

In dieser Version wird eine Indexvariable hochgezählt, bis die abschließende Null erreicht ist. Durch die Verwendung der while-Schleife ist sichergestellt, dass auch diese Null noch kopiert wird. Mit Zeigern kann man die gleiche Funktionalität schreiben als

```
void mystrcpy( char *a, char *b ) {
    while( ( *a++ = *b++ ) != '\0' ) ;
}
```

Anstelle der Indexvariablen werden die Zeiger selbst hoch gezählt. Die Zeiger können problemlos verändert werden, da es sich lediglich um die lokalen Kopien handelt. Diese Realisierung betont stärker den Charakter von Zeichenketten (Strings): ein Zeiger auf den Anfang und ein `'\0'` als Endmarkierung. Zeichenketten sind zwar in Feldern eingebettet, aber häufig liegen die Grenzen der Zeichenkette innerhalb der Feldgrenzen. Die Version mit Zeigerarithmetik ist eine typische Konstruktion für C. Mit der Vereinfachung, dass die Abfrage nach „ungleich 0“ Standard ist, erhält man eine sehr knappe, elegante Formulierung:

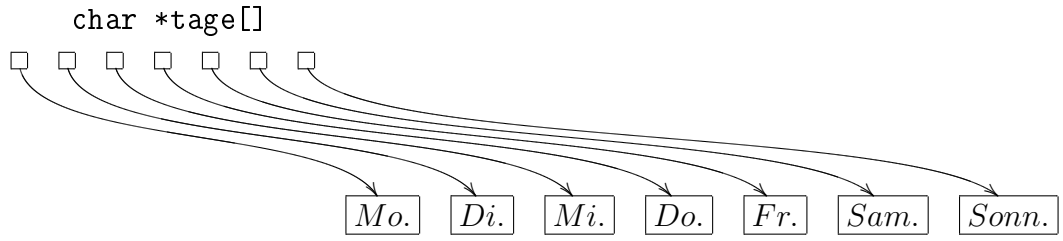


Abbildung 12.4: Zeiger auf Zeichenketten

```
void mystrcpy( char *a, char *b ) {
    while( *a++ = *b++ ) ;
}
```

12.3.2 Felder von Zeigern auf Zeichenketten

Mehrere Zeiger vom gleichen Typ können wie elementare Variablen zu Feldern zusammengefasst werden. Die Dimension wird wie gehabt bei der Definition in eckigen Klammern explizit angeben oder durch die Initialisierungswerte implizit vorgegeben.

Beispiel 12.2. Felder von Zeigern

```
int *a[5]; /* 5 Zeiger auf int */
float *f[2][3]; /* 2 x 3 Feld von Zeigern auf float */
char *tage[] =
{ "Mo", "Di", "Mi", "Do", "Fr", "Sam.", "Sonn." };
/* 7 Zeiger auf konstante Zeichenketten */
```

Das letzte Beispiel zeigt, wie man elegant mehrere zusammen gehörende, konstante Zeichenketten definieren kann. Das Feld `tage` wird automatisch groß genug angelegt, um die Zeiger aufzunehmen. Die Zeichenketten selbst werden vom Compiler in passende Speicherbereiche gelegt. Nimmt man der Einfachheit halber an, dass die Zeichenketten direkt hintereinander liegen, ergibt sich Bild 12.4 Mit `tage[i]` kann man auf die einzelnen Wochentage zugreifen. Eine Liste der Tage wird mit folgendem Code ausgegeben:

```
for( k=0; k< sizeof tage / sizeof( char * ); k++ ) {
    printf( "%d-ter Tag ist %s\n", k, tage[k] );
}
```

Die Zeichenketten — die Namen der Wochentage -– selbst sind konstant und können im Allgemeinen nicht verändert werden. Die Zeiger im Feld `tage` andererseits können auf andere Adressen geändert werden. In dem folgenden Code-Abschnitt werden nach diesem Prinzip die Zuordnungen zu den Zeichenketten verändert:

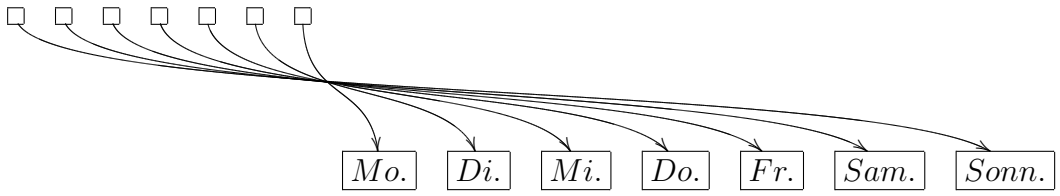


Abbildung 12.5: Zeiger auf Zeichenketten nach Umkehrung

```
char *tmp;
/* Wochentage umsortieren */
for( k=0; k<3; k++ ) {
    tmp = tage[k];
    tage[k] = tage[6-k];
    tage[6-k] = tmp;
}
```

Mit dieser Schleife wird die Reihenfolge der Tage umgekehrt. Die Variable `tmp` dient als Zwischenspeicher. Bild 12.5 zeigt die geänderte Zuordnung. Genauso kann ein Zeiger auf eine vollkommen andere Zeichenkette gesetzt werden:

```
tage[0] = "Monday";
```

Man kann beispielsweise die Farbnamen in einem zweidimensionalen Feld speichern:

```
char farbNamen[][FARB_GROESSE] =
{ "rot", "gruen", "blau", "schwarz", "gelb", "weiss",
    "braun", "lila", "orange", "violett", "tuerkis"};
```

Mit Zeigern würde die entsprechende Konstruktion

```
char *farbNamen[] =
{ "rot", "gruen", "blau", "schwarz", "gelb", "weiss",
    "braun", "lila", "orange", "violett", "tuerkis"};
```

lauten. In beiden Fällen kann `farbNamen` gleich benutzt werden. Die Anweisung

```
printf( "%8s ", farbNamen[zellen[i]]);
```

gilt für beide Konstruktionen. Der Unterschied liegt in der internen Repräsentation. Das zweidimensionale Feld stellt für jede Farbe gleich viele Zeichen zur Verfügung. Es wird insgesamt ein Speicherbereich der Größe (Anzahl der Farben) * `FARB_GROESSE` reserviert. Das Bild im Speicher ist

rot\0****	gruen\0**	blau\0***

bei `FARB_GROESSE` gleich 8. Es ist garantiert, dass ein zusammenhängender Speicherbereich reserviert wird. Beim Anlegen muss die Größe der Zeilen festgelegt werden. Dabei ist die längste Zeichenkette maßgebend. Die Größe muss so gewählt

werden, dass diese Zeichenkette – einschließlich der NULL – hinein passt. Bei den anderen – kürzeren – Zeichenketten bleiben Speicherzelle uninitialisiert. Im Programm können beliebige Zellen in dem zweidimensionalen Feld neu beschrieben werden, aber die Struktur – d. h. Größe von Zeilen und Spalten – bleibt erhalten. Im Vergleich wird deutlich, dass die Struktur Feld von Zeigern flexibler und effizienter ist. Der Einsatz ist dabei nicht auf konstante Zeichenketten beschränkt. Im übernächsten Kapitel werden wir sehen, wie man mit Methoden der dynamischen Speicherverwaltung auch Zeiger auf veränderbare Bereiche behandeln kann.

12.4 Zeiger auf Zeiger △

Zeigervariablen sind selbst auch Variablen. Es ist daher durchaus möglich, Zeiger auf Zeigervariablen zu definieren. Beispielsweise ist

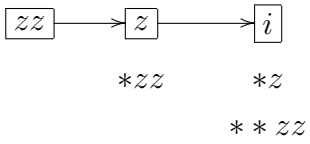
```
int **zz;
```

eine legale Definition eines Zeigers auf einen Integerzeiger. Bei richtiger Initialisierung deutet `zz` auf eine Adresse, in der wiederum die Adresse eines int Wertes steht. Durch doppelte Dereferenzierung mit `**` kann man auf den `int` Wert zugreifen.

Beispiel 12.3. Zeiger auf Zeiger
Mit

```
int i=1, *z, **zz;


z = &i; /* Zeiger auf i */
zz = &z; /* Zeiger auf Zeiger auf i */
printf( "Inhalt von **zz (Wert von i): %d\n", **zz);
```

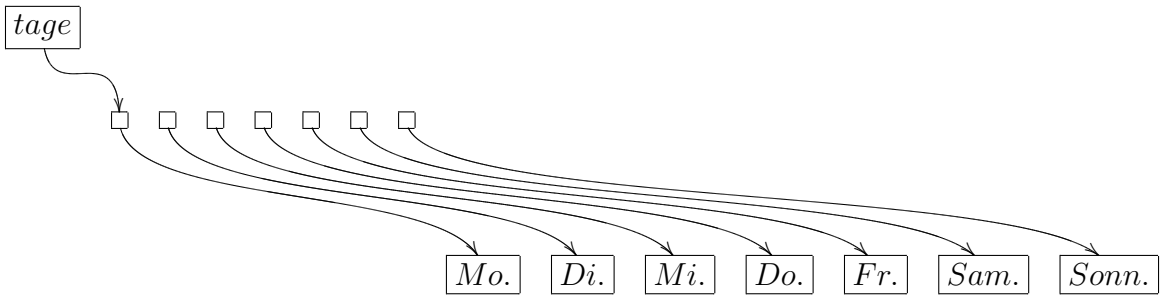
wird eine Verkettung über zwei Zeiger angelegt. Die folgende Darstellung zeigt die entsprechende Beziehungen:



In der Praxis spielen Zeigern auf Zeigern vor allem bei Feldern mit Zeigern auf Zeichenketten eine Rolle. In der Definition

```
char *tage[]
```

ist `tage` der Name eines Feldes. Wir hatten gesehen, dass in C Feldnamen auch als Zeiger interpretiert werden können. In diesem Sinne entspricht `tage` einem Zeiger vom Typ `char **` und `tage` deutet auf den ersten Zeiger im Feld, der wiederum auf die erste Zeichenkette deutet. Im Speicher erhält man folgendes Bild:



Der Wert von `tage` ist fest. Man kann aber den Wert in eine Variable kopieren und dann mit dieser Adresse rechnen. Beispielsweise kann `tage` als Argument an eine Funktion übergeben werden. Dann wird automatisch bei der Argumentübergabe eine Kopie erstellt, die in der Funktion verändert werden kann. Die Funktion

```
void textOut( char **p, int n ) {
    int i;
    for( i=0; i<n; i++ ) printf("%s\n", *p++);
}
```

kann mit

```
    textOut( tage, 7 );
```

aufgerufen werden. In der Funktion wird der Zeiger auf das Zeigerfeld hochgezählt. Eine einfache Dereferenzierung `*p` liefert dann den Zeiger auf die jeweilige Zeichenkette. Ein zweidimensionales Feld wird nicht als Argument für die Funktion `textOut` akzeptiert. Das zweidimensionale Feld besteht lediglich aus dem entsprechend großen Speicherbereich, es gibt aber keine Zwischenstruktur mit Zeigern auf einzelne Zeilen. Trotzdem wird auch das zweidimensionale Feld als Zeiger interpretiert. Allerdings deutet der Feldzeiger nicht auf einen weiteren Zeiger sondern auf eine ganze Zeile. Die Definition

```
char farbNamen[][FARB_GROESSE] = ...
```

kann als Zeiger auf `char` Felder der Länge `FARB_GROESSE` verstanden werden. `farbNamen` deutet auf das erste Feld (die erste Zeile). Entsprechend ist `farbNamen + 1` ein Zeiger auf die zweite Zeile. Die Addition von 1 bedeutet, dass der Zeiger auf das nächste Objekt weiter rückt. Mit der obigen Definition ist das Objekt ein char Feld der Länge `FARB_GROESSE` und damit wird der Zeiger um

```
FARB_GROESSE * sizeof( char )
```

Stellen erhöht. Daher kann man mit der Schleife

```
for( k=0; k<10; k++ ) printf("%s\n", farbNamen+k );
```

die Farben ausgeben. Eine äquivalente Schreibweise ist:

```
for( k=0; k<10; k++ ) printf("%s\n", farbNamen[k] );
```

Weiterhin ist es möglich, explizit Zeiger auf Felder zu definieren. Für das Beispiel der Farbnamen ist die Syntax:

```
char (*zf)[FARB_GROESSE] = farbNamen;
/* Zeiger auf char Felder der Länge FARB_GROESSE */
```

Danach kann man mit `zf` als Zeiger rechnen:

```
for( l=0; l<10; l++ ) printf("## %s\n", zf++);
```

12.5 Zeiger auf Funktionen △

Neben Zeigern auf Daten sind in C auch Zeiger auf Funktionen möglich. Der Zeiger enthält dann die Position im Speicher, an der der Code für die Funktion beginnt. Bei der Definition eines Zeigers auf eine Funktion werden die Argumentliste und der Rückgabewert angegeben. Um beispielsweise einen Zeiger `zfkt` auf eine Funktion mit einem `double` Argument und Rückgabewert `double` zu definieren, schreibt man:

```
double ( *zfkt)( double );
```

Die Klammer um den Namen ist notwendig, um die Definition von einer Funktionsdeklaration zu unterscheiden. Mit einer Anweisung wird dann der Zeiger auf eine konkrete Funktion gesetzt:

```
printf( "Sinus " );
zfkt = &sin; /* Zeiger auf Sinus Funktion */
printf( "Funktion ergibt :%g\n", zfkt( 0. ) );
printf( "Cosinus " );
zfkt = &cos; /* Zeiger auf Cosinus Funktion */
printf( "Funktion ergibt :%g\n", zfkt( 0. ) );
```

Funktion und Zeiger müssen in Argumentliste und Rückgabetyp übereinstimmen. Beim Aufruf kann auf die Dereferenzierung verzichtet werden. Die vollständige Form des Aufrufs ist:

```
( *zfkt)( 0. )
```

Zeiger auf Funktionen sind besonders als Argumente bei Funktionsaufrufen sinnvoll. Die Bibliotheksfunktion `qsort` für den Quicksort beispielsweise erwartet als viertes Argument den Zeiger auf eine Vergleichsfunktion. Dadurch kann man die gleiche Funktion für verschiedene Datentypen benutzen. Man muss dann lediglich eine passende Vergleichsfunktion implementieren und beim Aufruf angeben.

12.6 Beispiele

Beispiel 12.4. Argumente beim Programmaufruf
Beim Aufruf eines Programms kann man Parameter angeben. Typisch ist die

Angabe von Optionen (unter Windows in der Form `/opt`, unter UNIX `-opt`) und Dateinamen. Diese Parameter werden als Argumente an `main` übergeben. Dabei gibt das erste Argument die Anzahl der angegebenen Werte an. Die Werte selbst stehen in einem Feld von Zeigern auf Zeichenketten. Der erste Parameter -– selbst wenn beim Aufruf nichts angegeben wurde — ist der Name der Anwendung.

```
/* Ausgabe der Kommando-Zeile Argumente */
int main(int argc, char* argv[])
{
    int i;

    for( i=0; i<argc; i++ ) {
        printf("%2d: %s\n", i, argv[i]);
    }
}
```

12.7 Übungen

Übung 12.1. Zeiger
Welchen Wert nimmt die Integervariable `i` an:

```
int i, j, *p;
```

1. `j = 1;  p = &j;  i= *p * 3;`
2. `i = 2;  p = &i;  *p += 3;`
3. `p = &i;  *p++ = 5;  ++i;`

Übung 12.2. Zeiger
Gegeben sei

```
int feld1[40], feld2[40], i;
int *zeiger = feld2;
```

Von dem Feld soll eine Kopie erzeugt werden. Welche der folgenden Schleifen zum Kopieren von `feld1` nach `feld2` sind korrekt?

1. `for( i=0; i<40; i++) zeiger[i] = feld1[i];`
2. `for( i=0; i<40; i++) *zeiger++ = feld1[i];`
3. `for( i=0; i<40; i++) *zeiger++ = *feld1++;`
4. `for( i=0; i<40; i++) ++zeiger  = feld1[i];`
5. `for( i=0; i<40; i++) *zeiger++ = *(feld1+i);`

6. `for( i=0; i<40; i++) zeiger[i] = *feld1[i];`

Übung 12.3. Suche
Schreiben Sie eine Version mit Zeigerarithmetik der Funktion

```
char * strstr( char *text, char *muster )
```

Die Funktion durchsucht `text` nach `muster` und gibt entweder einen Zeiger auf die Fundstelle oder NULL - wenn das Suchmuster nicht gefunden wurde - zurück. Wie wird die Funktion eingesetzt, um alle Vorkommen des Musters nacheinander zu finden? Entwickeln Sie ein Hauptprogramm, das in einer Endlos-Schleife Zeilen einliest und jeweils alle Vorkommen eines Suchmusters anzeigt. Das Suchmuster können Sie entweder fest im Programm vorgegeben oder zu Beginn der Anwendung abfragen lassen.

