

Version 0.17

Programmieren in C



... und ein bisschen UNIX

Ulrich Schrader

Nur zur Verwendung in der Vorlesung - Nicht weitergeben

Inhaltsverzeichnis

1. UNIX — The Bare Necessities	9
1.1. Die wichtigsten Kommandos	9
1.1.1. Was sind Unix Dateien und Verzeichnisse?	9
1.1.2. Benennung von Dateien und Verzeichnissen	9
1.1.3. Eine neue Datei erstellen	10
1.1.4. Eine Datei kopieren	10
1.1.5. Eine Datei umbenennen	10
1.1.6. Eine Datei löschen	11
1.1.7. Ein Verzeichnis anlegen	11
1.1.8. Dateien in ein anderes Verzeichnis kopieren und verschieben	11
1.1.9. Ein Verzeichnis umbenennen	12
1.1.10. Ein Verzeichnis kopieren	12
1.1.11. Ein Verzeichnis löschen	12
1.1.12. In Verzeichnissen navigieren	12
1.2. Zusammenfassung der Befehle	13
1.2.1. Mit Dateien arbeiten	13
1.2.2. Mit Verzeichnissen arbeiten	13
1.2.3. Mit Dateien und Verzeichnissen arbeiten	13
2. Der Einstieg	15
2.1. Hello World!	16
2.1.1. Übersetzung mit GNU-C (gcc)	16
2.1.2. Hello World ... in kleinen Schritten	16
2.1.3. Hello World ... reloaded	18
2.1.4. ESCAPE-Sequenzen	19
2.2. Challenges	20
2.2.1. Ein Programm, um alle Zeilen von helloworld.c auszudrucken	20
2.2.2. Kalender	21
3. Variablen und grundlegende Datentypen	23
3.1. Eigenschaften von Variablen	23
3.2. Grundlegende Datentypen	23
3.3. Arbeiten mit ganzen Zahlen (int)	24
3.4. Arbeiten mit Fließkommazahlen (float)	24
3.5. Arbeiten mit Buchstaben (char)	25
3.6. Initialisierung von Variablen und der Zuweisungsoperator	25

3.7. Den Inhalt von Variablen ausgeben	26
3.7.1. Konversionsspezifikatoren	26
3.8. Einlesen mit scanf()	27
3.9. Arithmetik in C	28
3.10. Challenges	29
3.10.1. Umsatzberechnung	29
3.10.2. Umsatzberechnung und Gewinn	29
3.10.3. Body Mass Index	29
4. Bedingte Verzweigungen	31
4.1. if ... else	31
4.1.1. Logische Ausdrücke	31
4.2. Das switch-Statement	34
4.3. Nützliche Funktionen	35
4.3.1. isdigit() und isalpha() Funktion	35
4.3.2. Zufallszahlen - rand()	36
4.4. Challenges	37
4.4.1. Wahrheitstabelle	37
4.4.2. Groß- und Kleinbuchstaben	37
5. Schleifenstrukturen	39
5.1. Schleifen sind überall	39
5.2. Weitere Operatoren	40
5.3. while-Schleife	41
5.3.1. Fast-Endlosschleife	42
5.4. do ... while-Schleife	43
5.5. Die for-Schleife	44
5.6. Konzentrationsspiel	45
5.7. Challenges	47
5.7.1. Einfaches Zählen	47
5.7.2. Mathequiz	47
5.7.3. Konzentrationsspiel	47
6. Funktionen	49
6.1. Deklarieren einer Funktion	49
6.2. Definieren einer Funktion	50
6.3. Funktionsaufrufe	51
6.4. Sichtbarkeit von Variablen	53
6.4.1. Lokale Variablen	53
6.4.2. Globale Variablen	55
6.5. Challenges	56
6.5.1. Konzentrationsspiel 2.0	56
6.5.2. Geldwechselautomat	56

7. Arbeiten mit Arrays	57
7.1. Deklarieren von Arrays	57
7.1.1. Verwenden von Arrays	58
7.1.2. Initialisieren von Arrays	59
7.2. Mehrdimensionale Arrays	59
7.3. Beispielprogramm: Tic-Tac-Toe	61
7.4. Challenges	65
7.4.1. Finde den Fehler	65
7.4.2. Kumulative Summe	65
7.4.3. Zahlen sortieren	65
7.4.4. Funktion zur Berechnung der kumulativen Summe	66
7.4.5. Tic-Tac-Toe	66
8. Umgang mit Pointern	67
8.1. Deklarieren und Initialisieren von Pointer-Variablen	67
8.2. Funktionen und Pointer	69
8.3. Übergabe von Arrays an Funktionen	71
8.4. Die const Anweisung	72
8.5. Cryptogram	74
8.6. Challenges	75
8.6.1. Variablen und ihre Pointer	75
8.6.2. Würfelspiel	76
8.6.3. Cryptogramm	76
9. Arbeiten mit Strings	77
9.1. Strings – Einführung	77
9.2. Strings in Zahlen konvertieren	79
9.3. Stringverarbeitung	80
9.3.1. strlen() Funktion	80
9.3.2. tolower() und toupper() Funktionen	80
9.3.3. strcpy() Funktion	82
9.3.4. strcat() Funktion	82
9.4. Strings analysieren	83
9.4.1. strcmp() Funktion	83
9.4.2. strstr() Funktion	84
9.5. Beispielprogramm: Find Word	85
9.6. Challenges	86
9.6.1. In Großbuchstaben umwandeln (1)	86
9.6.2. In Großbuchstaben umwandeln (2)	86
9.6.3. Alphabetische Sortierung	87
10. Dynamische Speicherverwaltung	89
10.1. Grundlagen	89

10.2. malloc() Funktion	89
10.2.1. NULL-Pointer	90
10.2.2. Bestimmen des Speicherbedarfs	90
10.2.3. Speicher freigeben	91
10.3. calloc() und realloc() Funktionen	93
10.4. Challenges	95
10.4.1. Lieblingsfilm	95
10.4.2. Länge des Namens	95
10.4.3. String aneinanderfügen	95
11. Umgang mit Dateien	97
11.1. Einführung	97
11.2. Binärdateien und Textdateien	97
11.3. Dateipuffer	98
11.4. Grundlegende Dateifunktionen	98
11.4.1. Öffnen einer Datei mit fopen()	98
11.4.2. Schließen einer Datei mit fclose()	100
11.4.3. fgets()	101
11.4.4. fputs()	102
11.4.5. fprintf()	102
11.4.6. fscanf()	104
11.4.7. fseek()	105
11.5. Challenges	106
11.5.1. Anlegen einer Datei	106
11.5.2. Auslesen einer Datei	106
11.5.3. Anfügen an eine Datei	106
12. Komplexe Datenstrukturen (struct)	107
12.1. Die struct Anweisung	107
12.2. Übergabe von Strukturen an Funktionen	108
12.2.1. Übergabe von Strukturen an Funktionen - passing by value	108
12.2.2. Übergabe von Strukturen an Funktionen - passing by reference	109
12.3. Anwendungsbeispiel: Verkettete Listen	110
12.4. Challenges	113
12.4.1. Studierendendaten	113
12.4.2. Sortierte, verkettete Listen	113
12.4.3. Löschen aus verketteten Listen	113
13. Debugging und Umgang mit Fehlern	115
13.1. Debugging	115
13.1.1. Debugging mit printf()	115
13.1.2. Debugging mit assert.h	120
13.2. Umgang mit Fehlern	122
13.2.1. errno, perror() und strerror()	122

14. Rekursion	125
14.1. Einleitung	125
14.2. Beispiele für Rekursionen	125
14.2.1. Fakultät	125
14.2.2. Türme von Hanoi	126
14.3. Verkettete Listen reloaded	128
14.4. Challenges	134
14.4.1. Löschen in einer verketteten Liste	134
14.4.2. Löschen einer verketteten Liste	134
14.4.3. Fibonacci-Folge	134
A. Häufig verwendete Bibliotheksfunktionen	135
B. ASCII Tabelle - druckbare Zeichen	139

1. UNIX — The Bare Necessities

UNIX ist ein komplettes, umfassendes Betriebssystem mit einer großen Anzahl von Möglichkeiten. Hier ist das wichtigste zusammengefasst. Für alles weitere gibt es Dr. Google.

1.1. Die wichtigsten Kommandos

Dieses Kapitel führt die Kommandos zum Erstellen (create), Kopieren (copy), Umbenennen (rename) und Entfernen (remove) von UNIX Dateien (files) und Verzeichnissen (directories) auf. Dabei wird hier vorausgesetzt, dass dieses in einer UNIX-Umgebung erfolgt.

1.1.1. Was sind Unix Dateien und Verzeichnisse?

Eine Datei ist eigentlich nichts anderes als ein Behälter für Daten. Dabei wird kein Unterschied zwischen Dateitypen gemacht. Eine Datei kann beispielsweise den Text eines Dokuments enthalten, die Eingabedaten für ein Programm oder die von dem Programm erzeugten Ausgabedaten enthalten, oder aber das Programm selber sein.

Verzeichnisse dagegen bieten die Möglichkeit Dateien zu organisieren. In Verzeichnissen kann man etwa zusammengehörige Dateien gruppieren, etwa um sie schnell und leicht wiederzufinden. Verzeichnisse können dabei Dateien und/oder weitere Verzeichnisse enthalten. Unter Windows werden diese oft als Ordner bezeichnet.

1.1.2. Benennung von Dateien und Verzeichnissen

Jede Datei und jedes Verzeichnis hat einen Namen. Dieser muss innerhalb eines Verzeichnisses eindeutig sein. Allerdings dürfen Dateien und Verzeichnisse sehr wohl denselben Namen führen, sofern sie in unterschiedlichen Verzeichnissen enthalten sind.

Eine Datei- oder ein Verzeichnisname darf bis zu 256 Zeichen lang sein. Die Namen dürfen beinahe alle Zeichen enthalten außer dem Leerzeichen. Oft werden die Worte in mehrwortige Dateinamen durch einen Unterstrich (Underscore), einen Punkt (period), oder durch die sogenannte Camelhump-Notation, bei der jedes Wort mit einem Großbuchstaben beginnt, getrennt. So sind `hello_world.c`, `hello.world.c` oder auch `HelloWorld.c` jeweils unterschiedliche Dateinamen.