Kapitel 13

Strukturen

13.1 Einleitung

Bisher haben wir neben den Basistypen `int`, `float`, etc. den strukturierten Datentyp Feld benutzt. Ein Feld enthält eine Anzahl von Daten des gleichen Typs. Je nach Anordnung in ein oder zwei Dimensionen entspricht dies Vektoren oder Matrizen. Die einzelnen Elemente eines Feldes werden durch entsprechende Indizierung (oder über Zeiger) angesprochen. Mit dem Feld verbindet man die Vorstellung einer – unter Umständen geordneten – Menge von gleichartigen Daten. Die Position im Feld ist zwar wichtig – denken Sie an ein Zeichen in einem Feld von char – aber man verbindet mit den einzelnen Positionen keine feste Bedeutung. Die Beschränkungen des Datentyps Feld werden sichtbar wenn

- Daten verschiedenen Typs zusammen als ein Objekt modelliert werden sollen

oder

- die einzelnen Elemente über individuelle Namen gemäß ihrer Bedeutung angesprochen werden sollen.

Für diese Fälle stellen Programmiersprachen den Datentyp Verbund bereit. Ein Verbund enthält eine feste Anzahl von Komponenten, die von unterschiedlichem Datentyp sein können. Eine einzelne Komponente innerhalb eines Verbundes wird über den Bezeichner für den Verbund und ihren Komponentenbezeichner angesprochen.

13.2 Strukturen in C

In C wird der Datentyp Verbund durch Strukturen realisiert. Bei der Definition einer Struktur werden die Komponenten (Elemente, Felder, engl. *fields*, *members*) angegeben. Anzahl und Typ der Komponenten können zur Laufzeit nicht mehr

geändert werden. Um beispielsweise die Struktur für einen Punkt mit x- und y-Koordinate anzulegen, kann man schreiben:

```
struct punkt {
    double x;
    double y;
};
```

Damit ist eine Struktur mit dem Namen `punkt` definiert, die zwei Komponenten vom Typ `double` enthält. Nach dieser Definition definiert man Variablen in der Form

```
struct punkt test;
```

Die Elemente werden über ihre Namen in der Syntax

Variablenname . Komponentenbezeichner

angesprochen. Der Punkt-Operator zeigt die Verbindung von Variable und Komponente an. Der folgende Code setzt die Werte in der Variablen `test`:

```
test.x = 0.1;
test.y = 0.2;
```

Umgekehrt gibt

```
printf("Punkt: x = %g, y = %g\n", test.x, test.y );
```

die Werte wieder aus. In diesem einfachen Beispiel könnte man die beiden Koordinaten in eine Feld der Länge 2 abspeichern:

```
double test[2];
```

mit der Vereinbarung, dass `test[0]` die x– und `test[1]` die y-Koordinate enthält. Die Verwendung von Strukturen kann jedoch ein Programm übersichtlicher und damit lesbarer und besser wartbar machen. In der Struktur für einen Punkt ist durch die Bezeichner die Bedeutung der Komponenten offensichtlich. Das richtige Design der Strukturen, so dass sie

- angemessen für die Problemstellung
- gut erweiterbar
- und gleichzeitig effizient in Bezug auf Laufzeit und Speicherbedarf

sind, stellt eine große Herausforderung dar. Notwendig ist einerseits das Verständnis der Problemstellung und andererseits Erfahrung im Entwurf von Anwendungen. Strukturen werden als Einheit behandelt. Man kann

- einer Struktur eine andere Struktur zuweisen (bewirkt die Kopie aller Komponenten)

- Strukturen an Unterprogramme übergeben (Wie bei Basistypen wird eine lokale Kopie erzeugt – call by value.)
- Strukturen von Unterprogrammen als Rückgabewert zurück geben
- Zeiger auf Strukturen setzen
- den benötigten Speicherplatz mit `sizeof` bestimmen
- Vergleichsoperatoren nicht auf Strukturen anwenden

Beispiel 13.1. Struktur Punkt

```
/* Zeige Inhalt einer Struktur Punkt an */
void printPunkt( struct punkt p ) {
    printf("Punkt: x = %g, y = %g\n", p.x, p.y );
}

/* Punkt test ausgeben */
printPunkt( test );

/* jetzt in anderen Punkt kopieren */
andererPunkt = test;
printPunkt( andererPunkt );
```

Strukturen können wiederum Strukturen enthalten. So können aus der Beispiel-Struktur `punkt` komplexere Gebilde aufgebaut werden. Zwei Punkte definieren zusammen ein Rechteck. Eine entsprechende Struktur ist:

```
struct rechteck {
   struct punkt linksUnten;
   struct punkt rechtsOben;
};

/* Zeige Inhalt einer Struktur Rechteck an */
void printRechteck( struct rechteck r ) {
   printf("p1: (%g,%g)\n", r.linksUnten.x, r.linksUnten.y );
   printf("p2: (%g,%g)\n", r.rechtsOben.x, r.rechtsOben.y );
}
```

Bei geschachtelten Strukturen wird der Punkt-Operator entsprechend mehrfach angewandt. Die Reihenfolge geht von links nach rechts. Für kompliziertere Strukturen lohnt eine Funktion zur Initialisierung.

```
/* initialisiere ein Rechteck aus zwei Punkten */
struct rechteck baueRechteck(
```

```
  struct punkt linksUnten,
  struct punkt rechtsOben )
{
    struct rechteck neu;
    neu.linksUnten = linksUnten;
    neu.rechtsOben = rechtsOben;
    return neu;
}

...
    struct rechteck flaeche;
    flaeche = baueRechteck( ersteEcke, zweiteEcke );
```

Variablen können durchaus den gleichen Namen wie Komponentenbezeichner haben. Aus dem Zusammenhang ist die Zuordnung klar und es besteht keine Verwechslungsgefahr. Der gleiche Name kann sogar die Lesbarkeit verbessern.

13.3 Zeiger auf Strukturen

Strukturen werden als Wert an Unterprogramme übergeben. Damit ist der Aufwand des Kopierens der Komponenten verbunden. Weiterhin kann das Unterprogramm keine dauerhaften Änderungen in den Komponenten vornehmen. Daher ist es oft sinnvoll oder notwendig, nicht die Struktur selbst zu übergeben sondern einen Zeiger darauf. Die Syntax ist die gleiche wie bei den Basis-Datentypen. Die Operatoren `&` und `*` ergeben die Adresse einer Struktur bzw. den Inhalt eines Zeigers.

Beispiel 13.2. Zeiger auf Strukturen

```
/* tausche x und y Komponente */
void spiegel( struct punkt *pp ) {
    double tmp = (*pp).x;
    (*pp).x = (*pp).y;
    (*pp).y = tmp;
}
...
    spiegel( &test );
```

Die Klammern in dem Ausdruck `(*pp).x` sind erforderlich, da der Punkt-Operator eine höhere Priorität als die Dereferenzierung hat. `*pp.x` – entspricht `*(pp.x)` – würde den Inhalt eines Zeigers `x` liefern. Die Konstruktion „Komponente des Inhalts eines Zeigers auf eine Struktur“ kommt so häufig vor, dass dafür

ein eigener Operator -> definiert wurde. Aus dem Beispiel wird dann die deutlich übersichtlichere Version

```
void spiegel( struct punkt *pp ) {
    double tmp = pp->x;
    pp->x = pp->y;
    pp->y = tmp;
}
```

Bei komplizierteren Ausdrücken muss man sorgfältig die Reihenfolge der Operatoren beachten. Die Operatoren für Strukturen und die Klammern für Feldindizes binden wie gesagt sehr stark. Oft wird die Bedeutung eines Ausdrucks klar, wenn man sich die Komponenten als Variablen vorstellt, also etwa `pp->zaehler` als `zaehler`. Falls erforderlich kann die Reihenfolge durch Klammern geändert werden.

Beispiel 13.3. Strukturen und Zeiger

```
++pp->zaehler // Erhöht zaehler
(++pp)->zaehler // Zeiger wird auf nächste Struktur gesetzt,
                // dann Inhalt von zaehler benutzt
*p.zeiger // Inhalt von zeiger
*pp->zeiger++ // Inhalt von zeiger benutzen, zeiger inkrementieren
(*pp->zeiger)-- // Inhalt von zeiger wird dekrementiert
```

13.4 Bitfelder und Union △

In Strukturen kann man bei Elementen von ganzzahligen Datentypen die Größe einschränken. Dazu gibt man hinter dem Datentyp einen Doppelpunkt und danach die Anzahl der Bit an. Der Compiler kann dann mehrere Elemente effizient in wenige Speicherstellen packen. Betrachten wir als Beispiel zwei Strukturen für Schrift- und Hintergrundfarbe:

```
struct farbe {
    unsigned int schrift;
    unsigned int hintergrund;
};

struct bsFarben {
    unsigned int schrift:4;
    unsigned int hintergrund:4;
};
```

```
printf("farbe: %d\n", sizeof( farbe ) );
printf("bsFarben: %d\n", sizeof( bsFarben ) );
```

Im ersten Fall enthält die Struktur 2 normale Integerwerte. Dementsprechend gibt `sizeof` eine Größe von 8 an. Manche Betriebssystem verwenden nur 16 Farben. Dann genügen 4 Bit für jede der beiden Farben. Mit der entsprechenden Angabe können die beiden Elemente der Struktur `bSFarben` vom Compiler in ein Speicherwort gelegt werden. Bei meinem Rechner liefert `sizeof` für `bsFarben` dementsprechend den Wert 4, die kleinste Einheit in einem 32-Bit System. Die Werte kann man wie normale Variablen verwenden, wobei natürlich ihr Wertebereich eingeschränkt ist. Größere Werte werden automatisch auf den Wertebereich beschnitten. So ergibt

```
struct bsFarben b1;
b1.schrift = 77;

printf("Schrift %d\n", b1.schrift );
```

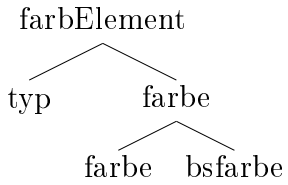
den Wert `77%16`, also `13`. Aufgrund der speziellen Art der Speicherverwaltung kann man die Adresse von Bitfeldern nicht verwenden. Die Anweisung

```
int *j= &b1.schrift;
```

wird vom Compiler nicht akzeptiert. Eine andere Möglichkeit, Speicherplatz zu sparen, besteht bei alternativen Elementen. Wenn man von mehreren Elementen in einer Struktur stets nur eines benötigt, kann man die Elemente in der gleichen Speicherzelle ablegen. Dazu verwendet man anstelle einer Struktur eine `Union`. Die Syntax ist die gleiche:

```
union {
    farbe f;
    bsFarbe bf;
} farbe;
```

definiert einen Speicherbereich, in dem entweder eine normale Farbe oder eine Betriebssystem-Farbe abgelegt ist. Auf die Elemente kann wie bei Strukturen über den Punkt- oder den `->`-Operator zugegriffen werden. In der Regel benötigt man noch eine Variable, die das aktuell gültige Element markiert. Für das Farben-Beispiel können wir folgende Gesamtstruktur verwenden:



Ein allgemeines Farbelement enthält eine Variable mit der Typinformation und eine der beiden Farbinformationen. Mit einem Aufzählungstyp für die beiden möglichen Farbsysteme erhält man folgende Definition:

```
enum farbSystem { normal, bs };

struct farbElement {
    enum farbSystem typ;
    union {
        farbe f;
        bsFarbe bf;
    } farbe;
};
```

Durch die Union wird nur der Speicherplatz für eine Farb-Struktur benötigt. Als Beispiel für die Verwendung soll folgende Funktion dienen:

```
struct farbElement generator() {
    struct farbElement fb;
    if( rand() % 2 == 0 ) {
        fb.typ = normal;
        fb.farbe.f.schrift = rand();
        fb.farbe.f.hintergrund = rand();
    } else {
        fb.typ = bs;
        fb.farbe.bf.schrift = rand() % 16;
        fb.farbe.bf.hintergrund = rand() % 16;
    }
    return fb;
}
```

Bei jedem Aufruf entscheidet der Zufallsgenerator, welche der beiden Farbtypen verwendet wird. Die Typvariable wird entsprechend gesetzt und das zugehörige Element gefüllt. Mit

```
for( i=0; i<10; i++ ) {
    test = generator();
    if( test.typ == normal ) {
        printf("normale Farbe: %d\n", test.farbe.f.schrift );
    } else {
        printf("BS Farbe: %d\n", test.farbe.bf.schrift );
    }
}
```

werden beispielshaft 10 Farbelemente erzeugt und ausgegeben.

13.5 Generator für Landkarten

Ziel ist die Generierung einer Landkarte mit verschiedenen Landschaftstypen als z. B. Fläche für ein Spiel. Zur Vereinfachung führen wir folgende Struktur mit

den Eigenschaften einer Zelle ein:

```
struct zelle {
    char *form;
    int farbe;
    int hintergrund;
    double groesse;
};
```

Funktion zum Malen einer Zelle:

```
void male( int n, struct zelle z) {
    form( n, z.form );
    farbe( n, z.farbe );
    hintergrund( n, z.hintergrund );
    symbolGroesse( n, z.groesse );
}
```

Anlegen von drei Landschaftstypen, speichern in einem Feld und dann Spielfeld zufällig füllen:

```
void fuelle( int N ) {
    struct zelle wald = {"c", GREEN, BROWN, 0.35 };
    struct zelle feld = {"none", BROWN, BROWN, 0.5 };
    struct zelle wasser = {"none", BLUE, BLUE, 0.5 };

    struct zelle auswahl[] = {wald, feld, wasser };

    for( int n=0; n<N; n++ ) {
        printf(" %d ", n );
        int anz = sizeof( auswahl ) / sizeof( auswahl[0] );
        int w = rand() % anz;
        male( n, auswahl[w] );
    }
}
```

Kapitel 14

Präprozessor

Vor der eigentlichen Kompilierung wird bei C-Programmen noch der so genannte Präprozessor aufgerufen. Damit werden Möglichkeiten wie das Einfügen von anderen Dateien sowie die Definition von speziellen Zeichenfolgen bereit gestellt. Weiterhin können mit einfachen Entscheidungsstrukturen der Ablauf der Kompilierung gesteuert werden. Anweisungen für den Präprozessor stehen in eigenen Zeile, die mit einem # beginnen. Präprozessor-Zeilen können an beliebigen Stellen innerhalb der Datei stehen. Der Effekt hält bis zum Ende der Datei an – unabhängig von Funktionsblöcken. Die Verarbeitung endet jeweils am Zeilenende (oder am Beginn eines Kommentars). Die Anweisungen werden **nicht** durch ein Semikolon abgeschlossen. Soll eine Anweisung über das Zeilenende hinaus gehen, so kann man mit einem Schrägstrich \ als letztem Zeichen das Zeilenende unterdrücken.

14.1 Einfügen von Dateien

Man kann eine andere Datei mit dem Befehl `#include` einbinden. Es gibt zwei Varianten:

```
#include "Dateiname"
#include <Dateiname>
```

In beiden Fällen muss der exakte Dateinamen angegeben werden. Platzhalter (*) sind nicht möglich. Allerdings kann durchaus ein Pfadname mit Unterverzeichnissen angegeben werden. Der Präprozessor sucht eine Datei mit dem Namen und fügt den Inhalt in die aktuelle Stelle ein. Der Unterschied zwischen den beiden Formen liegt im Suchmechanismus. Ist der Dateiname in Anführungsstrichen angegeben, so beginnt die Suche im aktuellen Verzeichnis. Falls die Datei dort nicht gefunden wird, werden dann verschiedene Standardverzeichnisse durchsucht. Bei Namen in spitzen Klammern geht der Präprozessor davon aus, dass es sich um Standard- oder Systemdateien handelt und die Suche im aktuellen Verzeichnis entfällt.

Die Details der Suche sind abhängig von der jeweiligen Entwicklungsumgebung. Die Liste der zu durchsuchenden Verzeichnisse ist als Umgebungsvariable INCLUDE oder Option für den Kompileraufruf (/I bei Visual C++) spezifiziert. Durch entsprechende Angaben kann man eigene Verzeichnisse in die Suche einbeziehen. Die Zeile mit der `#include` Anweisung wird durch den Inhalt der Datei ersetzt. Damit kann im Prinzip beliebiger C-Kode in der Datei stehen. Man könnte beispielsweise Funktionen aus anderen Dateien auf diese Weise einfügen. Dies führt allerdings zu schwer lesbarem Kode. Es gibt beliebige Möglichkeiten, die `#include` Anweisungen missbräuchlich einzusetzen und eine schwer zu entwirrende Dateiorganisation zu schaffen. Für einen sinnvollen Einsatz kann man sich an folgenden Regeln orientieren:

- include Dateien nur für allgemeine Definitionen und Deklarationen
- kein ausführbarer Programmteil in include Dateien
- include Dateien können weiter include Dateien einfügen, allerdings sollte Mehrfacheinbindung verhindert werden
- include Anweisungen vor Funktionsblöcke (am einfachsten an den Anfang der Datei)

Nach diesen Regeln sind auch die Definitionen und Deklarationen für die Funktionen der Standardbibliotheken organisiert. Man spricht dann auch von Header Dateien und benutzt die Endung .h für diese Dateien (z. B. stdio.h oder conio.h). Bei Visual C finden sich diese Header Dateien im Verzeichnis

```
c:\Programme\Microsoft Visual Studio\VC98\Include
```

und den darunter liegenden Unterverzeichnissen. Allgemein kann man Header Dateien nutzen, um Informationen für mehrere Dateien eines Projektes an einer Stelle zentral zu halten. Betrachten wir ein Projekt mit 3 Dateien

```
haupt.c, unterteil1.c, funktionen.c
```

In `funktionen.c` stehen einige Funktionen, die in den beiden anderen Dateien aufgerufen werden. Daher müssen diese Funktionen in diesen beiden Dateien deklariert werden. Man sollte die Funktionsdefinitionen in einer gemeinsamen Header Datei (z. B. funktionen.h) schreiben und dann diese Datei zweimal einbinden. Auf diese Art brauchen eventuelle Änderungen nur an einer Stelle vorgenommen zu werden. Ähnliches wie für Funktionen gilt auch für globale Variablen.

14.2 Definitionen

Die einfachste Definition im Präprozessor hat die Form

```
#define Name Text
```

Für die Bildung von Name gelten die gleichen Regeln wie für Variablennamen. Zur besseren Unterscheidung schreibt man die define Namen oft in Großbuchstaben. Die define Anweisung verknüpft diesen Namen mit dem dahinter stehenden Text. Überall wo im folgenden Programmkode der Name auftaucht, wird er durch diesen Text ersetzt.

Beispiel 14.1. Definition

```
#define MAXLAENGE 100
...
char zeile[MAXLAENGE];
```

In dieser Verwendung sind die define Konstanten ähnlich zu den mit const definierten Variablen. Aber die define Konstanten sind selbst keine Variablen: es wird kein Speicherplatz angelegt und sie sind auch nicht mit einem Datentyp verbunden. Man kann sie sehr viel freier verwenden und beispielsweise beliebige Textersetzungen realisieren.

```
#define endlos for( ;; )
...
endlos{
}
```

In diesem Fall wird der Name endlos vor der Kompilierung durch die Schleife ersetzt. Man kann in den Definitionen auf vorhergehende Definitionen aufbauen:

```
#define FALSE 0
#define TRUE ! FALSE
...
    printf("TRUE: %d FALSE: %d\n", TRUE, FALSE);
```

Die beiden Texte TRUE und FALSE innerhalb der Zeichenkette werden nicht ersetzt. Die Ersetzungsregel wird nur angewandt, wenn der Name als solcher erkennbar ist, d. h. nicht innerhalb einer Zeichenkette und nicht als Teil als längeren Namens (z. B. TRUECOLOR). Eine Definition gilt bis zum Ende der Datei. Bei einer erneuten Definition mit einem anderen Wert meldet der Präprozessor eine Warnung. Der Ersetzungstext kann auch fehlen. Die define Konstante ist dann zwar definiert, aber der Name wird überall ersatzlos entfernt. Man muss sich stets bewusst sein, dass lediglich eine Textersetzung statt findet. Anders als bei der Definition von Variablen wird ein Ausdruck nicht ausgewertet. Ein häufiger Fehler ist in der folgenden Konstruktion:

```
#define MLAENGE 100
#define MLAENGEP1 MLAENGE + 1
```

```
...
    printf( "MLAENGEP1 = %d, ", MLAENGEP1);
    printf( "2 * MLAENGEP1 = %d\n", 2 * MLAENGEP1);
```

führt zu

```
MLAENGEP1 = 101, 2 * MLAENGEP1 = 201
```

Durch die Textersetzung steht im Argument des zweiten printf der Ausdruck 2 * 100 + 1 und damit führt die Rechnung aufgrund der Vorrangregeln nicht zu dem gewünschten Ergebnis. Durch zusätzliche Klammern kann dieser Fehler vermieden werden:

```
#define MLAENGEP1 (MLAENGE + 1)
```

liefert das richtige Resultat.

14.3 Makros

Neben der einfachen Textersetzung sind auch Makros mit Argumenten möglich. Ähnlich wie bei Funktionen gibt man die Argumente in Klammern an. Ein Beispiel für ein Makro zur Berechnung von Quadraten ist:

```
#define QUAD( x ) ((x) * (x))
...
printf( "quad(5) = %d, quad( 5+1 ) = %d\n",
QUAD(5), QUAD(5+1) );
```

ergibt

```
quad(5) = 25, quad( 5+1 ) = 36
```

Die Klammern sind wiederum wichtig für das richtige Rechnen. Es ist generell eine sinnvolle Vorsichtsmaßnahme den Makroausdruck in Klammern zu setzten. Dadurch ist gewährleistet, dass der Ausdruck zusammen ausgewertet wird.

Übung 14.1. Makros
Was ergibt

```
#define QUAD( x ) x * x
...
printf( "quad(5) = %d, quad( 5+1 ) = %d\n",
QUAD(5), QUAD(5+1) );
```

Anders als Funktionen sind Makros neutral gegenüber dem Typ der Argumente. Das Makro `quad` kann in dieser Form sowohl auf Integer- als auch Gleitkommaausdrücke angewandt werden. Da ein Makro einfach in den Programmtext hinein kopiert wird, ist die Ausführung mit keinerlei Mehraufwand verbunden. Allerdings wird auf der anderen Seite auch keine Wissen bei der Auswertung eingesetzt. Betrachten wir den Ausdruck

```
QUAD( sin( omega ) )
```

Daraus wird durch die Expansion des Makros

```
sin( omega ) * sin( omega )
```

die Sinus-Funktion wird also zweimal aufgerufen. In diesem Fall bedeutet dies lediglich einen zusätzlichen Rechenaufwand. Wenn allerdings die Funktion oder allgemeiner der Ausdruck im Argument Seiteneffekte hat, treten diese auch zweimal auf. In Fällen wie

```
QUAD( iwert++ )
```

wird die Variable `iwert` auch zweimal erhöht. Man kann das Makro weiter absichern, indem man das Argument in eine Variable kopiert:

```
static float quad_arg;
#define QUAD( x ) (quad_arg=(X), quad_arg*quad_arg)
```

Das Makro ist jetzt relativ sicher, aber nur für einen Datentyp und auf Kosten einer zusätzlichen Variablen. In Zeichenketten findet keine Makro Ersetzung statt. Wenn allerdings bei dem Makrotext der Name eines Arguments mit einem # davor steht, wird das Argument in eine Zeichenketten eingebettet.

```
#define dprint( expr ) printf( #expr " = %g\n", expr)
dprint( sin( 3.14 ) );
```

Durch die Expansion wird daraus

```
printf( " sin( 3.14 ) " " = %g\n", sin( 3.14 ) )
```

Zwei direkt aufeinanderfolgende Zeichenketten werden in C zusammen gefasst zu einer einzigen Zeichenkette, so dass folgender Ausdruck entsteht:

```
printf( " sin( 3.14 ) = %g\n", sin( 3.14 ) )
```

und das Resultat bei der Ausführung ist

```
sin( 3.14 ) = 0.00159265
```

14.4 Bedingte Compilierung

Mit der Anweisung `#if` Ausdruck wird eine bedingte Compilierung eingeleitet. Ergibt der Ausdruck 0 werden die folgenden Anweisungen bis zu einem abschließenden `#endif` übersprungen. Der Ausdruck kann beliebige Integer-Konstanten, Zeichenkonstanten und insbesondere define Konstanten enthalten.

Beispiel 14.2. Bedingte Compilierung

```
#define MLAENGE 100

#if MLAENGE > 50
...
#endif
```

Als weiteres Kontrollelement gibt es `#else` und `#elif` (else if) Anweisungen. Eine typische Anwendung ist die Auswahl der richtigen Version einer Include Datei in Abhängigkeit vom System. Wenn in der define Konstanten SYSTEM der Name des Betriebssystems steht, kann man folgende Struktur verwenden:

```
#if SYSTEM == MSDOS
    #define HDR "msdos.h"
#elif SYSTEM == LINUX
    #define HDR "linux.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Die Anführungsstriche sind Bestandteil des Ersetzungstextes. Diese Beispiel zeigt eine gebräuchliche Methode, Abhängigkeiten vom Betriebssystem zu isolieren. Wenn die Konstante SYSTEM richtig gesetzt ist, braucht sich der Anwender nicht um systemspezifische Details zu kümmern. Die Konstruktion `defined( Name )` prüft, ob der angegebene Name definiert ist. Damit kann man das Vorhandensein von Konstanten zur Entscheidung benutzen.

Beispiel 14.3. Bedingte Compilierung

```
#if defined( DEBUG)
     printf( "Einige zusaetzliche Informationen\n" );
#endif
```

Die defined-Abfrage kann ausschließlich in `#if` und `#elif` Anweisungen eingesetzt werden. Da diese Abfragen recht häufig vorkommen, gibt es die Kurzformen `#ifdef` und `#ifndef` um zu testen, ob eine Konstante definiert ist oder nicht. Damit kann man für Testzwecke leicht auch größer Kodeabschnitte ausblenden, indem man sie in einen

```
#ifdef
   IRGENDWAS ...
#endif
```

Block einschließt. Da die Konstante nicht definiert ist, wird der ganze Abschnitt bei der Kompilierung übersprungen. Eine weitere Standardanwendung besteht

im Test, ob eine Header Datei bereits eingebunden wurde. Wird der eigentliche Inhalt der Header Datei in eine Abfrage in der Art

```
#ifndef DIESE_DATEI
#define DIESE_DATEI
...
#endif
```

eingebunden, so ist sichergestellt, dass der Inhalt nur einmal ausgeführt wird. Beim ersten Aufruf ist `DIESE_DATEI` noch nicht definiert. In der nächsten Anweisung wird die Konstante gesetzt, so dass bei einem weiteren Aufruf der Block nicht mehr betreten wird. Mit dieser Vorsichtsmaßnahme ist es unkritisch, eine header Datei mehrfach einzubinden.

14.5 Sonstiges

Von den anderen Möglichkeiten des Präprozessors sind nur drei von größerer Bedeutung:

```
#undef
#error
#pragma
```

Mit `#undef` Name wird eine eventuelle Definition von Name gelöscht. Die `#error` Anweisung beendet die Kompilierung mit einer Fehlermeldung. Ein optionaler Text in der Anweisung wird als Fehlertext angezeigt.

Beispiel 14.4. Fehlermedlungen

```
#ifndef LINUX
    #error LINUX ist erforderlich.
#endif
```

Wenn LINUX nicht definiert ist, bricht der Kompiler ab. Spezielle Anweisungen für einzelne Compiler können in `#pragma` Anweisungen stehen. Trifft der Compiler auf ein unbekanntes pragma, wird lediglich eine Warnung ausgegeben und ansonsten die Zeile ignoriert und die Kompilierung fortgesetzt. Die speziellen pragmas für den jeweiligen Compiler findet man in der zugehörigen Dokumentation. Eine interessantes pragma bei Visual C++ ist `message( text )`. Der Text im Argument wird beim Kompilieren ausgegeben.

Beispiel 14.5. Meldungen

```
#pragma message( "Einige Meldungen vom Programmierer:")
```

14.5.1 Vordefinierte Namen

Einige Namen sind vorab definiert. Standardmäßig sind beispielsweise definiert:

`__FILE__`	Der Dateiname
`__LINE__`	Die aktuelle Zeilennummer

Daneben biete Visual C++ weitere Konstanten wie `__TIMESTAMP__` für Datum und Uhrzeit der letzten Änderung der aktuellen Quelldatei. Diese Konstanten kann man in Meldungen integrieren:

```
#pragma message( "Kompilieren von \n" __FILE__ )
#pragma message( "Zuletzt geändert am " __TIMESTAMP__ )
```

14.6 Visual C

In Visual C++ werden alle Dateien – C Code und Header Dateien – für ein Projekt gemeinsam verwaltet. In einem Projekt werden neue Dateien für den Menüpunkt „Neu“ eingefügt. Alle C Dateien werden kompiliert und zusammen gelinkt. Daher darf auch nur eine Datei in einem Projekt einen `main()` Block enthalten. Um beim Kompilieren Zeit zu sparen, können Teile des C-Kodes und insbesondere Header Dateien in einer vorkompilierten Form abgelegt werden. Bei weiteren Kompilierungen wird darauf zurück gegriffen. Standardmäßig wird bei einem neuen Projekt der Einsatz von vorkompilierten Headern eingestellt und ein entsprechendes

```
#include "stdafx.h"
```

eingebaut. Dies führt zu Problemen, wenn man die Datei später mit dem Befehl `cl` in einem Eingabefenster übersetzen will. Es ist daher empfehlenswert, die entsprechende Einstellung zu ändern und im Menüpunkt Projekt – Einstellungen – C/C++ in der Kategorie *vorkompilierte Header* auf *vorkompilierte Header nicht verwenden* zu ändern.

14.7 Beispiele

Beispiel 14.6. Aufteilung Funktion und Hauptprogramm
Dieses Beispiel zeigt, wie mit einer include Datei die Definition für eine Funktion eingebunden wird.

`fkt.cpp:`

```
#include <stdio.h>

void fkt( void) {
    printf( "in Funktion fkt\n");
```

```
}

fkt.h:
#pragma message( "jetzt sind wir im header fkt.h" )
void fkt( void );

haupt.c:
#include "stdio.h"
#include "fkt.h"     /* damit fkt bekannt ist */

main()
{
    printf( "Funktion fkt aufrufen \n" );
    fkt( );
}
```

14.8 Übungen

Was bewirkt der Code in den folgenden Übungen?

Übung 14.2. Define

```
#define LF printf("\n")
#define ASTERIX printf("*************\
*****************************************\n")
...
LF;
ASTERIX;
LF;
```

Übung 14.3. ifdef

```
#ifdef SUN
#include </usr/demo/SOUND/include/multimedia/libaudio.h>
#else
short audio_u2s( unsigned char wert ) {
   printf("function audio_u2s not implemented on PC\n");
   exit(1);
}
unsigned char audio_s2u( short wert ) {
   printf("function audio_s2u not implemented on PC\n");
   exit(1);
```

```
}
#endif
```

Übung 14.4. ifdef

```
#ifdef MAINPROGRAM
#define STATUS
#define VALUE(x) =x
#else
#define STATUS extern
#define VALUE(x)
#endif

STATUS int verbose VALUE( 0 );
STATUS int level VALUE( 5 );
```

Kapitel 15

Dateien

15.1 Öffnen von Dateien

Bisher erfolgte die Ausgabe stets in das Konsolenfenster und die Eingabe kam von der Tastatur. Im allgemeinen ist es aber auch erforderlich, aus Dateien (engl. *files*) zu lesen und in Dateien zu schreiben. In C stehen dazu eine Reihe von Funktionen zur Verfügung. Damit kann der Entwickler -– ohne sich um die Details der Realisierung auf dem System kümmern zu müssen -– in einfacher Weise Datei anlegen, beschreiben, lesen, löschen, etc. Kernelement ist dabei ein so genannter *file pointer* (Dateizeiger). In `stdio.h` ist ein Typ `FILE` definiert. Für die Dateibearbeitung benutzt man dann Zeiger auf FILE Objekte. Im ersten Schritt öffnet man eine Datei zum Lesen und / oder Schreiben. Dazu dient die Funktion