Einige Buchstaben haben besondere Bedeutung in Unix. Daher sollten `/\"'?!*?~$<>` in Dateinamen vermieden werden:

Unix unterscheidet zwischen Groß- und Kleinschreibung (case-sensitive). Die folgenden Dateinamen sind daher unterschiedlich: `myfile`, `Myfile`, `myFile`, and `MYFILE`.

1.1.3. Eine neue Datei erstellen

Die einfachste Möglichkeit eine Datei zu erstellen die Verwendung eines Texteditors wie `vi(m)`, `nano` oder `kate`.

Es kann aber auch das Kommando `cat` verwendet werden, um einfache Dateien ohne einen Texteditor zu Übungszwecken zu erstellen. Um etwa eine Übungsdatei namens `firstfile` mit einer Zeile Text zu erstellen, gebe bei der Eingabeaufforderung (prompt) in der Shell das Kommando:

```
cat > firstfile
```

Jetzt die Eingabetaste (enter) drücken und die Testzeile eingeben:

```
Dieses ist die Testzeile
```

Wieder die Eingabetaste (enter) drücken und das ganze mit `Ctrl-D` oder `Strl-D` beenden.

Die Inhalte der Datei kann man sich dann mit dem Befehl:

```
cat firstfile
```

ansehen.

1.1.4. Ein Datei kopieren

Um ein genaues Duplikat einer Datei zu erzeugen, kann das Kommando `cp` (copy) verwendet werden. Soll etwa eine Kopie der Datei `firstfile` namens `secondfile` erstellt werden, geschieht dieses mit dem Kommando:

```
cp firstfile secondfile
```

Jetzt gibt es zwei Dateien mit unterschiedlichen Namen, die beide denselben Inhalt haben. Sollte es die Datei `secondfile` bereits geben, so wird der Inhalt mit dem der Datei `firstfile` überschrieben.

1.1.5. Eine Datei umbenennen

Im Gegensatz zu anderen Betriebssystem hat UNIX kein eigenständiges Kommando zum Umbenennen einer Datei. Stattdessen wird das Kommando `mv` (move) verwendet, dass Dateien umbenennen oder in ein anderes Verzeichnis verschiebt.

Um den Namen einer Datei zu ändern, wird as folgende Kommando verwendet.

```
mv oldfile newfile
```

Was passiert? Der Inhalt der Datei `oldfile` wird in die neue Datei `newfile` geschrieben und `oldfile` wird vollständig entfernt. Faktisch wurde damit `oldfile` in `newfile` umbenannt.

Wie bei dem Kommando `cp` überschreibt auch `mv` eine bereits bestehende Datei. Sollte also bei der Umbenennung `newfile` schon existieren, so wird der Inhalt mit dem von `oldfile` überschrieben.

Wenn eine Datei in ein anderes Verzeichnis verschoben werden soll, so lautet die Syntax

```
mv oldfile /dir/.../dir/newfile
```

1.1.6. Eine Datei löschen

Zum Löschen einer Datei wird das Kommando `rm` (remove) verwendet.

So löscht zum Beispiel...

```
rm file3
```

... die Datei `file3` mit ihrem gesamten Inhalt. Sollen mehrere Dateien gleichzeitig gelöscht werden, so kann eine Liste von Dateien angegeben werden.

```
rm firstfile secondfile
```

Zur Sicherheit wird noch einmal nachgefragt, ob Sie die Datei/en wirklich löschen wollen.

```
rm: remove firstfile (y/n)? y
```

```
rm: remove secondfile (y/n)? n
```

```
Type y or yes to remove a file; type n or no to leave it intact.
```

1.1.7. Ein Verzeichnis anlegen

Durch Verzeichnisse können Dateien sinnvoll gruppiert werden, damit diese schnell und sicher wiedergefunden werden können. Das Kommando ...

```
mkdir project1
```

... legt in dem aktuellen Ordner einen neuen Unterordner mit dem Namen `project1` an. In diesem können dann beispielsweise alle Dateien die zum dem Projekt gehören abgelegt werden.

Das Navigieren und Inspizieren von Verzeichnissen wird später beschrieben.

1.1.8. Dateien in ein anderes Verzeichnis kopieren und verschieben

Die `mv` und `cp` Anweisungen können verwendet werden, um Dateien in ein anderes Verzeichnis zu verschieben oder zu kopieren. Nehmen wir an, Sie wollen einige Dateien aus Ihrem aktuellen Verzeichnis in das neu angelegte Verzeichnis `project1` verschieben. Dieses kann mit der Anweisung

```
mv file1 project1
```

erfolgen. Wollen Sie zudem die Dateien an ihrem ursprünglichen Ort behalten, so müssen sie kopiert werden.

```
cp file1 project1
```

Jetzt befindet sich weiterhin `file1` in dem aktuellen Verzeichnis und eine Kopie davon in dem Unterverzeichnis `project1`.

1.1.9. Ein Verzeichnis umbenennen

Das schon bekannte `mv` Kommando kann auch verwendet werden, um Verzeichnisse umzubenennen oder auch als ganzes zu verschieben. Durch die Anweisung

```
mv project1 project2
```

wird das Verzeichnis `project1` in `project2` umbenannt, wenn es das Verzeichnis `project2` in dem aktuellen Verzeichnis noch nicht gibt.

Wenn aber bereits ein Verzeichnis `project2` im aktuellen Verzeichnis vorhanden ist, dann wird das Verzeichnis `project1` als Unterverzeichnis mit all seinen Dateien in dem Verzeichnis `project2` erzeugt.

1.1.10. Ein Verzeichnis kopieren

Mit dem `cp` Kommando können ganze Verzeichnisse mit ihrem Inhalt kopiert werden. Soll eine Kopie des Verzeichnis `project1` und dem Inhalt mit dem Verzeichnisnamen `proj1copy` erzeugt werden, so geschieht dieses mit:

```
cp -r project1 proj1copy
```

Auch hier gilt wieder, sollte `proj1copy` bereits existieren, so wird in diesem Verzeichnis ein Duplikat des Verzeichnisses `project1` als Unterverzeichnis mit seinem Inhalt erzeugt.

1.1.11. Ein Verzeichnis löschen

Die Anweisung `rmdir` wird verwendet um ein leeres Verzeichnis zu löschen.

```
rmdir project1
```

Hiermit wird das Unterverzeichnis `project1` des aktuellen Verzeichnis gelöscht, wenn dieses leer ist.

Sollte es nicht leer sein, erhalten Sie die Warnung:

```
rmdir: testdir3: Directory not empty
```

Außer der Warnung ist nichts passiert. Wenn sie aber sicher sein wollen, dass das gesamte Verzeichnis und sein Inhalt gelöscht werden soll, so wird die Anweisung, um die Option `-r` erweitert, die rekursiv den gesamten Inhalt löscht.

```
rm -r project1
```

Jetzt ist das Verzeichnis `project1` und sein gesamter Inhalt gelöscht.

1.1.12. In Verzeichnissen navigieren

««« Hier kommt noch was »»»

1.2. Zusammenfassung der Befehle

1.2.1. Mit Dateien arbeiten

```
mv file1 file2
```

Nennt `file1` in `file2` um. Wenn `file2` bereits existiert, dann werden dabei die Inhalte von `file2` überschrieben.

```
cp file1 file2
```

Kopiert `file1` in `file2`. Wenn `file1` bereits existiert, dann werden die Inhalte von `file2` überschrieben.

```
rm file3
```

Löscht `file3`. Vor dem Löschen wird eine Bestätigung angefragt.

1.2.2. Mit Verzeichnissen arbeiten

```
mkdir dir1
```

Legt in dem aktuellen Arbeitsverzeichnis ein neues Verzeichnis `dir1` an.

```
mv dir1 dir2
```

Wenn `dir2` nicht existiert, wird `dir1` in `dir2` umbenannt. Wenn `dir2` existiert, wird `dir1` als Unterverzeichnis in `dir2` verschoben.

```
cp -r dir1 dir2
```

Wenn `dir2` nicht existiert, wird `dir1` als `dir2` kopiert. Wenn `dir2` existiert, wird `dir1` als Unterverzeichnis in `dir2` kopiert.

```
rmdir dir1
```

Löscht das Verzeichnis `dir1`, falls es leer ist.

```
rm -r dir1
```

Löscht das Verzeichnis `dir1` mit allen darin enthaltenen Dateien und Unterverzeichnissen (VORSICHT).

1.2.3. Mit Dateien und Verzeichnissen arbeiten

```
cp file1 dir1
```

Kopiert die Datei `file1` in das existierende Verzeichnis `dir1`.

```
mv file2 dir2
```

Verschiebt die Datei `file2` in das existierende Verzeichnis `dir2`.

2. Der Einstieg

Dieses Skript basiert in weiten Teilen auf dem sehr gelungenen Online-Kurs von Jürgen Wolf. Sie finden den Kurs unter:

https://lueersen.homedns.org/Pronix_final/ckurs/ckurs.html

Ich kann diesen Kurs sehr empfehlen, er geht aber auf vieles mehr ein, als es in dieser Veranstaltung geschehen wird.

Das Skript darf daher nicht weitergegeben werden.

2.1. Hello World!

Um Ihnen den Einstieg zu erleichtern, wollen wir mit dem schon obligatorischen "Hello World" beginnen. Hier das Programm:

```
#include <stdio.h>

int main ( )
{
    printf("Hello  World\n") ;
    return 0;
}
```

2.1.1. Übersetzung mit GNU-C (gcc)

Nehmen wir an, Sie haben das Programm mit irgendeinem Texteditor geschrieben und den Quellcode unter den Namen `hello.c` abgespeichert. Wechseln sie in das Verzeichnis, wo sie den Code abgespeichert haben. Übersetzen sie das Programm dann mit (unter Linux)...

```
gcc -o hello hello.c
```

So werden sie die nächste Zeit Ihre Programme übersetzen. gcc ruft den Compiler auf. `hello` ist der Programmname mit dem sie das Programm dann aufrufen können. `hello.c` enthält den Quellcode, der übersetzt werden soll. Nun sollte in dem Verzeichnis in dem sie das Programm kompiliert und übersetzt haben, ein Programm namens `hello` stehen. Nun können sie das Programm mit...

```
hello
```

... starten, und es folgt die Ausgabe auf dem Bildschirm ...

```
Hello World
```

Gratuliere, Sie haben soeben ihr erstes Programm geschrieben!