```
FILE *fopen( char * Name, char * Modus )
```

Die Funktion hat zwei Argumente: den Dateinamen und den Modus, in dem die Datei geöffnet wird. Rückgabewert ist ein Dateizeiger. Ein Rückgabewert von NULL zeigt an, dass die Datei nicht geöffnet werden konnte.

Beispiel 15.1. Dateien

```
char fname[200];
FILE *fp;

printf( "Datei: ");
scanf( "%s", fname ); /* Einlesen des Dateinamens */
fp = fopen( fname, "r" ); /* Öffnen zum Lesen */
if( fp == NULL ) {
    printf( "Sorry, Datei kann nicht geoeffnet werden!\n");
} else {
    printf( "Glueckwunsch, Datei wurde geoeffnet!\n");
}
```

Der Dateiname kann entweder relativ zum aktuellen Verzeichnis oder absolut mit Laufwerk und gesamtem Pfad angegeben werden. Im einzelnen gelten die Regeln des aktuellen Betriebssystems. Der Dateimodus bestimmt, welche Operationen mit der Datei möglich sind:

r	Nur Lesen, Datei muss existieren
w	Nur Schreiben, Datei wird bei Bedarf neu erzeugt, eventueller alter Inhalt geht verloren
a	Anhängen, eventueller alter Inhalt bleibt erhalten
r+	Schreiben und Lesen (update), Datei muss existieren
w+	Schreiben und Lesen, eventueller alter Inhalt geht verloren
a+	Anhängen und Lesen, eventueller alter Inhalt bleibt erhalten

Weiterhin unterscheiden manche Betriebssysteme (z. B. NT) zwischen Dateien in Textmodus und Binärmodus. Eine Textdatei besteht aus einer Folge von Zeilen, während Binärdateien die Daten als unstrukturierten Strom von Daten enthalten. Der Modus kann durch Angabe von `t` (in der Regel Standard Vorgabe) oder `b` ausgewählt werden.

Beispiel 15.2. Modus

```
fp=fopen("beispiel.txt", "r" ); /* nur Lesen */
fp=fopen("daten.bin", "wb" ); /* nur Schreiben, binär */
fp=fopen( dFile, "a+b" ); /* Anhängen + Lesen, binär */
```

Nach dem Öffnen der Datei erfolgt der Zugriff über den Dateizeiger. Dazu gibt es spezielle Varianten der uns bekannten Ein- und Ausgabefunktionen. Die Varianten haben den Dateizeiger als zusätzliches Argument. Der Name der Funktion beginnt mit einem „f“. So ist z. B.

```
int fprintf( FILE *stream, const char *format [, argument ]...);
```

die entsprechende Variante von `printf`. Ein anderes Beispiel ist

```
char *fgets( char *string, int n, FILE *stream );
```

Leider ist die Position des Dateizeigers in der Argumentliste nicht konsistent. In manchen Fällen ist er das erste und in anderen Fällen das letzte Argument. Die Funktionen lesen soviel Daten wie gefordert bis zum Ende der Datei. Anhand der Rückgabewerte kann man feststellen, ob alle Werte gefunden wurde. `fgets` gibt den Wert NULL zurück, wenn das Dateiende erreicht ist oder ein Fehler auftritt. Im folgenden Beispiel werden alle Zeilen gelesen:

```
char line[200];
int n = 0;
while( fgets( line, 200, fp ) ) ++n;
printf( "%d Zeilen gefunden\n", n );
```

Am Dateiende wird der Ausdruck in der while Bedingung NULL und die Schleife bricht ab. In diesem einfachen Beispiel wird davon ausgegangen, dass keine Fehler auftreten. Man kann auch explizit prüfen, ob der Abbruch durch Dateiende oder Fehler verursacht wurde. Dazu gibt es das Funktionspaar

```
int feof( FILE *stream );
/* Rückgabewert ungleich Null bei Dateiende */
int ferror( FILE *stream );
/* Rückgabewert ungleich Null bei Fehler */
```

Die aufgetretene Fehlerbedingung kann mit der Funktion

```
void perror( const char *string );
```

ausgegeben werden. String ist dabei ein frei wählbarer Text. Die Funktion `perror` kann generell zur Ausgabe von Fehlermeldung eingesetzt werden. Typisch ist die Ausgabe von Fehlern beim Öffnen von Dateien in der Art

```
fp = fopen( fname, "r" );
if( fp == NULL ) {
    printf( "Sorry, kann Datei nicht oeffnen!\n");
    perror( fname );
}
```

mit z.B. der Ausgabe

```
Sorry, kann Datei nicht oeffnen!
datei.txt: No such file or directory
```

Die Verbindung zu einer Datei kann wieder unterbrochen werden. Dazu wird die Funktion

```
int fclose( FILE *stream );
```

aufgerufen. Es kann sinnvoll sein eine Datei abzuschließen,

- um versehentliches Schreiben in die Datei zu vermeiden
- weil das Betriebssystem die Anzahl der gleichzeitig offenen Dateien beschränkt

Ansonsten werden alle bei Programmende noch offenen Dateien automatisch geschlossen.

15.2 Positionierung

In manchen Fällen ist es sinnvoll, ab einer bestimmten Stelle in der Datei zu lesen oder gezielt an diese Stelle zu schreiben. Die einfachste Möglichkeit ist, den Dateizeiger wieder auf den Anfang der Datei zu setzen. Dazu dient die Funktion `rewind` (zurückspulen):

```
void rewind( FILE *stream );
```

Mit dem Aufruf von `rewind` wird der Schreib- bzw. Lesezeiger wieder auf den Anfang der Datei gestellt. Gleichzeitig werden eventuelle Fehlerindikatoren gelöscht. Eine flexiblere Positionierung ermöglicht die Funktion `fseek` mit der Syntax

```
int fseek( FILE *stream, long offset, int origin );
```

Origin (Ursprung) kann dabei die Werte

- `SEEK_SET` Dateianfang
- `SEEK_CUR` aktuelle Position
- `SEEK_END` Dateiende

annehmen. Bei Dateien im Binärmodus gibt `offset` die Anzahl der zusätzlichen Bytes an, die zu der angegebenen Position addiert werden. Bei Dateien im Textmodus kann man die Anzahl von Bytes nicht direkt angeben, da an den Zeilenenden und am Dateiende eine Umsetzung zwischen den Zeichen in der Datei und den zurück gelesenen Zeichen erfolgt. Daher ist im Textmodus nur der Offset 0 sicher. Alternativ kann man mit

```
long ftell( FILE *stream );
```

sich die aktuelle Position geben lassen. Dieser Wert kann später wieder in `fseek` eingesetzt werden.

Beispiel 15.3. Positionierung

```
long position;
...
position = ftell( fp ); /* Position merken */
...
fseek( fp, position, SEEK_SET); /* zurück zur Position */
```

15.3 Unformatierte Ein- und Ausgabe

Mit den bisher besprochenen Funktionen werden die Daten formatiert geschrieben und gelesen. Daneben gibt es die Möglichkeit, mit den unformatierten Daten – d. h. direkt mit den Bitmustern – zu arbeiten. Die Dateien sind dann zwar nicht mehr direkt lesbar, aber die Speicherung erfordert deutlich weniger Platz. Beispielsweise erfordert ein short Wert dann auch nur 2 Bytes Speicherplatz, während bei der formatierten Speicherung jede Ziffer ein Byte benötigt. Neben dem Gewinn an Speicherplatz hat man eine höher Geschwindigkeit beim Lesen und Schreiben. Zum einen müssen weniger Daten transferiert werden und zum anderen ist keinerlei Umsetzung mehr erforderlich. Die Funktionen zum direkten Schreiben und Lesen sind:

```
size_t fwrite( const void *buffer, size_t size, size_t count,
                                                    FILE *stream );
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

Das erste Argument ist ein Zeiger auf einen Speicherbereich, das zweite gibt die Größe eines Objektes und das dritte die Anzahl der Objekte an.

Beispiel 15.4. Unformatierter IO

```
char text[20];
float feld[15];
int nread, nwritten;
FILE *fp;
...
/* 20 char Objekte */
nwritten = fwrite( text, sizeof( char ), 20, fp );
/* 15 float Objekte */
nread = fread( feld, sizeof( float ), 15, fp );
/* 1 Feld Objekt */
nread = fread( feld, sizeof feld, 1, fp );
```

15.4 Standard Ein- und Ausgabe

Die Verwendung von FILE Zeigern ist nicht auf Dateien im Sinne von „Daten auf der Festplatte“ beschränkt. Vielmehr ist es ein allgemeines Konzept für Quellen und Senken von Bitströmen. Man spricht daher in diesem Zusammenhang im Englischen von *Streams*. Auch Tastatur und Bildschirm werden als Datenströme behandelt. Standardmäßig sind dafür die Dateizeiger

stdin	Eingabe
stdout	Ausgabe
stderr	Ausgabe für Fehlermeldungen

definiert. Diese Dateizeiger können in den oben beschriebenen Funktionen eingesetzt werden:

```
fprintf( stdout, ...
```

entspricht

```
printf( ...
```

Allerdings ist das Verhalten der Positionierungsbefehle auf solche Datenströme undefiniert. Die Zeiger sind als konstant definiert, Man kann also nicht einfach

```
stdout = fp_mein; // so geht's nicht
```

schreiben. Sie können aber mit der Funktion

```
FILE *freopen( const char *path, const char *mode, FILE *stream );
```

umdefiniert werden. Die standardmäßig vorgegebenen Ein- und Ausgabeströme können auch beim Aufruf mit den Operatoren > und < umgelenkt werden. Startet man eine Anwendung in dem Konsolenfenster mit dem Befehl

```
test.exe > listing.log
```

werden alle Bildschirmausgaben in die Datei `listing.log` geschrieben. Die Fehlermeldungen erscheinen aber nach wie vor am Bildschirm.

15.5 Pufferspeicher

Schreibt man einen Wert in eine Datei, so wird im allgemeinen noch keine Schreibaktion auf dem Medium ausgelöst. Es ist ineffektiv, für jedes Zeichen sofort die Festplatte zu positionieren und das Zeichen zu übertragen. Statt dessen werden Zeichen intern zunächst in Zwischenspeicher — Pufferspeicher, engl. *Buffer* -– gesammelt. Erst wenn ein Bereich voll ist, erfolgt der Transfer. Im Normalfall läuft dieser Prozess im Hintergrund und braucht nicht berücksichtigt zu werden. Allerdings kann dies im Fehlerfall — wenn eine Anwendung abstürzt -– dazu führen, dass ein Teil der Daten noch im Pufferspeicher steht und noch nicht in der Ausgabedatei. Für solche Fälle kann man die Schreibaktion gezielt mit dem Aufruf der Funktion

```
int fflush( FILE *stream );
```

auslösen. Mit dem Aufruf der Funktion wird unabhängig vom Füllstand des Pufferspeichers ein Schreibenvorgang ausgeführt.

15.6 Dateimanipulationen

In `stdio.h` sind zwei Funktionen zum Löschen und Umbenennen von Dateien definiert:

```
int remove( const char *path );
int rename( const char *oldname, const char *newname );
```

15.7 Beispiele

Beispiel 15.5. Allgemeines

```
#include "stdafx.h"
#include "stdlib.h"
#include "string.h"

void fill( FILE *fp ) {
    int i;
    for( i=0; i<77; i++ ) fprintf( fp, "%d\n", i );
}

void numLines( FILE *fp ) {
    char line[200];
    int n = 0;
    while( fgets( line, 200, fp ) ) ++n;
    printf( "%d Zeilen gefunden\n", n );
}

int main(int argc, char* argv[])
{
    char fname[200];
    char *test = "test.dat";
    FILE *fp;

    if( argc > 1 ) strcpy( fname, argv[1] );
    else {
        printf( "Datei: ");
        scanf( "%s", fname );
    }
    fp = fopen( fname, "r" );
    if( fp == NULL ) {
        printf( "Sorry, kann Datei nicht oeffnen!\n");
        perror( fname );
    } else {
        printf( "Glueckwunsch, Datei wurde geoeffnet!\n");
    }

    /* einige Tests mit Dateimodus */
    remove( test );
    printf( "Datei <%s> geloescht\n", test);

    fp = fopen( test, "r" );
    printf( "fp bei oeffnen zum Lesen : %p\n", fp );
    fp = fopen( test, "w" );
    printf( "fp bei oeffnen zum Schreiben : %p\n", fp );
```

```
    fill( fp );
    fclose( fp );
    fp = fopen( test, "r" );
    printf( "fp bei oeffnen zum Lesen : %p\n", fp );
    numLines( fp );
    fclose( fp );

    printf( "Datei mit w oeffnen und schliessen\n");
    fp = fopen( test, "w" );
    fclose( fp );
    printf( "Wieder Zeilen zaehlen\n");
    fp = fopen( test, "r" );
    numLines( fp );
    fclose( fp );
    return 0;
}
```

Ergibt:

```
Datei <test.dat> geloescht
fp bei oeffnen zum Lesen : 00000000
fp bei oeffnen zum Schreiben : 00426AC8
fp bei oeffnen zum Lesen : 00426AC8
77 Zeilen gefunden
Datei mit w oeffnen und schliessen
Wieder Zeilen zaehlen
0 Zeilen gefunden
```

Beispiel 15.6. Unformatierte Ein- und Ausgabe

```
float feld[15];

printf( "Direkt IO\n");
/* Datei oeffnen */
fp = fopen( "direkt.dat", "w+b" );

/* 2 mal schreiben */
nw = fwrite( feld, sizeof feld, 1, fp);
printf( "%d Werte geschrieben\n", nw);
nw = fwrite( feld, sizeof( float ), 15, fp);
printf( "%d Werte geschrieben\n", nw);

/* jetzt wieder lesen */
rewind( fp );
```

```
for( i=0; i<4; i++ ) {
    nr = fread( feld, sizeof( float ), 10, fp);
    printf( "%d: %3d Werte gelesen\n", i, nr);
}
fclose( fp );
```

Ergibt:

```
Direkt IO
1 Werte geschrieben
15 Werte geschrieben
0:  10 Werte gelesen
1:  10 Werte gelesen
2:  10 Werte gelesen
3:   0 Werte gelesen
```

15.8 Übungen

Kapitel 16

Sortierverfahren

16.1 Einleitung

Eine der am häufigsten vorkommenden Aufgabe in der Informationstechnik ist das Sortieren oder Ordnen von Daten. Typische Anwendungen sind:

- Namenslisten alphabetisch sortieren (Einträge in Telefonbuch, Kundenlisten, o.ä.)
- Ordnen der Treffer bei einer Internetsuche nach der jeweiligen Bewertung
- Sortieren von Angeboten bei Auktionshäusern nach unterschiedlichen Kriterien (Kategorie, Preis, Enddatum)
- Datei-Listen für die Darstellung im Explorer

Neben dem eigentlichen Ziel – der Herstellung einer Ordnung – kann man Sortierverfahren auch benutzen, um festzustellen, ob in einer Liste Einträge mehrfach vorhanden sind. Anstatt jeden Eintrag mit jedem zu vergleichen ordnet man alle Einträge. Gleiche Einträge stehen dann direkt hintereinander. D. Knuth zitiert eine Abschätzung der Herstellerfirmen, wonach in den 60ern die Computer etwa 25% ihrer Zeit mit Sortierverfahren verbringen (D. Knuth, *The Art of Computer Programming, Vol. 3 Sorting and Searching*).

Sortierverfahren werden eingesetzt, um Daten in die numerische oder alphabetische Reihenfolge zu bringen. Häufig wird nur ein Bestandteil eines komplexen Datensatzes zum Sortieren verwendet (z. B. der Name eines Kunden aus dem gesamten Datensatz mit Adresse, Einkäufen, etc. oder der Interpret zu einer MPEG Musikdatei). Man spricht dann vom Sortierschlüssel oder einfach vom Schlüssel. Wenn die Datenmengen nicht zu groß sind, kann man sie zum Sortieren komplett in den Speicher laden (internes Sortieren). Dabei kann man zwischen Verfahren unterscheiden, die in der Liste sortieren und keinen oder nur geringen zusätzlichen Speicher benötigen sowie Verfahren die zusätzlichen Speicherplatz für eine

teilweise oder vollständige Kopie der Daten benötigen. Ansonsten - wenn man stets nur Teile der Daten von z. B. der Festplatte laden kann - muss man externe Sortierverfahren einsetzen.