2.1.2. Hello World ... in kleinen Schritten

Nun wollen wir unser Programm schrittweise durchgehen. Also zerlegen wir das Programm in seine Einzelbestandteile...

```
#include <stdio.h>
```

`include` ist kein direkter Bestandteil der Sprache C, sondern ein Befehl des sogenannten Präprozessors. Der Präprozessor ist ein Teil Ihres Compilers, der noch nicht das Programm Übersetzt, sondern kontrollierend nicht bleibende Änderungen im Programmtext vornimmt, die nur temporär benötigt werden.

Präprozessorbefehle erkennt man am `#` in der ersten Spalte. Compilieren sie das Programm doch mal ohne `#include <stdio.h>`. Normalerweise sollte jetzt eine Fehlermeldung folgen wie:

Error. function 'printf' should have a prototype.

`printf` kommt doch im Programm vor, sagen sie. Richtig erkannt. `printf` ist nichts anderes, wie eine Funktion, die in `#include <stdio.h>` deklariert ist. D.h. es wird beispielsweise festgelegt, mit welcher Art von Argumenten diese Funktion aufgerufen werden darf und welche Art von Werten sie gegebenenfalls zurückgibt. `#include` - Dateien nennt man Headerdateien. Suchen sie mal das Verzeichnis `include` auf ihrem System (meist unter `/usr/include` oder `/usr/bin/include`. Sie werden noch viele andere Header-Dateien darin entdecken, die wir später noch verwenden werden. Sie können sich die Header-Dateien auch mit einem Editor oder `cat` ansehen. Aber bitte ändern Sie nichts darin. Die Abkürzung `<stdio>` steht für Standart I/O also Standard-Aus- und Eingabe. In dieser Header-Datei werden also u.a. wesentliche Funktionen für die Ein- und Ausgabe definiert.

Zerbrechen Sie sich jetzt nicht den Kopf, wenn sie das jetzt noch nicht so recht verstehen. Wir kommen noch öfters auf die Headerdateien zurück, die sowieso bei jedem Programm benötigt werden. Später werden wir auch eigene Header-Dateien entwerfen und in Programme einbinden.

Weiter zur Programmausführung...

```
int main( )
```

Hier beginnt das Hauptprogramm. Eine `main()` Funktion wird immer benötigt, damit der Compiler weiß, wo er beginnen muss, das Programm zu übersetzen. Auch wenn sie später mehrere Module (Funktionen), also mehrer Quellcodes, compilieren benötigen sie immer eine `main`-Funktion. `main` heisst auf deutsch soviel wie "Hauptfunktion". Das `int` steht dafür, dass dieses Programm beim Beenden eine Dezimalzahl zurückgibt, mit dem sich etwa "überprüfen ließe, ob es korrekt gelaufen ist. In diesem Fall gibt das Programm immer beim Beenden den Wert 0 zurück. Oft gibt man im Fehlerfalle, etwa einen Fehlercode zurück, der Aufschluss geben kann, wo das Programm einen Fehler erkannte. Im Fall einer beliebigen Funktion bedeutet dies, dass diese einen Rückgabewert bekommt. In diesem Programm bekommt unsere `main()` - Funktion den Rückgabewert 0 ...

```
return 0;
```

....was soviel bedeutet, dass unser Programm sauber beendet wurde. wir geben also mit `return` hier der Funktion `main()` den Wert 0 zurück. Mehr dazu erfahren sie in einem späteren Kapitel. Kommen wir zu...

```
{
printf(".....");
}
```

Zwischen den geschweiften Klammern steht der sogenannte Anweisungsblock. Das heißt, in diesem Block steht, was die Funktion `int main()` alles macht. Natürlich können in dem Anweisungsblock weitere Anweisungen und auch weitere Anweisungsblöcke verwendet werden. Das hört sich komplizierter an, als es ist. Ich komme noch darauf zurück.

2. Der Einstieg

Merken sie sich einfach: Geschweifte Klammern fassen Anweisungen zu einem Block zusammen.

Also fassen wir zusammen, was in unserem Block geschieht...

```
printf("Hello World\n");
```

`printf` ist eine Funktion, die in `#include <stdio.h>` festgelegt ist. Deswegen kann der Compiler auch nichts mit `printf` anfangen, wenn die Header-Datei `#include <stdio.h>` nicht im Programm angegeben ist, da in dieser die Funktion `printf` deklariert wird. Mit `printf` kann eine beliebige Stringkonstanten auf dem Bildschirm formatiert ausgegeben werden. Die auszugebende Stringkonstante, in diesem Fall `"Hello Wordl\n"` steht immer zwischen zwei Hochkommata `"..."`. Nicht erlaubt ist, die Stringkonstante über das Zeilenende fortzusetzen, wie in diesem Beispiel:

```
printf("Dieses ist in C  
nicht erlaubt");
```

Es gibt aber eine Ausnahme der Regel, indem man einen `\`(Backslash) als Fortsetzungszeichen setzt, etwas wenn die Stringkonstante zu lang wird. Hierzu das Beispiel:

```
printf("Hier ist die Ausnahme der Regel \  
dies hier ist erlaubt, dank Backslash");
```

Das Zeichen `\n` in der Funktion von `printf` bedeutet **newline** und erzeugt auf dem Bildschirm eine Zeilenvorschub, wie er mit der Tastatur mit der Taste ENTER ausgelöst wird. Es gibt noch mehr sogenannte ESCAPE-Sequenzen, die mit einem Backslash beginnen. Auch dazu werden Sie noch einiges erfahren.

So jetzt nur noch das sogenannte Semikolon `;`. Es wird eigentlich hauptsächlich dazu verwendet, um das Ende einer Anweisung anzuzeigen. Der Compiler weiß hier, aha, hier ist das Ende der Anweisung von `printf` und fährt dann mit der nächsten Zeile bzw. Anweisung fort. Merken sie sich: Anweisungen, denen kein Anweisungsblock folgt, werden mit einem Semikolon abgeschlossen. In unserem Fall ist das Programm dann beendet.

2.1.3. Hello World ...reloaded

Jetzt möchten wir das Programm etwas verändern. Es soll den Namen des Anwenders einlesen und ihn dann begrüßen, damit brauchen wir im wesentlichen zwei neue Elemente:

1. Einen Speicherplatz, in dem wir den eingegebenen Namen des Anwenders speichern können. Dieses geschieht mit der Anweisung

```
char strName[40];
```

Damit stellen wir dem Programm eine Variable mit dem Namen `strName` (einen Speicherpuffer) zur Verfügung, die maximal 40 Zeichen des Namens des Anwenders aufnehmen kann.

2. Eine Funktion, die maximal 40 Zeichen von der Tastatur liest und diese in der Variablen `strName` abspeichert, so dass der Name später verwendet werden kann. Dazu verwenden wir die ebenfalls in `stdio.h` deklarierte Funktion `fgets`. `stdin` gibt an, dass die Zeichen von der Tastatur dem Standard-Eingabegerät kommen. Die Funktion könnte auch die Zeichen aus einer Datei lesen. Davon aber später mehr.

```
fgets(strName, 40, stdin);
```

Jetzt lautet das erweiterte Programm:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char strName[40] ;
6
7     printf ("\nBitte geben Sie Ihren Namen ein:\n" ) ;
8     fgets(strName, 40, stdin);
9
10    printf("\nWillkommen %s\n\n", strName);
11
12    return 0;
13 }
```

Sehen wir uns das kleine Programm und seine Änderungen noch einmal genauer an. In Zeile 5 wird eine Variable `strName` deklariert, die maximal 40 Zeichen (char) aufnehmen kann. In Zeile 7 wird dann einfach nur die Aufforderungen seinen Namen einzugeben auf den Bildschirm geschrieben. Dann werden in Zeile 8 mittels der Funktion `fgets()` maximal 40 Zeichen von der Tastatur (`stdin`) gelesen und in der Variablen `strName` gespeichert. In Zeile 10 wird dann nach einem Wechsel in eine neue Zeile `\n` zunächst der konstante Text `Willkommen` gefolgt von dem Inhalt einer Stringvariablen `%s` ausgegeben. Abgeschlossen wird die Ausgabe durch zwei Leerzeilen `\n\n`. Die gesamte Textkonstante `"\nWillkommen %s \n\n"` regelt also das Format, in dem der Text auszugeben ist. Der Name der auszugebenden Stringvariable wird dann im folgenden mit `strName` angegeben.

Gratulation, jetzt haben Sie Ihr zweites Programm geschrieben, jetzt schon mit einer Texteingabe!

2.1.4. ESCAPE-Sequenzen

Wir haben schon in dem kurzen Programm in der Textkonstanten der `printf`-Anweisung das erste ESCAPE-Sequenzen gesehen. Das `\n` gibt nicht einfach nur eine Zeichensequenz aus, sondern wird in einem Zeilenvorschub umgesetzt. Ähnliche ESCAPE-Zeichen dienen dazu, Zeichen, die normalerweise anders verstanden werden, etwa das " Anführungszeichen, dass normalerweise als Stringanfang oder -ende interpretiert wird und somit nicht ausgegeben wird, dennoch auszugeben. Die einige dieser ESCAPE-Sequenzen sind in dem kleinen Beispielprogramm enthalten.

2. Der Einstieg

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Ich gebe ein akustische Signal aus\n");
6     printf("Ich springe in die naechste Zeile\n");
7     printf("Und ich springe 2 Zeilen weiter\n\n");
8     printf("Ich loese \t einen Zeilenvorschub aus\n");
9     printf("\n\tC\n\tI\n\tS\n\tT\n\tTOLL\n");
10    return 0;
11 }
```

Wenn Sie dieses Programm übersetzen und laufen lassen. Erhalten Sie das folgende Ergebnis am Bildschirm angezeigt:

```
Ich gebe ein akustische Signal aus
Ich springe in die nächste Zeile
Und ich springe 2 Zeilen weiter
```

```
Ich löse          einen Zeilenvorschub aus

      C
      I
      S
      T
      TOLL
```

ESCAPE-Sequenzen werden auch oft Steuerzeichen genannt. Wie der Name sagt, und das Programm eben gezeigt hat, können sie die Ausgabe auf dem Bildschirm beeinflussen. Steuerzeichen beginnen immer mit dem \ und der nachfolgenden Konstante. Die folgenden Steuerzeichen werden oft verwendet:

\a	BEL(bell) akustisches Warnsignal
\n	NL(newline) Cursor geht zum Anfang der nächsten Zeile
\t	HT(horizontal tab) Zeilenvorschub zur nächsten horizontalen Tabulatorposition
\"	" wird ausgegeben
\'	' wird ausgegeben
\?	? wird ausgegeben
\\	\ wird ausgegeben

2.2. Challenges

2.2.1. Ein Programm, um alle Zeilen von helloworld.c auszudrucken

Schreiben sie das Programm "helloWorld.c" aus dem vorherigen Abschnitt 2.1.3 so um, dass es seinen Quelltext am Bildschirm ausgibt. Dieses ist mit dem bisher erlernten `printf` und den Escape-Sequenzen zu lösen.