Die folgende Darstellung beschränkt sich auf interne Verfahren ohne zusätzlichen Speicherplatz. Dabei betrachten wir numerische Daten (`int` Werte). Die zu sortierenden Werte sollen in einem Feld stehen. Die Verfahren lassen sich aber auch auf andere Datentypen und insbesondere Zeichenketten anwenden. Eine sehr ausführliche Darstellung findet man in R. Sedgewick, *Algorithmen in C++* [Sed94]. Das Problem besteht darin, in einem gegebenes Feld mit `int` Werten, die einzelnen Werte so zu vertauschen, dass sie anschließen in absteigender (aufsteigender) Reihenfolge sind:

135	-78	399	...	2004

nach

2004	399	135	...	-78

16.2 Kriterien

Wesentliche Kriterien für die Leistungsfähigkeit eines Sortierverfahrens (Algorithmus) sind Laufzeit und Speicherbedarf. Um das Verhalten im Detail zu bewerten, kann man folgende Abhängigkeiten untersuchen:

- Abhängigkeit von der Anzahl N der Daten (Feldgröße)
- Abhängigkeit von der Anfangsordnung und den jeweiligen Werten
- durchschnittliche Laufzeit
- minimale Laufzeit (günstigster Fall)
- maximale Laufzeit (ungünstiger Fall)

Derartige Untersuchungen sind Gegenstand der mathematische Analyse von Algorithmen. Ziel dabei ist, unabhängig von einer speziellen Plattform und Realisierungsdetails grundsätzliche Bewertungen zu erhalten. Typische Aussagen über die Komplexität haben die Form $O(f(N))$, wobei N die Anzahl der zu verarbeitenden Werte bezeichnet. Der Ausdruck $O(f(N))$ besagt, dass die Komplexität proportional ist zu einer Funktion $f(N)$. Mit anderen Worten, es gibt zwei Konstanten a und b, so dass für die Anzahl der Rechenschritte die Formel

$$a \cdot f(N) + b \tag{16.1}$$

gilt. Bei einem Algorithmus mit $O(N)$ etwa steigt der Aufwand proportional zu Anzahl der Daten. Demgegenüber würde bei einem Algorithmus mit $O(N^2)$ der

Aufwand quadratisch ansteigen. Solche Bewertungen liefern wertvolle Hinweise für die Entscheidung über den Einsatz eines bestimmten Algorithmus. Allerdings müssen die Randbedingungen der geplanten Anwendung einbezogen werden. Es gilt zu abzuschätzen:

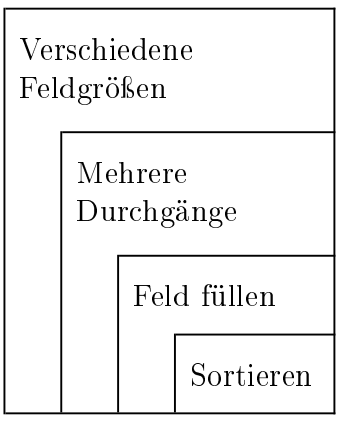
- wie groß werden die Felder sein?
- wie wahrscheinlich kommt der ungünstigste Fall vor?
- wie sind die Anforderungen an die Reaktionszeit?

Neben der theoretischen Analyse von Algorithmen kann man für eine konkrete Realisierung das Laufzeitverhalten anhand von typischen Testfällen messen. Dieses Vorgehen bedingt, dass man den Algorithmus implementiert und die richtige Funktionsweise sicher stellt. Dafür erhält man konkrete Ergebnisse über Laufzeit und eventuell Speicherbedarf. Im folgenden wird der Ansatz einer gemischten Darstellung verfolgt. Bei den Algorithmen werden einerseits ihre grundlegenden Eigenschaften diskutiert. Andererseits werden anhand von beispielhaften Implementierungen das Verhalten untersucht.

16.2.1 Rahmenprogramm

Um das Laufzeitverhalten zu untersuchen, benötigt man ein Hauptprogramm, das Testdaten erzeugt, Methoden mit Sortierverfahren aufruft und dabei die Laufzeit für verschiedene Konfigurationen misst. Als Beispiel sollen Felder mit Ganzzahligen Werten sortiert werden. Die Felder werden zunächst mit Zufallszahlen gefüllt. Dazu wird die Standardfunktion rand() eingesetzt. Die Funktion liefert eine Zufallszahl zwischen 0 und 32767 (Konstante `RAND_MAX` in stdlib.h) zurück. Wenn nicht anders initialisiert (Funktion `srand( unsigned int seed)`, beginnt die Zufallszahlenfolge stets mit dem gleichen Wert. Das Struktogramm für das allgemeine Testprogramm sieht wie folgt aus:

Nassi-Shneiderman — Sortieren



In der äußeren Schleife wird die Feldgröße variiert. Falls gewünscht, können mit einer Feldgröße mehrere Durchläufe erfolgen. In jedem Durchlauf wird das Feld neu mit Zufallszahlen gefüllt und anschließend sortiert. Das entsprechende C-Programm dazu ist:

```
int main(int argc, char* argv[])
{
    int const MAX = 100000;
    int feld[MAX];
    int laenge;
    int durchgang, wiederholungen = 1;
    int i;
    clock_t start, finish; /* Variablen fuer Zeitmessung */
    double elapsed_time;

    printf( "RAND_MAX = %d\n", RAND_MAX);

    /* Feldlaenge variieren */
    for( laenge=10; laenge<=30000; laenge += 1000 ){
        start = clock();
        /* mehrere Durchgaenge */
        for( durchgang=0; durchgang<wiederholungen;
durchgang++ ) {
            for( i=0; i<laenge; i++ ) feld[i] = rand();
            meinSort( feld, laenge );
        }
        finish = clock();
        elapsed_time =
(double)(finish - start) / CLOCKS_PER_SEC;
        printf("laenge = \t%d\t", laenge);
        printf( "Sortieren brauchte \t%6.2f \tSekunden.\n",
elapsed_time );
    }
}
```

Die benötigte Zeit wird mit Hilfe der Funktion `clock()` gemessen. Diese Funktion liefert die Prozessorzeit zurück. Im Programm wird die Zeit vor und nach dem Sortieren gespeichert und die Differenz dann in Sekunden ausgegeben. Alle Messungen wurden auf einem PC mit einem 1,7 GHz Pentium Prozessor und 1 GByte RAM unter Windows XP durchgeführt. Zur Programmentwicklung wurde Microsoft Visual C++ .NET verwendet. Die Programme liefen in der Konfiguration *Release*.

16.3 Selection Sort

Eine sehr anschauliche Art des Sortierens ist die Auswahl (Selection) des jeweils kleinsten Elementes und anschließendes „Nach Hinten Stellen“. Das Verfahren besteht aus drei Schritten:

- suche das kleinstes Element
- vertausche das kleinste Element mit dem letzten Element
- verkleinere den Suchbereich

Das Verfahren bringt einen nach dem anderen Wert in die richtige Reihenfolge. Es ist daher auch gut geeignet, wenn man nur die M kleinsten oder – entsprechend modifiziert – größten Werte sucht. Bei der Suche nach dem jeweils kleinsten Element kann man wie folgt vorgehen:

- Nehme das erste Element (`feld[min=0]`) als kleinstes an. Die Variable `min` dient als Merker für den bisher kleinsten Wert.
- Prüfe für alle weiteren Elemente, ob sie kleiner sind.
- Falls ein kleineres Element gefunden wird, den Merker `min` aktualisieren.

Damit benötigt man für N Elemente $N-1$ Vergleichsoperationen. Für das Sortierverfahren sucht man im ersten Durchlauf unter N Elementen, im zweiten $N-1$, u.s.w.. Für die Anzahl der benötigten Vergleichsoperationen gilt:

1. Durchgang	$N-1$
2. Durchgang	$N-2$
...	
k. Durchgang	$N-k$

und für die Gesamtzahl G der Vergleichsoperationen

$$G = 1 + 2 + \ldots (N-1) = 1/2 \cdot N(N-1) = 1/2N^2 - 1/2N \tag{16.2}$$

Weiterhin muss man in jedem Durchgang zwei Werte tauschen. Nach dem Tausch steht das Element bereits an seiner endgültigen Stelle. Jedes Element muss höchstens einmal getauscht werden. Damit ist das Verfahren besonders geeignet, wenn man sehr große Daten hat, bei denen der Tausch aufwändig wird. Abhängig von dem Ausgang der Vergleichsoperation muss weiterhin der Merker für den bisher kleinsten Werte aktualisiert werden. Wie oft dies notwendig wird, hängt von den Daten ab. Es gibt zwei Extreme:

- das Feld ist optimal sortiert, d. h. der kleinste Wert steht jeweils an erster Stelle und keine Aktualisierung ist notwendig

- das Feld ist genau falsch sortiert und nach jedem Vergleich ist eine Aktualisierung erforderlich

An dieser Stelle wirkt sich die Anfangsordnung auf die Komplexität aus. Allerdings ist diese Aktualisierung des Merkers eine relativ unaufwändige Operation. Im wesentlichen wird der Rechenbedarf durch die Vergleichsoperationen bestimmt.

```
void tausche( int feld[], int i, int j ) {
    int zwi;
    zwi = feld[i];
    feld[i] = feld[j];
    feld[j] = zwi;
}

void selectionSort( int feld[], int laenge ) {
    int i, j, min;

    for( i=laenge; i>1; i-- ) {
        min = 0;
        for( j=1; j<i; j++ ) {
            if( feld[j] < feld[min] ) min = j;
        }
        tausche( feld, min, i-1);
    }
}
```

Beispiel 16.1. Selection Sort mit N=10 Elementen:

41	1846	633	2650	1916	1572	1147	2935	2696	2446
2446	1846	633	2650	1916	1572	1147	2935	2696	**41**
2446	1846	2696	2650	1916	1572	1147	2935	**633**	**41**
2446	1846	2696	2650	1916	1572	2935	**1147**	**633**	**41**
2446	1846	2696	2650	1916	2935	**1572**	**1147**	**633**	**41**
2446	2935	2696	2650	1916	**1846**	**1572**	**1147**	**633**	**41**
2446	2935	2696	2650	**1916**	**1846**	**1572**	**1147**	**633**	**41**
2650	2935	2696	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	**41**
2696	2935	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	**41**
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	**41**
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	**41**

Zusammengefasst gilt für den Selection Sort:

- Komplexität mit N^2

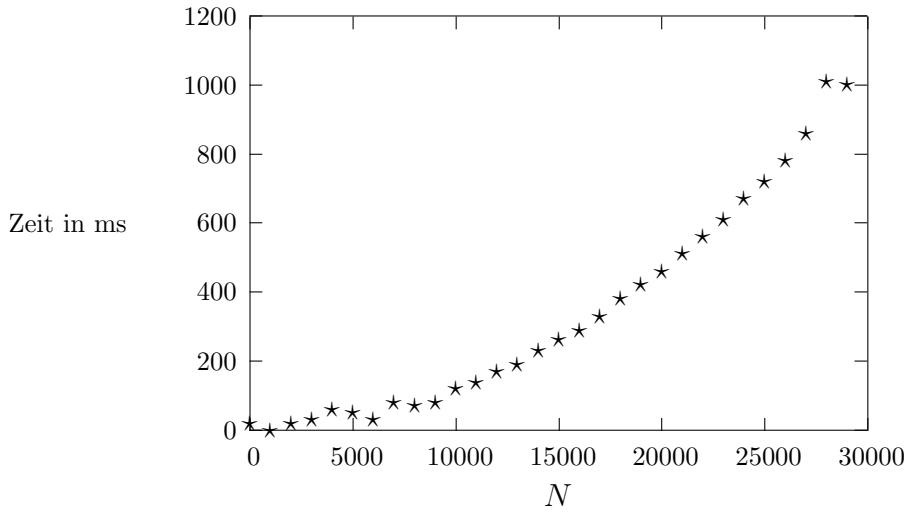


Abbildung 16.1: Gemessene Rechenzeit für Selection Sort mit wachsender Feldgröße N.

- relativ geringe Abhängigkeit der Rechenzeit von Werten
- einfache Implementierung
- wenige Tauschoperationen

16.4 Insertion Sort

Bei dem Selection Sort sucht man stets das verbleibende kleinste (größte) Element im unsortierten Bereich. Eine andere Methode ist, jeweils nur ein neues Element zu nehmen und in einen bereits sortierten Bereich einzufügen. Nach dieser Methode verfahren z. B. Kartenspieler, wenn sie eine Karte nach der anderen aufnehmen und einsortieren. Für das Verfahren gilt:

- setze ersten Wert
- ab zweitem Wert:
- finde Position im bereits sortierten Bereich (Zielposition)
- verschiebe alle Werte hinter der Zielposition
- füge Wert an Zielposition ein

Der Aufwand hängt jetzt stärker von der Werten ab. Betrachten wir das Element W_M an der Position M. Bisher sind bereits $M-1$ Werte sortiert. Um die richtige Position zu finden, muss W_M nicht mit allen bisherigen Werten verglichen werden. Es reicht vielmehr aus – beginnend mit dem ersten Wert – den ersten Wert zu suchen, der kleiner ist als W_M. Im Mittel - bei zufälliger Verteilung - benötigt man also nur $M/2$ Vergleiche. Damit ergibt sich für die mittlere Anzahl der benötigten Vergleiche

1. Durchgang	0
2. Durchgang	1
3. Durchgang	1,5
...	
k. Durchgang	$k/2$

bis $N/2$. Der Vergleich mit der entsprechende Form für den Selection Sort zeigt, dass der Insertion Sort nur etwa halb so viele Vergleichsoperationen benötigt. Nachteil ist, dass man im bereits sortierten Bereich ständig wieder Platz schaffen muss, um das jeweils nächste Elemente einzusortieren. Dies erfordert, dass alle nachfolgenden Elemente um eine Position verschoben werden. Wenn man im Durchschnitt jedes neue Element in der Mitte einfügt, werden wiederum $M/2$ Verschiebeoperationen benötigt.

Methode insertionSort:

```
/* Hilfsfunktion position
 * suche Position von wert in einem sortierten Feld
 */
int position( int wert, int feld[], int laenge ) {
    int i;

    for( i=0; i<laenge; i++ ) {
        if( feld[i] < wert ) return i;
    }
    return laenge;
}

void insertionSort( int feld[], int laenge ) {
    int i, p, k, zwi;

    for( i=1; i<laenge; i++ ) {
        p = position( feld[i], feld, i );
        zwi = feld[i];
        for( k=i-1; k>=p; k-- ) tausche( feld, k, k+1);
        feld[p] = zwi;
    }
}
```

Ein Hauptaufwand liegt im Vertauschen von Elementen, d. h. dem häufigen Aufruf der entsprechenden Funktion. Im Verhältnis zu der elementaren Verschiebeoperation ist der Aufwand für den Methodenaufruf recht groß. Außerdem wird die volle Funktionalität „Vertauschen“ nicht benötigt, sondern es geht nur darum, einen Wert nach hinten zu schieben. Daher lohnt es sich, in diesem Fall die Verschiebeaktionen direkt zu programmieren[1]:

```
for( k=i-1; k>=p; k-- ) feld[k+1] = feld[k];
```

Beispiel 16.2. mit N=10 Elementen:

41	1846	633	2650	1916	1572	1147	2935	2696	2446
1846	**41**	633	2650	1916	1572	1147	2935	2696	2446
1846	**633**	**41**	2650	1916	1572	1147	2935	2696	2446
2650	**1846**	**633**	**41**	1916	1572	1147	2935	2696	2446
2650	**1916**	**1846**	**633**	**41**	1572	1147	2935	2696	2446
2650	**1916**	**1846**	**1572**	**633**	**41**	1147	2935	2696	2446
2650	**1916**	**1846**	**1572**	**1147**	**633**	**41**	2935	2696	2446
2935	**2650**	**1916**	**1846**	**1572**	**1147**	**633**	**41**	2696	2446
2935	**2696**	**2650**	**1916**	**1846**	**1572**	**1147**	**633**	**41**	2446
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	**41**

Das Sortieren verläuft gerade entgegengesetzt zu Selection Sort. Die Elemente im linken Bereich sind bereits in der richtigen Rangordnung, aber die einzelnen Werte sind noch nicht unbedingt an ihrer endgültigen Position. Abbildung 16.2 zeigt den Rechenaufwand mit dem Insertion Sort.

Zusammenfassung für Insertion Sort:

- Komplexität mit N^2
- Abhängigkeit der Rechenzeit von Werten
- einfache Implementierung
- viele Tauschoperationen

16.5 Teilweise sortierte Felder

In vielen praktischen Fällen muss nicht stets ein komplettes Feld sortiert werden, sondern man kann davon ausgehen, dass bereits ein Teil des Feldes in der gewünschten Reihenfolge vorliegt. Beispielsweise kommen zu einer CD-Sammlung immer wieder neue Titel hinzu. Diese neuen Titel werden dann in die bestehende

[1] Ein guter Compiler wird allerdings bei voller Optimierung dies selbständig vornehmen.

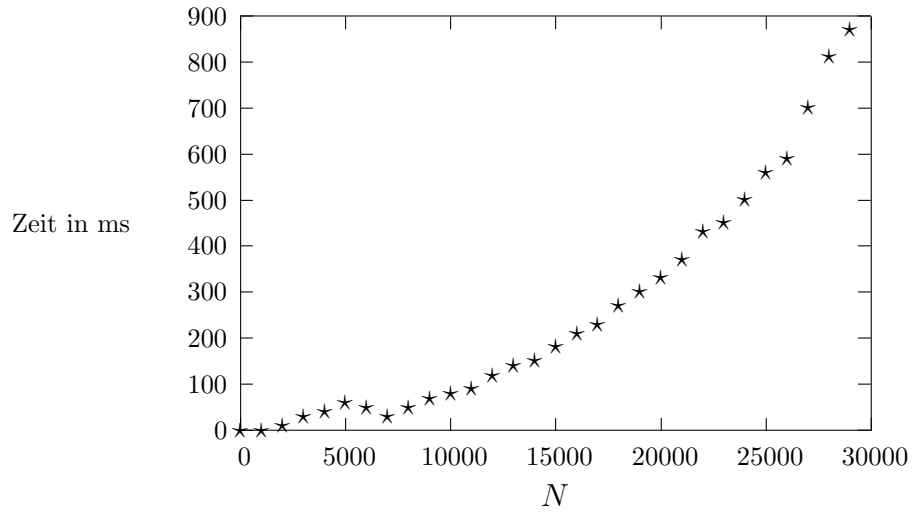


Abbildung 16.2: Gemessene Rechenzeit für Insertion Sort mit wachsender Feldgröße N.

Reihenfolge einsortiert. Man kann dies bei entsprechenden Anwendungen berücksichtigen und nach einem ersten vollständigen Sortiervorgang nur noch spezielle Funktionen zum Einsortieren aufrufen. Interessant ist allerdings auch die Frage, ob die Verfahren selbst bereits solche teilweise Ordnung ausnutzen können.

Bei dem Verfahren Selection Sort steckt der Hauptaufwand in der Maximumsuche. An dieser Stelle ist es kaum möglich, ohne spezielle Annahmen eine Aufwandsreduktion zu erreichen. Demgegenüber profitiert der Insertion Sort unmittelbar von der vorhandenen Ordnung. Wenn die ersten M Werte sortiert sind, entspricht dies genau einer Zwischenstufe im Sortierverfahren. D. h. in den ersten M Schritten kann sehr schnell festgestellt werden, dass das jeweils nächste Element an seiner - vorläufig - richtigen Position steht und damit keine Vertauschungen notwendig sind. Nur die $N-M$ neuen Elemente müssen dann einsortiert werden.

Zum Testen dieses Verhaltens wurde das Rahmenprogramm erweitert, so dass jetzt optional (Wert der Variablen `neu`) nicht mehr das komplette Feld, sondern nur noch die neuen Werte mit Zufallszahlen besetzt werden:

```
for( laenge=anfang; laenge<=end; laenge += ink ){
if( neu != 0 && laenge > anfang ) {
       zufall_anfang = laenge - ink;
} else {
zufall_anfang = 0;
}
for( i=zufall_anfang; i<laenge; i++ ) feld[i] = rand();
```

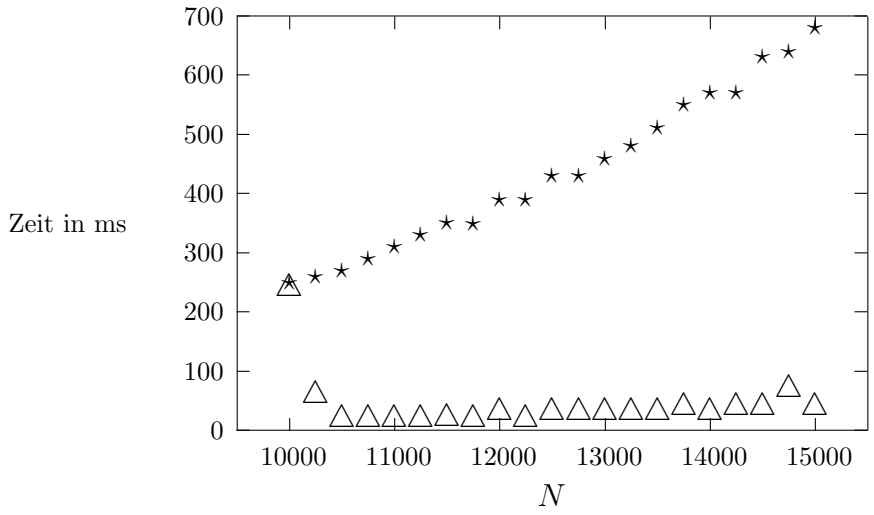


Abbildung 16.3: Rechenzeit zum Sortieren von nur teilweise zufällig besetzten Feldern (Selection Sort:⋆ Insertion Sort: △). Ausgehend von einem Feld der Größe 10000 wurden nach jedem Sortiervorgang jeweils 250 neue, unsortierte Zahlen angefügt.

. . .

Die entsprechenden Resultate sind in Bild 16.3 dargestellt. Die Ergebnisse bestätigen, dass der Insertion Sort die vorhandene Ordnung gut ausnutzen kann und dadurch deutlich schneller zum Ziel kommt als der Selection Sort. Für die gewählte Realisierung des Selection Sorts ist die absteigende Reihenfolge sogar besonders ungünstig, da dadurch in jedem Vergleichsschritt ein neues Minimum gefunden wird. Der damit verbundene Aufwand zum Umspeichern des Merkers führt zu einer längeren Laufzeit als bei den vollständig zufälligen Ausgangswerten.

16.6 Binäre Suche

Bei dem Insertion Sort sucht man die Position für einen Wert in einem bereits sortierten Feld. Bisher hatte wir nur eine allgemeine Suchfunktion benutzt, die sequentiell einen nach dem anderen Wert vergleicht, bis die passende Position gefunden ist. Bei M zufälligen Werten vom Max bis Min muss man im Mittel $M/2$ Vergleiche durchführen. Durch das Vergleichen Wert für Wert bewegt man sich nur sehr langsam durch das Suchfeld.

Es liegt nahe, effizienter Verfahren für diese spezielle Suche zu benutzen. Ein Ansatz ist, den Suchbereich schneller einzuengen. Eine Methode dazu ist die Binäre Suche. Nach dem Prinzip „Teile und Herrsche“ wird dabei in jedem Schritt

die Anzahl der Vergleichswerte halbiert, indem man X mit dem Wert in der Mitte des Suchbereichs vergleicht. Je nachdem ob X kleiner oder größer als der Wert in der Mitte ist, braucht man nur noch die linke oder rechte Hälfte des Feldes zu berücksichtigen. Dieses Verfahren setzt man fort, indem man im nächsten Schritt wieder mit dem mittleren Wert im verbleibenden Intervall vergleicht. Damit erreicht man in jedem Schritt eine Halbierung der Anzahl von Vergleichswerte.

Insgesamt ist dadurch die Anzahl der benötigten Vergleichsoperationen von der Ordnung $\log_2 M$ gegenüber $M/2$ bei der oben beschriebenen linearen Suche. Insbesondere bei großen M erreicht damit eine deutliche Reduktion. Beispielsweise reduziert sich bei $M = 30000$ die mittlere Anzahl der benötigten Vergleichsoperationen von 15000 auf $\log_2 30000 \approx 15$. Eine Realisierung der entsprechenden Suche ist:

```
int positionBinarySearch(
int wert, int feld[], int laenge ) {
    int m, u=0, l=laenge-1;

    if( wert < feld[laenge - 1 ] ) return laenge;

    while( l-u > 1 ) {
        m = (u + l ) / 2; /* Mitte bestimmen */
        if( wert == feld[m] ) return m;
        else if( wert < feld[m] ) u = m;
        else l = m;
    }
    if( wert > feld[u] ) return u;
    else return l;
}
```

Die beiden Variablen `u` und `l` enthalten die jeweils aktuellen Grenzen des Suchintervalls. Das Intervall wird solange in der Mitte geteilt, bis die Position gefunden ist. Das Laufzeitverhalten dieser Optimierung gegenüber der ersten Version des Insertion Sort zeigt Bild 16.4. Die Kurve verläuft deutlich flacher. Allerdings verbleibt nach wie vor der große Aufwand des Verschiebens beim Einsortieren.

16.7 Bubble Sort

Ein populäres Suchverfahren beruht auf dem Vertauschen benachbarter Werte. Dabei geht man durch die ganze Liste und wenn ein Paar nicht in der richtigen Reihenfolge ist, werden die Werte getauscht. Angenommen man beginnt bei dem letzten Paar, dann wird im ersten Durchgang der größte Wert immer wieder mit seinem Vorgänger getauscht, bis er am Ende des Durchgangs an der ersten Stelle steht. Bildlich kann man sich vorstellen, dass der größte Wert wie eine Gasblase

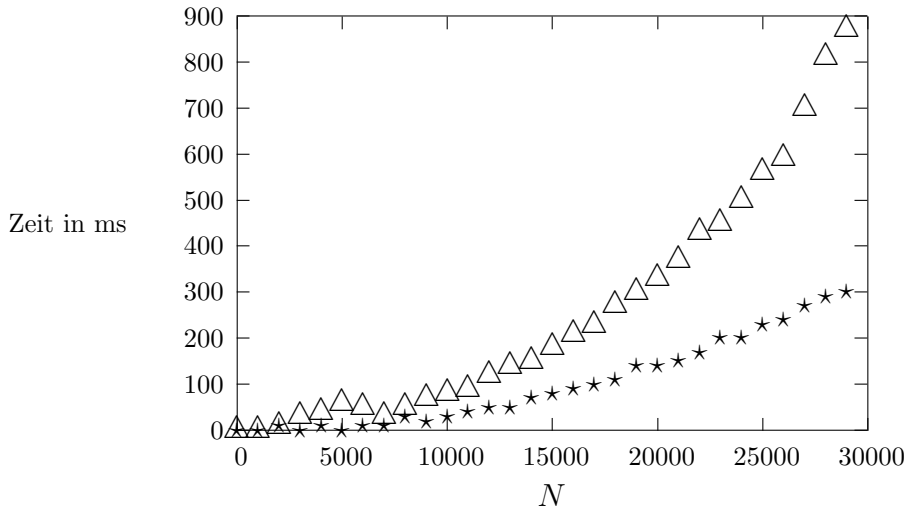


Abbildung 16.4: Insertion Sort mit binärer Suche ($\star$) und mit linearer Suche ($\triangle$).

(engl. bubble) in einem Glas mit Wasser langsam bis zur Oberfläche aufsteigt.

Das Verfahren hat große Ähnlichkeit mit dem Selection Sort. Allerdings wird die Suche nach dem verbleibenden größten Wert durch das fortgesetzte Vertauschen ersetzt. Wie bei dem Selection Sort erreicht das größte Element im jeweiligen Durchgang seine endgültige Position und braucht in den weiteren Durchgängen nicht mehr berücksichtigt zu werden. Die Anzahl der Vergleiche ist daher wie beim Selection Sort $N^2/2$. Bei zufälligen Eingangswerten legt jedes Element im Mittel den halben Weg zurück. Für die Anzahl der Vertauschoperationen ergibt sich daher

$$N * N/2 = N^2/2$$

In Summe benötigt der Bubble Sort daher genau so viele Vergleiche wie der Selection Sort, aber sehr viel mehr Tauschoperationen. Der Bubble Sort ist damit deutlich aufwändiger und sollte daher nicht eingesetzt werden. Er bietet zwar ein schönes Konzept und einige interessante theoretische Eigenschaften, aber keine praktischen Vorteile.

Im folgenden ist eine Implementierung angegeben. Die benötigte Rechenzeit ist in Bild 16.5 dargestellt. Man erkennt einen um etwa den Faktor 3 größeren Aufwand als beim Insertion Sort.

```
void bubbleSort( int feld[], int laenge ) {
    int i, j;
    int zwi;

    for( j=1; j<laenge; j++ ) {
```

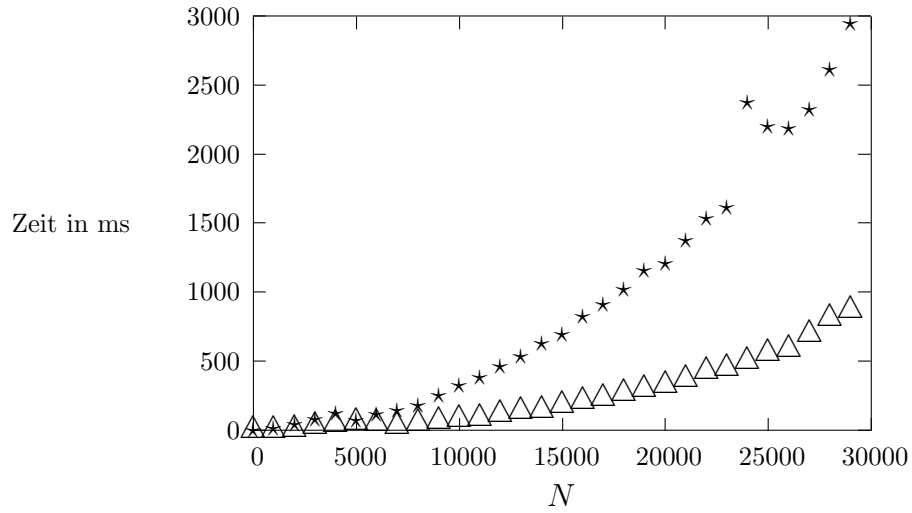


Abbildung 16.5: Rechenzeit mit Bubble Sort ($\star$) im Vergleich zu Insertion Sort ($\triangle$).

```
        for( i=laenge-1;i>=j; i-- ) {
            if( feld[i] > feld[i-1] ) {
                zwi = feld[i];
                feld[i] = feld[i-1];
                feld[i-1] = zwi;
            }
        }
    }
}
```

Beispiel 16.3. Bubble Sort mit N=10 Elementen:

41	1846	633	2650	1916	1572	1147	2935	2696	2446
2935	41	1846	633	2650	1916	1572	1147	2696	2446
2935	**2696**	41	1846	633	2650	1916	1572	1147	2446
2935	**2696**	**2650**	41	1846	633	2446	1916	1572	1147
2935	**2696**	**2650**	**2446**	41	1846	633	1916	1572	1147
2935	**2696**	**2650**	**2446**	**1916**	41	1846	633	1572	1147
2935	**2696**	**2650**	**2446**	**1916**	**1846**	41	1572	633	1147
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	41	1147	633
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	41	633
2935	**2696**	**2650**	**2446**	**1916**	**1846**	**1572**	**1147**	**633**	41

16.8 Shell Sort

Ein wesentlicher Nachteil im Bubble Sort ist die langsame Bewegung der Werte. Bei jeder Tauschaktion kann ein Wert nur um eine Position bewegt werden. Das gleiche Problem stellt sich beim Insertion Sort, wenn die bereits sortierten Werte verschobenen werden müssen, um für einen weiteren Wert Platz zu schaffen. Eine wesentliche Verbesserung wäre erreicht, wenn mit einem Tausch große Distanzen überbrückt werden könnten. Einen Ansatz dazu bietet der *Shell Sort*, erstmals vorgeschlagen von Donald L. Shell in 1959.