2.2.2. Kalender

Schreiben Sie ein Programm, dass den aktuellen Monat in dem untenstehenden Format ausgibt:

Oktober 2021						
Mo	Di	Mi	Do	Fr	Sa	So
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

3. Variablen und grundlegende Datentypen

3.1. Eigenschaften von Variablen

Wir haben bereits erste Variablen deklariert. In C haben alle Variablen aber vier ganz wesentliche Eigenschaften, auf die wir immer wieder zurückkommen werden.

Attribute	Beschreibung
Name	Der Name der Variablen mit dem im Programm auf sie Bezug werden kann.
Typ	Der Datentyp der Variablen (Ganze Zahl, Fließkommazahl, Buchstabe, ...).
Wert	Der Wert der Variablen, der an den Speicherplatz der Variablen geschrieben wird.
Adresse	Die Adresse der Variablen, die den Speicherplatz der Variablen angibt.

So kann etwa eine Variable mit dem Namen `x` vom Typ Integer sein und mit vier Bytes dargestellt den Wert `47` besitzen. Die vier Bytes sind ab der Adresse `1AD0` (hexadezimal) hinterlegt. Die Adresse der Variablen wird dabei in der Regel durch den Compiler oder das Programm vergeben.

3.2. Grundlegende Datentypen

Beim Programmieren werden viele Datentypen verwendet. Wir haben bereits Strings (Zeichenketten) kennengelernt. Aber daneben gibt es eine Vielzahl anderer Datentypen, wie Zahlen, Booleans, Arrays oder komplexe Datenstrukturen, die wir noch kennenlernen werden. In diesem Kapitel befassen wir uns mit ganz grundlegenden Datentypen.

1. Integers
2. Fließkommazahlen
3. Buchstaben

Darüber hinaus gibt es viele weitere Typen, wie `short`, `long`, ... die sich im wesentlichen durch die mehr oder weniger intensive Nutzung von Speicherplatz unterscheiden.

3.3. Arbeiten mit ganzen Zahlen (int)

Integer sind ganze Zahlen, die sowohl positive als auch negative Zahlen, wie zum Beispiel -3, -2, -1, 0, 1, 2, 3 darstellen können. Der Integer Datentyp kann maximal 4 Byte an Informationen aufnehmen. Eine Integer Variable `x` wird mit dem `int` Schlüsselwort deklariert, wie im folgenden dargestellt.

```
int x;
```

Es ist auch möglich, mehrere Variablen auf einmal zu deklarieren, dieses sorgt aber nicht immer für einen gut lesbaren Code.

```
int x, y, z;
```

In diesem Falle werden drei Variablen `x`, `y`, `z` als Integer deklariert. Wie immer in C muss die Anweisung mit einen `;` abgeschlossen werden.

3.4. Arbeiten mit Fließkommazahlen (float)

Fließkommazahlen können sehr große aber auch sehr kleine Zahlen sein, die in der Regel durch 8 Bytes repräsentiert werden. Beispiele sind

- 9.45612
- 356808.2
- -342.567
- -45678912.34567

Sollen Variablen vom Typ Fließkommazahl deklariert werden, so wird das Schlüsselwort `float` verwendet.

```
float operand1;  
float operand2;  
float result1, result2;
```

In diesem Beispiel werden vier Variablen `operand1`, `operand2`, `result1` und `result2` deklariert.

3.5. Arbeiten mit Buchstaben (*char*)

Variablen, die für einen Buchstaben stehen, werden in C in einem Byte gespeichert. Damit sind prinzipiell 256 verschiedene Zeichen möglich. Hier geht es zunächst um einzelne Zeichen, nicht um Zeichenketten, die als Arrays oder Strings dargestellt werden. Wir kommen noch genauer dazu. Die verschiedenen Zeichen erhalten dabei gemäß dem American Standard Code for Information Interchange (ASCII) jeweils einen Wert. So entspricht etwa der Wert 90 dem Großbuchstaben 'Z' und der Wert 122 dem Kleinbuchstaben 'z'.

In C werden ZeichenVariablen (Character) mit dem Schlüsselwort **char** deklariert.

```
char firstInitial;
char middleInitial;
char lastInitial;
```

Soll einer Zeichenvariable ein Zeichen zugeordnet werden, so muss dieses Zeichen in einfachen (') Anführungszeichen eingeschlossen werden.

3.6. Initialisierung von Variablen und der Zuweisungsoperator

Wenn eine Variable deklariert wird, so zeigt die Adresse der Variablen auf einen freien Speicherplatz, so der Wert der Variablen stehen soll. Es ist nie sicher, dass dort zu Beginn des Programms nichts steht. Meist kann sich dort ein beliebiger, zufälliger Inhalt aus vorangegangenen Programmläufen befinden, der nicht immer Sinn macht. Um dieses zu verhindern kann man Variablen bereits bei der Deklaration mit sinnvollen Werten initialisieren.

```
/* Deklariere die Variablen */
int x;
char firstInitial;
/* Initialisiere die Variablen */
x = 0;
firstInitial = '\0';
```

Dieser Code deklariert zwei Variablen : **x** vom Typ Integer und **firstInitial** vom Typ Character. Danach wird den Variablen jeweils ein Wert zugeordnet, sie werden initialisiert, wie man sagt. Hier wird die Variable **x** mit dem Wert 0 und die Variable **firstInitial** mit dem Zeichen `\0` initialisiert, dass auch als **NULL** Zeichen bekannt ist.

Den Code in dem obigen Beispiel kann man auch einfacher (kürzer) schreiben, indem die Deklaration und die Initialisierung zusammengefasst werden.

```
/* Deklariere und initialisiere die Variablen */
int x = 0;
char firstInitial = '\0';
```

Beide Beispiele sind identisch in ihrer Auswirkung.

3.7. Den Inhalt von Variablen ausgeben

Wie wir schon gesehen haben, kann der Inhalt der Variablen mit der `printf` Funktion ausgegeben werden. Allerdings muss jetzt der Datentyp der Variablen berücksichtigt werden, wie in dem folgenden Programm gezeigt wird:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     // variable declaration
6     int x;
7     float y;
8     char c;
9     // variable initialization
10    x = -1234;
11    y = 555.44;
12    c = 'M';
13    // print all variables to the standard output (Bildschirm)
14    printf("\nDer_Wert_der_Integer_Variablen_x_ist_%d", x);
15    printf("\nDer_Wert_der_Float_Variablen_y_ist_%f", y);
16    printf("\nDer_Wert_der_Character_Variablen_ist_%c", c);
17    return 0;
18 }
```

Dieses Programm zeigt beispielhaft die Konzepte dieses Kapitels, die Deklaration und Initialisierung von Variablen sowie die anschließende Ausgabe mittels `printf`, wobei durch `%d`, `%f` und `%c`, den sogenannten Konversionsspezifikatoren, festgelegt wird, wie die Variablen auszugeben sind. Hierauf wird im nächsten Abschnitt eingegangen.

3.7.1. Konversionsspezifikatoren

Da die Information im Speicher immer nur als ein Muster von Einsen und Nullen abgelegt ist, muss in dem Programm festgelegt werden, wie diese den für einen Menschen lesbar, etwa durch die Funktion `printf` dargestellt werden sollen. Diese scheinbar schwierige Aufgabe wird durch die Konversionsspezifikatoren leicht gemacht. Diese bestehen immer aus zwei Zeichen. Das erste Zeichen ist dabei das `%` Zeichen, das von einem weiteren Zeichen gefolgt wird. Dieses gibt an, wie die Folge von Einsen und Nullen konvertiert werden soll.

Konversionsspezifikator	Beschreibung
<code>%d</code>	Eine Ganzzahl, ggf. mit Vorzeichen, wird ausgegeben.
<code>%f</code>	Eine Fließkommazahl wird ausgegeben.
<code>%c</code>	Ein einzelnes Zeichen wird ausgegeben.
<code>%s</code>	Eine Zeichenfolge, ein String, wird ausgegeben.

Bei Fließkommazahlen gibt es noch eine Besonderheit. Hier will man oft erreichen, dass eine bestimmte Anzahl von Nachkommastellen ausgegeben wird.

Hier ein Beispiel:

```
float result;  
result = 3.123456;  
printf("Der Wert von result ist %f", result);
```

Üblicherweise werden immer 6 Nachkommastellen ausgegeben. Diese ist manchmal etwas störend und kann zu unleserlichen Ausgaben führen. Daher kann man explizit die Anzahl der Nachkommastellen angeben.

```
printf("\n%.1f", 3.123456);  
printf("\n%.2f", 3.123456);  
printf("\n%.3f", 3.123456);  
printf("\n%.4f", 3.123456);  
printf("\n%.5f", 3.123456);  
printf("\n%.6f", 3.123456);
```

Am Bildschirm liest sich dann die Ausgabe wie folgt:

```
3.1  
3.12  
3.123  
3.1234  
3.12345  
3.123456
```

Beachten Sie, dass dabei aber keine Rundung stattfindet. Das muss durch den Aufruf einer entsprechenden Funktion erfolgen. Es wird lediglich die Anzahl der auszugebenden Nachkommastellen festgelegt.

3.8. Einlesen mit scanf()

Neben der Ausgabe von Variablen möchte man aber oft Werte von der Tastatur einlesen. Dieses kann mit der `scanf` Funktion erfolgen. Sie liest die einzelnen von der Tastatur kommenden Zeichen und konvertiert sie in den gewünschten Datentyp und weist dann den Wert einer Variablen zu. Der Aufruf erfolgt durch

```
scanf("Konversionsspezifikator", variable);
```

Durch den Konversionsspezifikator wird `scanf` mitgeteilt, wie die eingegebenen Zeichen zu konvertieren sind. Es werden dabei dieselben Konversionsspezifikatoren verwendet, die auch bei `printf` Verwendung finden. Sie brauchen also gar nichts Neues zu lernen. Die Funktion `scanf` wird in `stdio.h` deklariert.

3. Variablen und grundlegende Datentypen

Konversionsspezifikator	Beschreibung
%d	liest eine Ganzzahl
%f	liest Fließkommazahl
%c	liest ein einzelnes Zeichen
%s	liest eine Zeichenfolge, einen String

Das folgende Beispiel nutzt dieses aus, indem zwei Werte von der Tastatur gelesen werden und dann addiert werden.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int iOperand1 = 0;
6     int iOperand2 = 0;
7     int iResult = 0;
8
9     printf("\n\tAdder Program, by Keith Davenport\n");
10    printf("\nEnter first operand: ");
11    scanf("%d",&iOperand1);
12    printf("\nEnter second operand: ");
13    scanf("%d",&iOperand2);
14
15    iResult = iOperand1 + iOperand2;
16    printf("\nThe result is %d\n", iResult);
17    return 0;
18 }
```

Eines fällt bei der `scanf` Funktion auf. Nach der Angabe des Konversionsspezifikators folgt die Angabe der Variablen, die den Wert aufnehmen soll. Hier ist aber noch ein `&` vorangestellt. Hiermit wird die Speicheradresse, ein Pointer auf den Speicherplatz, wo der Inhalt der Variablen abgelegt werden soll, angegeben. Während also `iOperand1` den Wert angibt, bedeutet `&iOperand` einen Pointer (eine Adressangabe), wo der Wert gespeichert werden soll. Ein häufiger Fehler bei der Verwendung von `scanf` besteht darin, dass das `&` vergessen wird.

3.9. Arithmetik in C

Wie an dem obigen Beispiel gezeigt, kann man mit C umfangreiche Berechnungen durchführen. Die gebräuchlichen Operatoren sind alle vorhanden.

Operator	Beschreibung	Beispiel
*	Multiplikation	<code>fResult = fOperand1 * fOperand2</code>
/	Division	<code>fResult = fOperand1 / fOperand2</code>
%	Rest (Modulo)	<code>fResult = fOperand1 % fOperand2</code>
+	Addition	<code>fResult = fOperand1 + fOperand2</code>
-	Subtraktion	<code>fResult = fOperand1 - fOperand2</code>

Dabei werden komplexe Ausdrücke in der folgenden Reihenfolge ausgewertet:

Reihenfolge der Auswertung	Beschreibung
()	Klammern werden zuerst ausgewertet. Dabei wird von innen nach außen vorgegangen
*,/,%	diese Operationen werden als zweites von links nach rechts evaliiert.
+, -	Diese Operationen werden als letztes von links nach rechts evaluiert

Die Reihenfolge der Auswertung eines komplexen arithmetischen Ausdrucks orientiert sich dabei an der auch sonst üblichen Reihenfolge.

3.10. Challenges

3.10.1. Umsatzberechnung

Schreiben Sie ein Programm, das den Anwender auffordert, die Anzahl und den Preis einer verkauften Ware einzugeben, um dann den Umsatz nach der Formel:

$$fTotalUmsatz = fPreis * iAnzahl$$

zu berechnen, und anschließend den damit erzielten Umsatz auszugeben.

3.10.2. Umsatzberechnung und Gewinn

Erweitern Sie das Programm so, dass auch die Einkaufskosten der Ware erfragt werden, so dass in Folge auch der erzielte Gewinn ausgegeben werden kann.

3.10.3. Body Mass Index

Schreiben Sie ein Programm, dass zunächst das Körpergewicht und die Größe einer Person abfragt, um dann den Body Mass Index zu berechnen und auszugeben.

4. Bedingte Verzweigungen

In Programmen begegnet man immer wieder sogenannten bedingten Verzweigungen, diese werden manchmal auch als Entscheidungen, Kontrollstrukturen, oder ähnlich bezeichnet. Der Grundgedanke ist ganz einfach. In Abhängigkeit von einem oft berechneten Wert soll ein Programm unterschiedlich verlaufen. Ein ganz einfaches Beispiel wäre ein Programm, das zunächst aus Körpergröße und -gewicht den Body Mass Index berechnet, und welches dann ausgibt, ob die Person über-, normal- oder untergewichtig ist. In Abhängigkeit von dem Wert den Body Mass Index werden also unterschiedliche Anweisungen (hier zum Ausgeben von Text) ausgeführt.

4.1. if ... else ...

Der grundsätzliche Aufbau einer solchen bedingten Verzweigung hat dann folgende Form:

```
if (logischer Ausdruck wahr)
{
    Anweisungen1;
    ...
}
else
{
    Anweisungen2;
    ...
}
```

Immer wenn der logische Ausdruck wahr wird, und wir werden gleich sehen, wie so etwas formuliert werden kann, dann wird die Anweisungen1 ausgeführt, ansonsten werden die Anweisungen2 ausgeführt. Besteht Anweisungen1 oder Anweisungen2 aus nur einer Anweisung, dann kann auf die umfassenden geschweiften Klammern verzichtet werden. Es kann sogar vorkommen, dass es gar keinen `else` Block gibt. Auf diesen kann dann verzichtet werden. Wir werden das nach einem Abstecher über logische Ausdrücke sehen.

4.1.1. Logische Ausdrücke

Die meisten logischen Ausdrücke, die in Programmen verwendet werden, vergleichen zwei Werte miteinander. Hierfür stellt C die in Tabelle 4.1 dargestellten logischen Operatoren zur Verfügung.

4. Bedingte Verzweigungen

Operator	Beschreibung	Beispiel	Ergebnis
		sei <code>i = 5;</code> und <code>k = 6;</code>	
<code>==</code>	ist gleich	<code>i == k</code>	<code>false</code>
<code>!=</code>	ist nicht gleich	<code>i != k</code>	<code>true</code>
<code>></code>	ist größer als	<code>i > k</code>	<code>false</code>
<code><</code>	ist kleiner als	<code>i < k</code>	<code>true</code>
<code>>=</code>	ist größer oder gleich	<code>i >= k</code>	<code>false</code>
<code><=</code>	ist kleiner oder gleich	<code>i <= k</code>	<code>true</code>

Tabelle 4.1.: Logische Operatoren

Wollten wir also auf ganz einfache Weise jetzt den Body Mass Index abprüfen, so könnte das etwa durch die folgenden Anweisungen erfolgen:

```
iBodyMassIndex = 20;
if (iBodyMassIndex <= 25)
{
    printf("\nFolgt man dem Body Mass Index, dann sind Sie sind nicht übergewichtig."
}
else
{
    printf("\nFolgt man dem Body Mass Index, wären Sie übergewichtig.");
}
```

Die geschweiften Klammern sind hier nicht unbedingt erforderlich, da jeweils nur genau eine Anweisung auszuführen ist. Allerdings macht es bei möglichen Änderungen deutlich, dass man sich im `if`- oder `else`-Block befindet. Hierdurch wird der Programmcode übersichtlicher und bei Änderungen passieren weniger Fehler.

Hier noch ein kleines Beispiel, in dem der Anwender selber die Entscheidung über den weiteren Programmverlauf treffen kann.

Listing 4.1: Einfaches Eingabemenü

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char cResponse = '\0';
6     int iHealth = 0;
7     int iError = 0;
8
9     printf("\n\tIn-Battle-Healing\n");
10    printf("\na\tDrink_Health_Potion");
11    printf("\nb\tResume_Battle\n");
12
```



```

13     printf("\nEnter your selection:");
14     scanf("%c", &cResponse);
15
16     if (cResponse == 'a')
17     {
18         printf("\nNow drinking health potion\n");
19         iHealth = iHealth + 1;
20     }
21     else
22     {
23         if (cResponse == 'b')
24         {
25             printf("\nNow resuming battle\n");
26         }
27         else
28         {
29             printf("\nError: Unvorhergesehene Eingabe!");
30             iError = 1;
31         }
32     }
33     return iError;
34 }

```

Probieren Sie dieses Programm aus, um zu sehen, wie es auf die verschiedenen Eingaben reagiert. Bis jetzt haben wir recht einfache logische Ausdrücke gesehen, die im wesentlichen auf den Vergleich einer Variablen mit einem konstanten Wert hinauslaufen.

Darüber hinaus können natürlich auch Variablen miteinander verglichen werden, und es können mit den booleschen Operatoren **and**, **or** und **not** unter Verwendung von Klammern () komplexe logische Ausdrücke gebildet werden. Dabei gelten die in der Tabelle 4.2 dargestellten Rechenregeln für die Booleschen Operatoren.

Hinweis: Jeder von 0 (null) verschiedene Ausdruck wird dabei von als Wahr C interpretiert. So wäre etwa auch der Ausdruck "5-2" wahr. Nur Ausdrücke, die genau 0 (null) sind, sind logisch falsch.

4. Bedingte Verzweigungen

Operator	Bedeutung	Ausdruck1	Ausdruck2	Beispiel
&&	and	bA false true false true	bB false false true true	bA && bB false false false true
	or	bA false true false true	bB false false true true	bA bB false true true true
!	not	bA false true		!bA true false

Tabelle 4.2.: Boolsche Operatoren

4.2. Das switch-Statement

Gerade das Beispiel der Evaluation der Nutzereingabe zeigt, dass sich leicht unübersichtlich verschachtelte `if...else ...` Strukturen ergeben, wenn die Nutzereingabe evaluiert werden soll. Hier bietet C mit der `switch`-Anweisung eine sehr gute Möglichkeit das Programm übersichtlicher zu gestalten. Die Anweisung testet dabei eine Variable oder einen Ausdruck `a` auf mehrere mögliche Werte. Im folgenden Listing 4.2 erkennt man die Grundstruktur.

Listing 4.2: Aufbau der switch-Anweisung