Die grundsätzliche Idee ist, zunächst mit Vergleichen zwischen weit entfernt liegenden Elementen die einzelnen Elemente „schnell“ in die Nähe der endgültigen Position zu bringen. Dabei werden zunächst Folgen von Werten mit festen Distanzen untereinander sortiert (z.B. bei Distanz 50 Element 1, 51, 101, ...). Die Distanz für den Vergleich wird dann nach und nach reduziert bis zum Schluss - bei der Distanz 1 - ein normaler Insertion Sort abläuft. Allerdings sind zu diesem Zeitpunkt die Elemente schon dicht bei ihren Zielpositionen, so dass der Insertion Sort sehr schnell abläuft.

Angenommen man beginnt mit einer Distanz von 10. Dann werden zunächst die Elemente 1, 11, 21, ... sortiert, dann die Elemente 2, 12, 22, .. u.s.w. bis 10, 20, 30, ... Danach sind 10 parallel sortierte Reihen entstanden. Im nächsten Schritt wird die Distanz verringert, z.B. auf 5, und wieder werden die einzelnen Reihen sortiert. Es gibt keine allgemein gültige Methode, die optimale Folge von Distanzen zu bestimmen. Für verschieden große Sortierfelder unterscheiden sich die optimalen Distanzfolgen. Gute Erfahrungen hat man mit auch der Folge

$$\ldots, 1093, 364, 121, 40, 13, 4, 1$$

gemacht. Die Werte sind leicht mit der Rekursion

$$d_i = 3 * d_{i-1} + 1 \tag{16.3}$$

zu berechnen. Im Einzelfall gibt es Folgen, die zu einer schnelleren Lösung führen. Allerdings ist der Unterschied nicht sehr groß. Andererseits gibt es keine Fälle, bei denen diese Folge ausgesprochen schlecht wäre. Generell ist der Algorithmus Shell Sort sehr schwer zu analysieren. So ist etwa die exakte Abhängigkeit der Laufzeit von der Feldgröße nicht bekannt. Vermutungen dafür sind $N(\log N)^2$ und $N^{1,25}$. Die unter vorgestellte Implementierung des Shell Sorts basiert auf einer effizienten Realisierung des Insertion Sorts. Dabei ist die Suche nach der richtigen Position und das „Platz schaffen“ in einer einzigen Schleife kombiniert. Für jede der zunehmend kleiner werdenden Distanzen wird ein eigener Insertion Sort durchgeführt.

```
/* ****************************************************
 * shell Sort nach R. Sedgewick: Algorithmen in C++ */
void shellSort( int feld[], int laenge ) {
```

```
int i, j, ink, v;

for( ink=1; ink<=laenge/9; ink=3*ink+1);
for( ; ink>0; ink /= 3 ) {
    for( i=ink+1; i<=laenge; i++ ) {
        v = feld[i-1];
        j = i-1;
        /* Verschieben (Platz machen)*/
        while( j>=ink && feld[j-ink]<v ) {
            feld[j] = feld[j-ink];
            j -= ink;
        }
        feld[j] = v; /* Einfuegen */
    }
}
```

Beispiel 16.4. Shell Sort mit N=10 Elementen:

	41	1846	633	2650	1916	1572	1147	2935	2696	2446
ink=4	2696				1916				41	
		2446				1846				1572
			1147				633			
				2935				2650		
ink=1	2696	2446	1147	2935	1916	1846	633	2650	41	1572
	2935	2696	2650	2446	1916	1846	1572	1147	633	41

Die Rechenzeit für Shell Sort ist in Bild 3 dargestellt. Aufgrund der hohen Geschwindigkeit wurden dabei wesentlich größere Felder (bis etwa 1.000.000 Werte) sortiert.

16.9 Quick Sort

Ziel bei der Optimierung von Sortierverfahren ist es, möglichst viel mit möglichst wenigen Operationen zu erreichen. So ist der Nachteil bei Bubble Sort, dass die vielen Vergleiche und Vertauschungen jeweils nur einen geringen Nutzen haben. Sie bringen ein Element auf dem Weg zu seiner Zielposition immer nur um eine Stelle weiter. Mit dem Shell Sort kann eine einzelne Vergleich- und Vertauschoperation ein Element wesentlich weiter bewegen.

Eine weitere, besonders effektive Art der Sortierung ist der Quick Sort (C.A.R. Hoare, 1960). Das Ziel ist, ein Element schnell an die Endposition zu bringen und gleichzeitig einen maximalen Nutzen aus den bei den dazu notwendigen Vergleichen zu ziehen. Der Ablauf ist wie folgt:

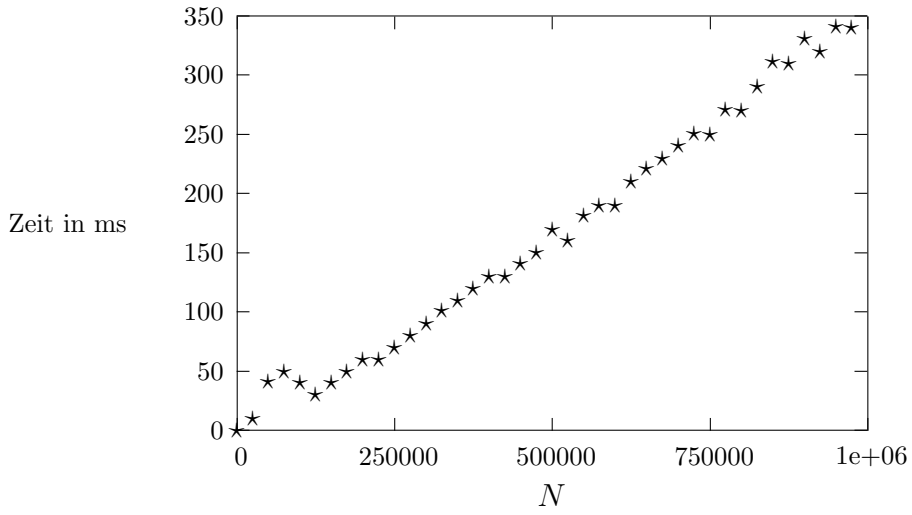


Abbildung 16.6: Rechenzeit mit Shell Sort.

1. Wähle ein beliebiges Element X aus („X soll in die Mitte kommen“).
2. Beginnend von links, suche das nächste Element K das kleiner als X ist.
3. Beginnend von rechts, suche das nächste Element G das größer als X ist.
4. K und G stehen auf den falschen Seiten, daher werden sie vertauscht.
5. Wiederhole Schritt 2 und folgende bis die Suche von links und rechts zusammen stößt.
6. Setze dann X an diese Stelle (durch Vertauschen).

Danach stehen alle Werte größer X auf der linken Seite und alle Werte kleiner X auf der rechten Seite von X. X selbst ist bereits an der richtigen Stelle. Bei der Auswahl des Vergleichswertes X ist man zunächst recht frei. Es liegt nahe, entweder den ersten oder letzten Wert zu nehmen. Man kann dann die Schleife einen Wert später beginnen oder einen Wert früher aufhören lassen.

Beispiel 16.5. Erster Durchgang im Quick Sort:

41	1846	633	2650	1916	1572	1147	2935	2696	2446^1
2696								41^2	
	2935						1846		
		2650	633^3						
			2446^4						633
2696	2935	2650	**2446**	1916	1572	1147	1846	41	633

1. Der letzte Wert – in diesem Fall 2446 – wird als Vergleichswert gewählt. Alle anderen Werte sollen auf die jeweils passende Seite gebracht werden. Damit beginnt die Suche nach Werten, die auf der falschen Seite stehen.

2. 2696 und 41 ist das erste Paar, das getauscht werden muss.

3. Als letztes Paar werden 2650 und 633 getauscht. Jetzt stehen alle Werte größer als 2446 links und alle kleineren Werte rechts von diesem Treffpunkt.

4. Der Vergleichswert wird an diese Position gestellt. Dies ist bereits die richtige Endposition.

In diesem Beispiel bleiben 3 Werte größer und 6 Werte kleiner als der betrachtete Wert. Die nächste Aufgabe ist, diese beiden Gruppen weiter zu sortieren. Dazu kann wiederum der Quick Sort mit den neuen Grenzen angewandt werden. Dieses Verfahren - auftrennen in zwei Teile und anschließend sortieren der Teile - wird immer wieder wiederholt, bis nur noch ein Wert in jedem Teil übrig bleibt.

Dies ist der grundlegende Algorithmus von Quick Sort. Man kann zeigen, dass er im Mittel $2N \ln N$ Vergleiche benötigt. Aufbauend auf dieser Grundidee gibt es eine Reihe von Modifikationen. Wichtig ist z. B. die Auswahl des Vergleichswertes. Wählt man stets den ersten oder letzten Wert, so erhält man bei bereits sortierten Dateien eine ungünstige Aufteilung zwischen den beiden Gruppen. Im Extremfall liegen dann alle Werte in einer Gruppe und die andere Gruppe ist leer. Besser ist es, einen zufälligen Wert oder etwa einen Wert aus der Mitte des Feldes auszuwählen.

Quick Sort ist ein schönes Beispiel für die Anwendung rekursiver Methodenaufrufe. In einem Schritt wird ein größeres Problem – Sortieren von N Werten – auf zwei kleiner aber gleichartige Problem – Sortieren von $\approx (N-1)/2$ Werten – zurück führen. Diese Probleme können weiter zerlegt werden, bis zum Schluss nur noch einzelne Werte übrig bleiben. Dann endet die Rekursion und die Aufgabe ist gelöst.

In den Simulationsmessungen wurde die Implementierung des Quick Sort Verfahrens nach Kernighan und Ritchie [KR88] benutzt. Bei dieser Variante ruft die Methode rekursiv sich selbst mit immer kleiner werdenden Feldgrößen auf. Die Realisierung benutzt nur einen Zeiger. Dabei werden alle Werte größer als X nacheinander in aufeinanderfolgenden Positionen an die linke Seite getauscht.

Beispiel 16.6. Erster Durchgang im modifizierten Quick Sort:

41	1846	633	2650	1916	1572	1147	2935	2696	2446
1916[1]				41					
	2650[2]		1846						
		2935[3]					633		
			2696					1846	
				2446[4]					41
2446				1916[5]					
2446	2650	2935	2696	**1916**	1572	1147	633	1846	41

1. Ein Wert X aus der Mitte wird gewählt – in diesem Fall 1916 – und mit dem ersten Element getauscht (Position null). Anschließend beginnt von links aus die Suche nach Werten, die größer als X sind.

2. 2650 ist der erste gefundene Wert. Er wird an die Position eins getauscht.

3. Der zweite gefundene Wert ist 2935. Er kommt an Position zwei.

4. Als letzter Wert wird 2446 nach vorne gebracht. Jetzt stehen alle Werte größer X=1916 ab Position eins.

5. Schließlich muss noch der Vergleichswert 1916 mit dem letzten größeren Wert getauscht werden. Im Ergebnis steht 1916 schon an der richtigen Stelle. Alle größeren Werte stehen links davon, alle kleineren rechts davon.

```
/* ******************************************************
 * quicksort, realisierung aus K&R
 */
void quickSort( int feld[], int laenge ) {
    quickSortR( feld, 0, laenge - 1 );
}
void quickSortR( int feld[], int links, int rechts ) {
    int i, letzte;

    if( links >= rechts ) return;
    tausche( feld, links, (links + rechts) / 2);
    letzte = links;
    for( i=links+1; i<=rechts; i++ ) {
        if( feld[i] > feld[links] ) {
 tausche( feld, ++letzte, i);
}
    }
    tausche( feld, links, letzte);
    quickSortR( feld, links, letzte-1);
    quickSortR( feld, letzte+1, rechts );
}
```

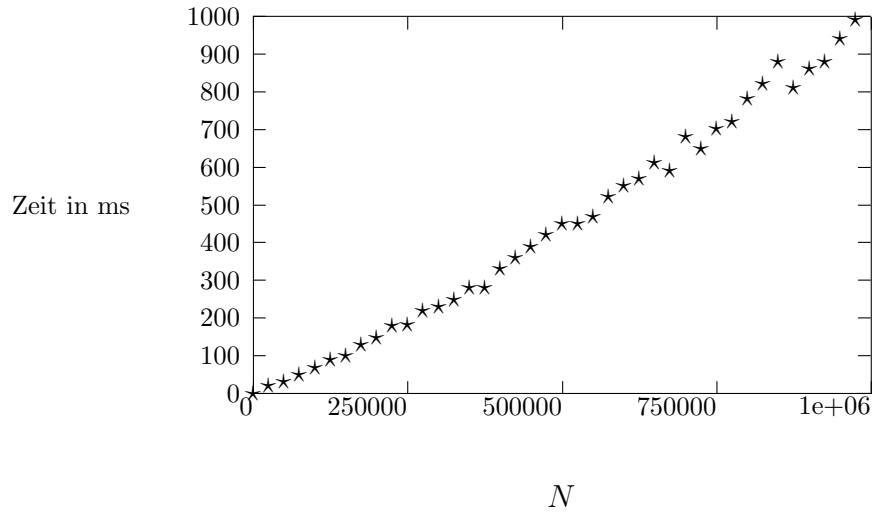


Abbildung 16.7: Rechenzeit mit QuickSort.

Anhang A

Aufgabenblätter

Das C Quiz

1. Wann wurde C entwickelt?

2. In welcher Firma?
   ◯ Apple ◯ AT&T ◯ IBM ◯ Microsoft

3. Woher kommt der Name C?

   ◯ vom Anfangsbuchstaben des Vornamens der Frau des Entwicklers

   ◯ von der Vorläufersprache B

   ◯ vom Anfangsbuchstaben von **C**alifornia

4. Welche Programmiersprachen haben wesentliche Elemente aus C übernommen?
   ◯ Basic ◯ Cobol ◯ Java ◯ JavaScript ◯ Lisp ◯ Pascal ◯ PHP

5. Welche Ausgabe wurde als Einstiegsbeispiel durch C berühmt?

   ◯ *To be or not to be*

   ◯ *It's my way*

   ◯ *Hello world*

6. Wie viele Ergebnisse liefert die Suche *Bücher : Fachbücher : Informatik : Praktische Informatik : Programmiersprachen & Compiler : C & C++* bei amazon?
   ◯ 135 ◯ 976 ◯ 1238 ◯ 2126 ◯ 4608

7. Unter `spectrum.ieee.org/static/interactive-the-top-programming-languages` findet man IEEE Spectrum's 2014 Ranking *The Top Programming Languages*. Auf welchem Platz erwarten Sie C? Falls Sie C nicht auf dem ersten Platz sehen – welche andere Programmiersprache erwarten Sie dort?