```
1 switch(a) {  
2     case 1:  
3         printf("a_ist_eins\n");  
4         break;  
5     case 2:  
6         printf("a_ist_zwei\n");  
7         break;  
8     case 3:  
9     case 4:  
10        printf("a_ist_drei_oder_4\n");  
11        break;  
12    default:  
13        printf("a_ist_irgendwas_anderes\n");  
14        break;  
15 }
```

In die Klammern nach dem Schlüsselwort `switch` steht die Variable oder der Ausdruck, der ausgewertet werden soll. Danach folgen jeweils mit dem Schlüsselwort `case` die verschiedenen Fälle. Nach dem Doppelpunkt kommen dann die im jeweiligen Fall

auszuführenden Anweisungen. Überlicherweise wird der `case`-Block mit `break`; abgeschlossen. Hierdurch wird bei erfolgreichem Ausführen eines Falles die `switch` Anweisung verlassen. Wird kein `case` Fall erreicht, wird der `default` Block ausgeführt. Hat ein `case`-Block keine `break`;-Anweisung, so werden zusätzlich die Anweisungen des nachfolgenden `case`-Blocks ausgeführt, wie im Listing 4.2 gezeigt. Wollte man dasselbe mittels `if ... else` erreichen, so müssten viele solche Anweisungen geschachtelt werden.

4.3. Nützliche Funktionen

4.3.1. `isdigit()` und `isalpha()` Funktion

Gerade bei der Überprüfung der Eingabe des Nutzers kommen oft die Funktionen `isdigit()` und `isalpha()` zum Einsatz. `isdigit()` prüft, ob eine Variable oder ein Ausdruck eine Ziffer darstellt. Ganz analog überprüft `isalpha()`, ob ein Zeichen eines Alphabets vorliegt. Groß- oder Kleinschreibung spielen dabei keine Rolle. Damit beide Funktionen genutzt werden können, müssen sie mittels der Präprozessoranweisung `#include <ctype.h>` deklariert werden.

Listing 4.3: Beispiel für `isdigit()` und `isalpha()`

```

1 #include<stdio.h>
2 #include<ctype.h> // isalpha() und isdigit() werden deklariert
3
4 int main() {
5     char val1 = 's';
6     char val2 = '8';
7
8     // Anwendung von isalpha()
9     if(isalpha(val1))
10         printf("The character %c is an alphabet\n", val1);
11     else
12         printf("The character %c is not an alphabet\n", val1);
13
14     if(isalpha(val2))
15         printf("The character %c is an alphabet\n", val2);
16     else
17         printf("The character %c is not an alphabet", val2);
18
19     // Anwendung von isdigit()
20     if(isdigit(val1))
21         printf("The character %c is a digit\n", val1);
22     else
23         printf("The character %c is not a digit\n", val1);
24
25     if(isdigit(val2))
26         printf("The character %c is a digit\n", val2);
27     else
28         printf("The character %c is not a digit", val2);

```

4. Bedingte Verzweigungen

```
29     return 0;
30 }
```

4.3.2. Zufallszahlen - rand()

Spiele oder Verschlüsselungsprogramme leben von Zufallszahlen. Ebenso können diese eine zufällige Eingabe simulieren oder ganze Datensätze mit zufälligen Inhalten füllen. Eine sehr häufig verwendete Funktion zur Erzeugung von zufälligen ganzzahligen Zufallszahlen ist `rand()`. Allerdings ist ein Nachteil dieser Funktion, dass sie immer dieselben zufälligen Zahlen generiert, wenn man nicht aufpasst. Um dieses zu verhindern, kann die Funktion `srand()` vor dem ersten Aufruf von `rand()` mit einem zufälligen Inhalt gefüllt werden. Beide Funktionen sind in `stdlib.h` deklariert. Für die meisten Zwecke reicht es aus, einfach die aktuelle Zeit in Sekunden als sogenannten Seed zu verwenden. Hierfür kann die in `time.h` deklarierte Funktion `time()` genutzt werden, mit der die aktuelle Zeit in Sekunden seit dem 1.1.1970 abgefragt werden kann.

Um ganzzahlige Zufallszahlen in einem engen Rahmen zu erzeugen, beispielsweise die Ziffern 1 bis 6 eines simulierten Würfels, wird oft auf den Modulo-Operator `%` zurückgegriffen.

```
iRandom = (rand() % 6) + 1;
```

Da `rand() % 6` zufällig Werte zwischen 0 und 5 generiert, wird noch 1 hinzuaddiert, so dass sich die Werte dann zwischen 1 und 6 bewegen. So können beispielsweise die Würfe eines Würfels simuliert werden.

Listing 4.4: Zahlenratespiel

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main()
6 {
7     int iRandomNum = 0;
8     int iResponse = 0;
9
10    // initialisiere den Zufallsgenerator mit der aktuellen Zeit
11    srand(time(NULL));
12
13    // erzeuge eine Zufallszahl zwischen 1 und 10
14    iRandomNum = (rand() % 10) + 1;
15
16    // rate die Zufallszahl
17    printf("\n\nRate die Zufallszahl zwischen 1 und 10: ");
18    scanf("%d", &iResponse);
19
20    // vergleiche die Zufallszahl mit der geratenen Zahl
21    if (iResponse == iRandomNum)
```

```

22     {
23         printf("\nHURRA!_Richtig_geraten);
24     }
25     else
26     {
27         printf("\nSorry, leider falsch geraten.");
28         printf("\nKorrekt ist %d \n\n",_iRandomNum);
29     }
30     return_0;
31 }

```

4.4. Challenges

4.4.1. Wahrheitstabelle

Schreiben Sie ein Programm, das für alle `true/false`-Kombinationen von `bA` und `bB` eine Wahrheitstabelle für den folgenden komplexen Booleschen Ausdruck ausgibt.

$$(bA \parallel bB) \&\& !(bA \parallel !bB)$$

4.4.2. Groß- und Kleinbuchstaben

Modifizieren Sie das Listing 4.1 so, dass sowohl Klein- als auch Großbuchstaben eingegeben werden dürfen und korrekt behandelt werden. Als weitere Eingabemöglichkeit soll ein `q` oder `Q` hinzukommen, dass das Programm beendet und den Fehlercode `-1` zurückgibt. Die Lösung soll dabei einmal mittels `if ... then ...`-Statements und einmal als `switch ...`-Statement ausgeführt werden.

5. Schleifenstrukturen

5.1. Schleifen sind überall

Sehr oft begegnen einem in Alltag Situationen, in denen man etwas wiederholt ausführen muss. In der Programmierung wird das wiederholte Ausführen von Anweisungen meist in einer sogenannten Schleife abgebildet. Typische Beispiele für Schleifen sind etwa:

- Dem Nutzer wird immer wieder ein Menü angezeigt, so dass er nach und nach verschiedene Aktionen ausführen kann.
- Ein Spiel spielen, bis der Spieler gewinnt, verliert oder aufhört.
- Für alle Mitarbeitenden einer Firma den Lohn überweisen
- In einem Spiel solange einen Gesundheitstrunk zu sich nehmen, bis man wieder fit für die Schlacht ist.
- Im Autopilotenstatus bleiben, bis ein Pilot ihn abschaltet und die Steuerung übernimmt.

Alle diese Beispiele haben gemeinsam, dass sie mit Schleifen arbeiten. dabei gibt es bei den Schleifen drei grundsätzliche Varianten, wie sich mit folgenden Anweisungen aus Rezepten illustrieren lassen:

- Schmecke die Suppe ab, und füge ggf. etwas Salz hinzu und schmecke die Suppe erneut ab. Wiederhole dieses, solange nicht ausreichend Salz an der Suppe ist.

Hier wird die Anweisung abzuschmecken und ggf. nachzusalzen solange wiederholt, bis die Suppe ausreichend gesalzen ist. Die Häufigkeit des Abschmeckens und Nachsalzens ist dabei nicht festgelegt. Sollte die Suppe bereits ausreichend gesalzen sein, muss überhaupt nicht gesalzen werden. Nach einem ersten Abschmecken werden die Anweisungen nachzusalzen und erneut abzuschmecken, gar nicht, falls es bereits ausreichend gesalzen ist, oder ein- oder mehrmals ausgeführt, bis ausreichend Salz an der Suppe ist.

- Gebe ein wenig Öl an das Eigelb und schlage es auf. Wiederhole diese Schritte bis die Mayonnaise eine feste Konsistenz bekommt

Im Gegensatz zum vorangegangenen Beispiel, in dem die Suppe beim ersten Abschmecken bereits salzig genug sein kann, wird hier in jedem Fall mindestens einmal Öl an das Eigelb gegeben und aufgeschlagen. Dieses kann sich mehrmals wiederholen.

5. Schleifenstrukturen

Operator	Name	Syntax	Bedeutung
++	Inkrement (postfix)	x++	nach der Auswertung des Ausdrucks um 1 erhöhen
	Inkrement (prefix)	++x	vor der Auswertung des Ausdrucks um 1 erhöhen
--	Dekrement (postfix)	x--	nach der Auswertung des Ausdrucks um 1 verringern
	Dekrement (prefix)	--x	vor der Auswertung des Ausdrucks um 1 verringern
+=	Addition/Zuweisung	x+=y	entspricht x = x + y
	Subtraktion/Zuweisung	x-=y	entspricht x = x - y

Tabelle 5.1.: Abkürzende Operatoren

- Arbeite nacheinander genau 5 Eier in den Teig ein.

In diesem Fall ist die Anzahl der Wiederholungen von vornherein bekannt. Das Hinzufügen der Eier muss genau fünf mal wiederholt werden.

. Auch hier gibt es drei verschiedene Grundformen. die **for**-Schleife, die **while**-Schleife und die **do while**-Schleife.

5.2. Weitere Operatoren

Im Zusammenhang mit Schleifen werden einige weitere Operatoren verwendet, die eigentlich nur Abkürzungen schon bekannter Operationen sind, allerdings in Schleifen oft eine zusätzliche Bedeutung erlangen. Diese sind in Tabelle 5.1 aufgelistet.

Das folgende Programm erläutert den Unterschied zwischen der Prefix- und der Postfix-Variante.

Listing 5.1: Beispiel für prefix- und postfix-Inkrement

```
1 #include<stdio.h>
2
3 int main() {
4     int x = 0;
5     int y = 1;
6     // part 1 - postfix
7     x = y++ * 2; //increments y after the assignment
8     printf("\nThe value of x is: %d\n", x);
9
10    // part 2 - prefix
11    x = 0;
12    y = 1;
13    x = ++y * 2; //increments y before the assignment
14    printf("\nThe value of x is: %d\n", x);
```



```

15
16     return 0;
17 }

```

In dem ersten Teil wird der Wert von `y` mal 2 genommen und das Ergebnis `x` zugewiesen. Erst danach wird `y` um 1 erhöht (postfix). Im zweiten Teil wird dagegen zuerst `y` um 1 erhöht (prefix) und dann mal 2 genommen, um das Ergebnis dann `x` zuzuweisen.

5.3. while-Schleife

Die `while`-Schleife hat eine Ein- und Austrittsbedingung. Dabei sieht das Grundgerüst ganz einfach aus:

```

while (Bedingung == wahr)
{
    /* Ausführen der Anweisungen, bis die Bedingung ungleich wahr ist. */
    Anweisungen;
}

```

Die in der `while`-Schleife enthaltenen Anweisungen werden nur ausgeführt, wenn die Bedingung wahr ist. Nach erfolgter Ausführung des Schleifeninhalts wird die Bedingung erneut getestet. Ist diese wahr, wird die Schleife weiterhin ausgeführt, ansonsten wird sie beendet. Der Schleifeninhalt wird also nicht, einmal oder mehrmals ausgeführt. Das Listing 5.2 soll die Verwendung illustrieren.

Listing 5.2: Beispiel für eine `while`-Schleife

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int iZahl = 1;
6     int iSum = 0;
7
8     while(iZahl <= 10)
9     {
10         iSum += iZahl;          // iSum = iSum + iZahl
11         iZahl++;                // iZahl = iZahl + 1
12     }
13     printf("\nDie Summe aus 1+2+3...+9+10 = %d\n", iSum);
14
15     return 0;
16 }

```

Das Programm führt die `while`-Schleife solange aus, bis `iZahl` größer als 10 ist. Sehen wir uns das im Detail an:

```
while(iZahl <= 10)
```

5. Schleifenstrukturen

Ist `iZahl` kleiner oder gleich 10?! Zunächst ist `iZahl` gleich 1, also springen wir in den Anweisungsblock. Hier wird als erstes die Addition `iSum += iZahl` durchgeführt. Somit ist der Wert von `iSum` jetzt 1. Jetzt wird noch die Variable `iZahl` um 1 erhöht. Nun ist der `while`-Anweisungsblock zu Ende und das Programm springt wieder zum Anfang der `while`-Abfrage. Nun erfolgt wieder die Überprüfung, ob `iZahl` immer noch kleiner oder gleich 10 ist. Da `iZahl` jetzt den Wert 2 hat geht's erneut in die Schleife und der Anweisungsblock wird wieder ausgeführt. Die Schleife wird solange Wiederholt bis `iZahl` den Wert 11 erreicht hat. Dann ist die Bedingung, ob `iZahl` kleiner oder gleich 10 ist falsch und das Programm läuft überspringt den Anweisungsblock der `while`-Schleife und gibt das Ergebnis 55 aus.

5.3.1. Fast-Endlosschleife

Manchmal wünscht man sich eine Schleife, die nahezu immerwährend ausgeführt wird. Man spricht dann von einer Endlosschleife, wie in Listing 5.3 dargestellt.

Listing 5.3: Beispiel für eine kontrollierte Endlosschleife

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char cQuit;
6
7     while(1)
8     {
9         printf("\nAus dieser Schleife geht raus mit 'q' und <ENTER>\n");
10        printf("Eingabe: ");
11        scanf("%c",&cQuit);
12
13        if(cQuit == 'q')
14            break;
15    }
16    printf("Geschafft, sie sind draussen!!\n");
17
18    return 0;
19 }
```

Die Zeile...

```
while(1)
```

stellt eine Endlosschleife dar, da die Bedingung 1 immer wahr ist. Also wird die Schleife im Prinzip ewig ausgeführt. Nun werden sie solange genervt, bis sie entnervt den Buchstaben 'q' eingegeben haben. Dann bricht die Schleife mit der Anweisung ...

```
if(quit == 'q')
    break;
```

... **break** ab und setzt das Programm mit den Anweisungen nach der Schleife fort.

Seien sie vorsichtig mit Schleifen, denn sollten sie falsche Bedingungen stellen, kommt das Programm nicht mehr von selbst aus der Schleife heraus. Meist hilft dann nur noch eine Tastenkombination wie **<Ctrl>+<C>** oder **<Ctrl>+<Alt>+**. Wenn gar nichts mehr geht, dann hilft oft nur noch ein brutaler Shutdown Ihres Rechners. Deswegen stellen Sie sicher, dass sie alles gespeichert haben, wenn Sie üben. Dies gilt übrigens für alle Schleifenprogramme, die Sie programmieren werden.

5.4. **do ... while-Schleife**

. Die Schleife **do...while** ist ganz ähnlich wie eine **while**-Schleife, die wir im vorherigen Abschnitt kennen gelernt haben. Der einzige Unterschied besteht darin, dass die Bedingung nach dem Anweisungsblock der Schleife überprüft wird. Damit wird der Anweisungsblock der Schleife mindestens einmal ausgeführt.

```
do
{
    /* Ausführen der Anweisungen */
    Anweisungen;
}
while (Bedingung == wahr)
```

Der Anweisungsblock der Schleife wird mit dem Wort **do** eingeleitet. Im Anweisungsblock wird dann irgendetwas ausgeführt. Am Ende des Anweisungsblocks steht jetzt unser bekanntes **while** und überprüft, ob die Bedingung wahr ist. Ist die Bedingung wahr, wird unser Anweisungsblock erneut ausgeführt und es wird wieder mit **do** begonnen. Wenn die Bedingung unwahr ist, geht es weiter nach der **while**-Bedingung.

Hinweis: Nach der **do...while**-Schleife folgt nach **while(bedingung);** in jedem Fall ein Semikolon.

Die **do ... while**-Schleife eignet sich hervorragend für ein Auswahlmenü mit einer anschließenden **switch**-Anweisung wie im Listing 5.4 dargestellt.

Listing 5.4: Auswahlmenü mit **do...while**

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int iAuswahl;
6
7     do
8     {
9         printf("\n-1-␣Auswahl1\n");
10        printf("-2-␣Auswahl2\n");
```

5. Schleifenstrukturen

```
11         printf("-3-␣Auswahl3\n");
12         printf("-4-␣Programmende\n");
13         printf("\n\nIhre␣Auswahl:␣");
14         scanf("%d",&iAuswahl);
15
16         switch(iAuswahl)
17         {
18             case 1 : printf("\nDas␣war␣Auswahl␣1␣\n"); break;
19             case 2 : printf("\nDas␣war␣Auswahl␣2␣\n"); break;
20             case 3 : printf("\nDas␣war␣Auswahl␣3␣\n"); break;
21             case 4 : printf("\nProgrammende\n"); break;
22             default: printf("\nUnbekannte␣Auswahl\n");
23         }
24     } while(iAuswahl != 4);
25     return 0;
26 }
```

5.5. Die for-Schleife

Wenn Sie jetzt die Aufgabe hätten, genau 10mal Ihren Namen auszugeben, so würden Sie dieses wahrscheinlich wie im Listing 5.5 lösen:

Listing 5.5: Schleife mit Namensausgabe mittels while

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int i = 1;
6
7     while(i <= 10)
8     {
9         printf("Bond... James␣Bond\n");
10        i++;
11    }
12    return 0;
13 }
```

Typisch für diese Art Schleife ist, dass man weiß, wie oft die Schleife wiederholt werden soll. Dieses kann sehr elegant durch eine **for**-Schleife gelöst werden. Sie sieht ein wenig anders aus:

```
for (initiierung; bedingung; anweisung)
{
    /* führe den Anweisungsblock aus, bis die Bedingung falsch ist */
    Anweisungsblock;
}
```

Damit können wir jetzt unser Programm zur Namensausgabe umschreiben, wie im Listing 5.6 gezeigt.

Listing 5.6: Schleife mit Namensausgabe mittels for

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int i;
6
7     for(i=1; i<=10; i++)
8     {
9         printf("Bond . . . . . James_Bond\n");
10    }
11    return 0;
12 }
```

Nun sieht das Programm schon anders aus. Beim ersten Eintritt in die **for**-Schleife wird die erste Anweisung ausgeführt, nämlich die Variable **i** mit dem Wert 1 initialisiert. Diese Anweisung wird nur einmal beim Eintritt in die Schleife ausgeführt. Die Initialisierung von **i** wird mit einem Semikolon abgeschlossen. Als nächstes wird die Bedingung überprüft, ob **i** kleiner oder gleich 10 ist. Da sie mit dem Wert 1 initialisiert wurde, ist diese Bedingung wahr und der Anweisungsblock wird ausgeführt. Hier gibt das Programm nun etwas auf dem Bildschirm aus. Wenn der Anweisungsblock ausgeführt wurde, springt unser Programm zur anderen Anweisung in der **for**-Schleife, um unsere Variable **i** um den Wert 1 zu erhöhen. Nun ist der Wert von **i** gleich 2. Jetzt wird wieder die Überprüfung, ob die Bedingung wahr ist, durchgeführt (**i**<=10). Da die Bedingung wiederum wahr ist, führt das Programm erneut den Anweisungsblock aus und gibt den Text auf dem Bildschirm aus. Dann wird **i** wieder um 1 erhöht, womit der Wert jetzt 3 ist. Wieder wird die Bedingung abgefragt, die wiederum wahr ist. Das geht jetzt solange, bis unsere Variable **i** den Wert 11 hat. Dann ist die Bedingung falsch und die Schleife wird verlassen und die darauf folgenden Anweisungen ausgeführt.

5.6. Konzentrationsspiel

Einige der bisher besprochenen Techniken werden in dem Beispielprogramm angewendet. Das Programm zeigt kurz drei Zufallszahlen und löscht dann den Bildschirm. Danach werden Sie aufgefordert, die Zahlen in der richtigen Reihenfolge aus dem Gedächtnis wiederzugegeben.

Die Generierung von Zufallszahlen wurde in Abschnitt 4.3.2 erläutert. Neu die der Aufruf von Funktionen des Betriebssystems mit der Funktion **system()**, die hier verwendet wird, um den Bildschirm zu löschen.

Listing 5.7: Konzentrationsspiel