Integer Quiz

Die Variablen `i` und `j` seien vom Typ `long`. Welche Werte haben sie nach folgenden Anweisungen :

`i = 32 / 6 + 7 % 2;`	`i =`
`i = 7654; i = i / 100 * 10`	`i =`
`i = 6; j = 4; j = ++i * j;`	`i =  j =`
`i = 6; j = 4; j = i++ * j;`	`i =  j =`
`i = 2; j = 3; i *= j + 1;`	`i =`

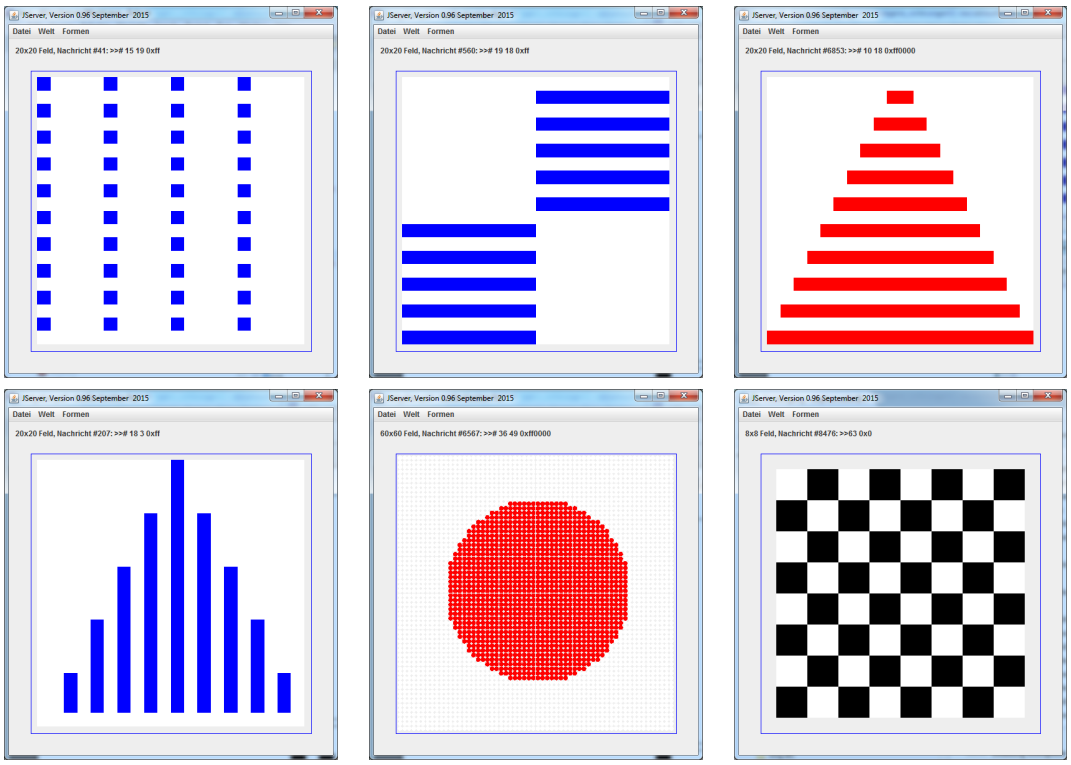
A.1 Einstieg in BoS

Übung A.1. Testat-Wertung: Buchstabe
Schreiben Sie Anweisungen für eine schöne Darstellung des Anfangsbuchstabens Ihres Namens. Bei sehr einfachen Buchstaben (*I*, *L*) sollten Sie mehr Aufwand in die Gestaltung legen, z. B. mit Serifen, Schatten oder Kursiv-Schrift.
Tipp: skizzieren Sie zuerst den Buchstaben auf Papier, am besten auf einem Ausdruck mit den nummerierten Feldern.
Abgabe:

- zeigen Sie Ihre Lösung in Ihrer Übungsgruppe
- speichern Sie die Bilddatei in der Moodle-Aufgabe. Die Dateien sollen dann gesammelt als Galerie online gestellt werden. Bitte lassen Sie daher den Datei Namen mit Ihrem Buchstaben beginnen (z. B. `E_ueb1.png`)

A.2 Abläufe

Übung A.2. Muster
Schreiben Sie Anweisungen, um folgende Muster zu erzeugen:



Testat: 5 Muster

A.3 Gleitkomma-Zahlen

Übung A.3. Nach einer alten Legende wünschte sich der Erfinder des Schachspiels vom König:

- 1 Reiskorn auf dem ersten Feld eines Schachbrettes
- 2 Reiskörner auf dem 2. Feld
- 4 Reiskörner auf dem 3. Feld
- u.s.w.

Geben Sie die Anzahl der Reiskörner mit wachsender Anzahl von Feldern aus. Berechnen Sie zusätzlich, wie viele LKWs (je 7,5 Tonnen Ladung) man für den Transport benötigt, wenn jedes Reiskorn 30 mg wiegt. Wenn weiterhin ein Reiskorn ein Volumen von etwa $3.5 \cdot 10^{-8} m^3$ hat, wie hoch wird dann ein Fußballfeld (70 auf 105 m) bedeckt?
Ausgabe mit `printf`, kein BoS.

Übung A.4. Muster
Lassen Sie die Funktion $sin(x) * cos(y)$ für Werte von x und y von 0 bis 2π darstellen.

- Rechnen Sie die Brettkoordinaten $0, 1, \ldots, N-1$ in entsprechende Werte aus dem Bereich $[0, 2\pi]$ um.
- Berechnen Sie dann für jedes Feld den entsprechenden Wert $sin(x) * cos(y)$.
- Negative Werte sollen rot, positive grün gefärbt werden.
- Die Funktion liefert Werte zwischen -1 und 1 zurück. Dementsprechend soll die Größe der Symbole gewählt werden. Beachten Sie, dass ein Radius von 0,5 das ganze Feld ausfüllt. Das Programm akzeptiert auch negative Werte.
- (freiwillig) Welche Muster ergeben sich bei anderen Funktionen wie $sin(x) * sin(y)$

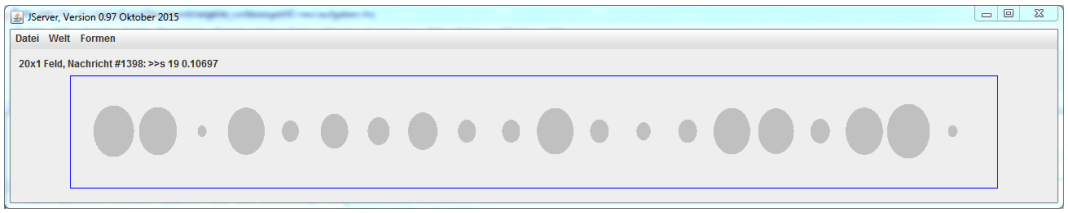
A.4 Zeichen und Felder

Übung A.5. Erweitern Sie das Beispiel zur Darstellung der ASCII-Zeichen, in dem Sie die drei Gruppen Ziffern, Klein- und Großbuchstaben farbig markieren. Verwenden Sie zur Prüfung die Funktionen `isdigit( c )`, ...

Übung A.6. Rechnen mit Feldern
Im folgenden Programm-Schnipsel wird ein Feld mit Zufallszahlen zwischen 0,1 und 0,5 gefüllt. Dazu passend werden Symbole auf die jeweilige Größe gesetzt.

```
#define anzahl 20

int i;
double feld[anzahl];

groesse( anzahl, 1);

// Initializiere Zufallszahlengenerator mit aktueller Uhrzeit
// sonst kommt immer die gleiche Zahlenfolge
srand ( time(NULL) );

// Feld mit Zufallszahlen fuellen
for( i=0; i<anzahl; i++ ) {
    feld[i] = 0.1 + 0.4* (float) rand()/RAND_MAX;
    symbolGroesse( i, feld[i] );
}
```

Das Ergebnis sollte wie folgt aussehen:



- Bestimmen Sie Minimum und Maximum des Feldes und markieren es farblich (z. B. Minimum Rot, Maximum Grün).
- Geben Sie zusätzlich Minimum, Maximum und Mittelwert des Feldes mit `printf` aus.

A.5 Zeichenketten und Felder

Übung A.7. Passwort-Generator
Ein Programm soll mit Hilfe der Funktion `rand()` zufällige Passworte generieren. Jedes Passwort ist 8 Zeichen lang. Realisieren Sie folgende Versionen von Passworten:

1. 8 Buchstaben (Beispiel `ahgtzjui`)
2. 3 Buchstaben 1 Ziffer 3 Buchstaben 1 Ziffer (Beispiel `hgt4klx1`)
3. 1 Konsonant 1 Vokal 1 Konsonant 1 Ziffer 1 Konsonant 1 Vokal 1 Konsonant 1 Ziffer (Beispiel `hag6pox0`)

`Hinweis`: Legen Sie char-Felder mit den jeweiligen Zeichen (Vokale, Konsonaten) an. Lassen Sie dann an den einzelnen Stellen immer zufällig ein Zeichen aus dem jeweiligen Feld aussuchen (Funktion rand()).

Übung A.8. Spielsimulation
In einer Freistunde treffen sich 6 Studenten. Aus Langeweile beginnen sie ein einfaches Spiel: Jeder Spieler erhält eine der Nummern 1 bis 6. Dann wird gewürfelt. Der Spieler mit der gewürfelten Nummer erhält einen Punkt. Dann wird immer wieder gewürfelt und die erzielten Punkte pro Spieler werden hochgezählt. Das Spiel ist zu Ende, sobald der erste Spieler 10 Punkte erreicht hat.

Um abzuschätzen, wie viele Spiele in einer Freistunde möglich sind, möchten die Studenten die mittlere Dauer eines Spiels wissen. Minimum sind 10 Würfe (immer die selbe Zahl) und Maximum 55 Würfe (alle 6 Spieler haben schon 9 Punkte, der nächste Wurf entscheidet).

- Schreiben Sie ein Programm, um ein solches Spiel zu simulieren. Der Punktestand soll in einem Feld gespeichert werden.
- Lassen Sie dann viele Spiel durchlaufen und ermitteln daraus die mittlere Spieldauer.

A.6 Funktionen

Übung A.9. Quersumme
Gesucht sind Funktionen zur Berechnung der Quersumme einer Zahl.

- Schreiben Sie eine Funktion mit iterativer Berechnung.
- Schreiben Sie eine Funktion mit rekursiver Berechnung.
- Testen Sie die beiden Funktionen. Rufen Sie dazu die Funktionen z. B. mit zufälligen Werten auf und prüfen, dass die Ergebnisse überein stimmen.

Übung A.10. Klausurnoten
In den *Allgemeine Bestimmungen für Bachelorprüfungsordnungen* ist in § 6 die Bewertung der Leistungen geregelt. Verwenden Sie die Formel

$$N = 4 - 3 * (P - 50)/45 \quad 95 \geq P \geq 50$$

zur Berechnung den Note N bei P erzielten Prozent. Werte außerhalb des angegebenen Bereichs sind einheitlich 1,0 beziehungsweise 5,0.

1. Implementieren Sie auf dieser Basis eine **Funktion**, um aus einer Prozentzahl die Note zu berechnen. Die Methode soll einen `int`-Wert mit den Prozenten als Zahl zwischen 0 und 100 als Parameter erhalten und die Note als `double`-Wert zurückgeben. Die Rundung auf eine Nachkommastelle ist nicht notwendig.

2. Testen Sie die Funktion, indem Sie in einer Schleife die Prozentzahlen von 0 bis 100 durchgehen und das Ergebnis der Funktion ausgeben.

3. Schreiben Sie nun eine zweite Funktion, die anstelle der Prozentzahl die erzielten Punkte und die Maximalzahl erhält. Diese Funktion soll dann daraus die Prozentzahl berechnen, damit die bereits vorhandene Funktion aufrufen und deren Ergebnis zurück geben.

4. Schließlich wird noch eine Funktion benötigt, die ein ganzes Feld von Prozentwerten erhält und die Noten in ein zweites Feld einträgt. Übergeben Sie dieser Funktion die beiden Felder und die Anzahl der Klausuren, ein Vorschlag dazu:

```
void vieleNoten( int p[], double n[], int anz )
```

Wichtig bei dieser Aufgabe ist, dass die Berechnung gemäß der Formel tatsächlich nur einmal im Code auftaucht. Dann lässt sich das Programm bei eventuellen Änderungen der PO leicht anpassen.

A.7 PQ und BoS

Übung A.11. pq-Formel
Schreiben Sie ein Programm, das die Lösungen der quadratischen Gleichung

$$x^2 + p \cdot x + q = 0$$

berechnet. Dabei soll q den festen Wert 0,1 haben und p in Schritten von 0,1 von 0 bis 2 laufen. Prüfen Sie jeweils, ob eine reelle Lösung existiert. Falls ja, berechnen Sie die beiden Lösungen und geben sie mit `printf` aus. Zur Kontrolle setzten Sie die gefundenen Werte in die Gleichung ein und geben das Ergebnis ebenfalls aus. Im Fall von komplexen Lösungen geben Sie einen entsprechenden Hinweis aus. `Hinweis`: zum Berechnen der Wurzel gibt es die Methode `sqrt()`;

Übung A.12. BoS
Ziel dieser Aufgabe ist es, die Darstellung durch BoS mit einem C-Programm zu verbinden. Folgen Sie den Beispielen im Skript (Abschnitt 8.3) und schreiben eine einfache Animation. Vorschläge dazu

- ein Punkt, der von links unten nach rechts oben wandert
- eine Zufallsbewegung ab einem Startpunkt (*random walk*), einmal besuchte Brettfelder bleiben eingefärbt
- eine Spiralbewegung ab dem Mittelpunkt
- ...

A.8 Zeiger und BoS

Übung A.13. Zeiger
Gesucht ist eine Funktion zur Addition zweier Brüche:

$$\frac{z1}{n1} + \frac{z2}{n2} = \frac{z}{n}$$

Diese Funktion erhält also vier Werte (zwei Zähler und zwei Nenner) und berechnet daraus zwei neue Werte (Zähler und Nenner der Summe). Mit `return` könnten wir nur einen Wert zurück geben. Daher übergeben wir der Funktion als Alternative zwei Zeiger auf Variablen, die das Ergebnis aufnehmen sollen.

- Schreiben Sie eine entsprechende Funktion mit insgesamt 6 Argumenten (4 `int`, 2 `int *`).
- Testen Sie diese Funktion an einigen einfachen Beispielen.
- Die Harmonische Reihe ist definiert als

  $$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

  Verwenden Sie Ihre Additions-Funktion um die ersten Werte zu berechnen. Lassen Sie jeweils Zähler und Nenner sowie den Wert als Dezimalzahl ausgeben. Ein Vorschlag:

  ```
  H1: 1 / 1 = 1.000000
  H2: 3 / 2 = 1.500000
  H3: 11 / 6 = 1.833333
  H4: 50 / 24 = 2.083333
  H5: 274 / 120 = 2.283333
  H6: 1764 / 720 = 2.450000
  ```

- (freiwillig) Beim Vergleich mit den Werten bei Wikipedia[1] fällt auf, dass die Brüche noch gekürzt werden können. Implementieren Sie eine entsprechende Funktion. Verwenden Sie dabei eine Funktion, die den größten gemeinsamen Teiler zweier Zahlen berechnet. Sie können dabei gerne auf eine der zahlreichen Lösungen im Internet zurückgreifen (aber mit Quellenangabe).

Übung A.14. X-BoS
Gestalten Sie einen schönen Weihnachtsbaum mit BoS (Sammlung in einer Galerie wie bei Anfangsbuchstaben ist geplant).

[1] `https://de.wikipedia.org/wiki/Harmonische_Reihe`

A.9 Bruch-Struktur und Buchstabenlotto

Übung A.15. Bruch strukturiert
Die Lösung von Aufgabe A.13 soll mit einer Struktur übersichtlicher werden:

1. definieren Sie eine Struktur für Brüche.
2. schreiben Sie dann eine Funktion zur Addition, die zwei Brüche erhält und den resultierenden Bruch zurück gibt.
3. verwenden Sie die neue Funktion zur Berechnung der Harmonische Reihe.
4. (freiwillig) bauen Sie die Vereinfachung der Brüche mittels Kürzen ein.

Tipp: mit

```
typedef struct bruch Bruch;
```

können Sie zur besseren Übersicht einen eigenen Typ `Bruch` definieren.

Übung A.16. Buchstabenlotto
Gegeben ist ein String mit den Buchstaben aus dem Wort *friedberg*

```
char buchstaben[] = "bdefgir";
```

1. Generieren Sie nach dem Verfahren in der Passwort-Aufgabe aus den Buchstaben zufällige Wörter der Länge 9. Wiederholen Sie dies so lange, bis irgendwann das Wort *friedberg* erzeugt wird. Wie viele Versuche werden benötigt?
2. Schreiben Sie ein Variante, bei der man das Zielwort frei vorgeben kann. Dabei sollen die vorkommenden Buchstaben automatisch herausgesucht werden.

A.10 Dateien

Übung A.17. Webseiten-Generator
Im Archiv `material.zip` finden Sie einige Bilddateien sowie die Datei `dateien.txt`. Daraus sollen Webseiten erzeugt werden.

1. Schreiben Sie ein C-Programm, das die Liste aus `dateien.txt` liest und daraus eine Webseite mit entsprechenden img-Elementen erzeugt und in einer html-Datei speichert (eine Datei mit allen Bildern). Sie können den HTML-Text aus einer anderen Datei lesen (flexiblere Lösung) oder fest im Programm eintragen.

2. Erweitern Sie das Programm so, dass für jedes Bild eine eigene HMTL-Seite angelegt wird.

Literaturverzeichnis

[Dij82] Edsger W. Dijkstra. Why numbering should start at zero. August 1982.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 1991.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, 1988.

[Rit93] Dennis M Ritchie. The Development of the C Language. 1993.

[Sed94] R. Sedgewick. *Algorithmen in C++*. Addison-Wesley, 1994.

Index