```

1 #include <stdio.h>
2 #include <stdlib.h>
```

5. Schleifenstrukturen

```
3 #include <time.h>
4
5 // #define UNIX 1
6 #define WINDOWS 1
7
8 #ifdef WINDOWS
9 #define CLEARSCREEN for (int i = 0; i < 24; i++) printf("\n")
10 #else
11 #define CLEARSCREEN system("clear")
12 #endif
13
14 int main()
15 {
16     // deklariere und initialisiere die Variablen
17     char cYesNo = '\0';
18     int iResp1 = 0;
19     int iResp2 = 0;
20     int iResp3 = 0;
21     int iElapsedTime = 0;
22     int iCurrentTime = 0;
23     int iRandomNum = 0;
24     int i1 = 0;
25     int i2 = 0;
26     int i3 = 0;
27     int iCounter = 0;
28
29     // initialisiere den Zufallszahlengenerator
30     srand(time(NULL));
31
32     // frage, ob das Spiel gespielt werden soll
33     printf("\nPlay a game of Concentration? (y or n): ");
34     scanf("%c", &cYesNo);
35
36     // wenn ja, dann spiele es
37     if (cYesNo == 'y' || cYesNo == 'Y') {
38         // generiere drei Zufallszahlen
39         i1 = rand() % 100;
40         i2 = rand() % 100;
41         i3 = rand() % 100;
42
43         //zeige die Zahlen
44         printf("\nConcentrate on the next three numbers\n");
45         printf("\n%d\t%d\t%d\n", i1, i2, i3);
46         iCurrentTime = time(NULL);
47         // warte 3 Sekunden
48         do {
49             iElapsedTime = time(NULL);
50         }
51         while ((iElapsedTime - iCurrentTime) < 3);
```

```

52
53         // loesche den Bildschirm
54         CLEARSCREEN;
55
56         // frage nach den drei gezeigten Zahlen
57         printf("\nEnter each number separated by one space: ");
58         scanf("%d%d%d", &iResp1, &iResp2, &iResp3);
59
60         // Sind dieses die drei Zufallszahlen? Die
61         if (i1 == iResp1 && i2 == iResp2 && i3 == iResp3)
62             printf("\nCongratulations!\n\n");
63         else
64             printf("\nSorry, correct numbers were %d %d %d\n\n", i1, i2, i3);
65     }
66     return 0;
67 }

```

5.7. Challenges

5.7.1. Einfaches Zählen

Schreiben Sie ein Programm, das von 100 bis 1 in Schritten von 10 herunterzählt.

5.7.2. Mathequiz

Schreiben Sie ein Programm, das den Anwender zunächst fragt, wieviele Fragen der Form $a * b = ?$ er beantworten möchte. Das Programm sollte dem Anwender gratulieren, wenn die Antwort korrekt ist, und ihm die korrekte Antwort zeigen, falls er falsch geantwortet hat. Das Programm merkt sich, wieviele Antworten insgesamt korrekt beantwortet wurden und zeigt ihm am Ende des Mathequiz sein Ergebnis.

5.7.3. Konzentrationsspiel

Verändern Sie das Konzentrationsspiel in Listing 5.7 so, dass es mit einem Hauptmenü startet. Hier kann man einen Schwierigkeitsgrad angeben oder das Programm beenden. Der Schwierigkeitsgrad bestimmt, wie viele Zahlen sich der Anwender merken soll. Jedesmal, wenn ein Spiel beendet ist, soll wieder das Hauptmenü erscheinen, so dass man mit demselben Schwierigkeitsgrad oder einem anderen weiterspielen kann, bzw. das Spiel beenden kann. Hier das Hauptmenü:

1. Einfach (3 Zahlen werden 3 Sekunden gezeigt)
2. Mittel (5 Zahlen werden 5 Sekunden gezeigt)
3. Schwer (5 Zahlen werden 2 Sekunden gezeigt)
4. Quit (beendet das Spiel)

6. Funktionen

Wir haben bisher lediglich die von C bereits vordefinierten Funktionen verwendet und alles in der Hauptfunktion `main()` geschehen lassen. Diese führt dazu, dass unser Programm oft nicht gut lesbar wird, und die Grundstruktur des Programms nicht einfach zu verstehen ist. War eine Funktion einmal vorhanden, so konnte sie wie eine Blackbox verwendet werden. So war beispielsweise die Funktion `printf()` für uns im Grunde genommen eine solche Blackbox, in der alles relevante für das Ausdrucken unserer Variablen geschah, ohne dass wir näheres über die genaue Funktionsweise kennen mussten. Auf diese Weise wurden wir bei der Programmierung deutlich entlastet.

Oft entwickeln wir selber bei der Programmierung eigene Lösungen, die wir eventuell an anderer Stelle wiederverwenden wollen. Denken wir an das Konzentrationsspiel im Listing 5.7. Dort haben wir einige Anweisungen geschrieben, die dazu führen, dass die Anwendung scheinbar für eine vorgegebene Anzahl Sekunden wartet.

```
iCurrentTime = time(NULL);
// warte 3 Sekunden
do {
    iElapsedTime = time(NULL);
}
while ((iElapsedTime - iCurrentTime) < 3);
```

Eine solche kleine Routine könnte unter Umständen an anderer Stelle wiederverwendet werden, und es wäre hilfreich, eine solche Routine selber zu definieren und als Funktion etwa in der Form `wait(3)` aufrufen zu können.

Damit könnte man so zum einen bereits geschriebenen (und getesteten) Programmcode wiederverwenden und zum anderen, den Programmierenden davon entlasten, wissen zu müssen, wie denn diese Routine im einzelnen funktioniert.

6.1. Deklarieren einer Funktion

Es ist eine gute Praxis in der Programmierung mit C zunächst einmal die Prototypen benötigter Funktionen zu deklarieren, bevor man die eigentliche Funktion schreibt, beziehungsweise programmiert. Nehmen wir das Beispiel der Funktion `wait()`. Wir müssten wissen, was sie genau erledigen soll, und ob diese Funktion irgendwelche Werte zurückgeben soll. Betrachten wir die obigen Anweisungen, so ist hier noch fest programmiert, dass die Funktion genau 3 Sekunden wartet. Viel universeller wäre, wenn sie eine von uns jederzeit festzulegende Anzahl Sekunden warten würde. Diese könnten wir dann als Integer-Argument an die Funktion übergeben. Allerdings macht es bei dieser Funktion keinen

6. Funktionen

Sinn, dass sie irgendeinen Wert zurückgibt, wenn sie fertig ist. Die Deklaration eines solchen Funktionsprototypen in einer Anwendung sähe dann wie folgt aus:

```
#include <stdio.h>
#include <time.h>

void wait(int);      // Funktionsprototyp, es wird ein Integer übergeben
                     // und kein Wert zurückgegeben

int main()
{
}
```

Es werden also in dem Prototypen nur die Datentypen vereinbart, die übergeben beziehungsweise zurück gegeben werden. Dabei können beliebig vieler solcher Funktionsprototypen in einem C-Programm vereinbart werden. Sobald ein solcher Funktionsprototyp definiert ist, kann ein Compiler prüfen, ob die Funktion dann später in dem Programm korrekt aufgerufen wird; es kann also so geprüft werden, dass die `wait()` Funktion ein Integer-Argument zugewiesen bekommt, und dass sie keinen Wert zurückgibt.

Dahingegen deklariert die Anweisung

```
float addTwoFloatValues (float, float);
```

eine Funktion namens `addTwoFloatValues()`, die als Argument zwei `float`-Werte zugewiesen bekommt und einen Wert vom Typ `float` zurückgibt.

6.2. Definieren einer Funktion

Wir kennen bereits eine Funktion, die wir definiert haben, und das ist die `main()` Funktion, die in jedem C-Programm vorhanden sein muss. Die Definition einer weiteren Funktion sieht dabei ganz ähnlich aus. Nehmen wir das Beispiel der `wait()` Funktion:

```
void wait (int nSeconds)
{
    // Die Funktion benötigt time.h

    int iStartTime = 0;
    int iElapsedTime = 0;

    iStartTime = time(NULL);
    // warte nSeconds Sekunden

    do {
        iElapsedTime = time(NULL) - iStartTime;
    }
    while (iElapsedTime < nSeconds);
}
```

```
    return;
}
```

Damit haben wir die Funktion `wait()` definiert und genau festgelegt, was sie zu machen hat. In diesem Fall soll sie die Schleife solange durchlaufen, bis genau `nSeconds` vergangen sind. Jetzt können wir diese Funktion jederzeit verwenden, ohne uns Gedanken machen zu müssen, wie sie genau funktioniert.

Ganz gleiches gilt für eine Funktion `addTwoFloatValues()`, auch diese können wir genauso definieren.

```
float addTwoFloatValues(float fOp1, float fOp2)
{
    float fResult;
    fResult = fOp1 + fOp2;
    return fResult;
}
```

Da der Prototyp ganz am Anfang vorangestellt wird, kann jederzeit auf die später definierten Funktionen zugegriffen werden, da der Compiler die syntaktische Korrektheit der Verwendung durch den Prototypen überprüfen kann, auch wenn die Definition der Funktion noch nicht bekannt ist.

6.3. Funktionsaufrufe

Um die von uns definierten Funktionen einsetzen zu können, werden diese ganz gleich wie die bereits in C definierten Funktionen aufgerufen. Dabei müssen die an die Funktionen übergebenen Argumente mit den Typdeklarationen im Funktionsprototyp übereinstimmen. Das Listing 6.1 illustriert die Funktionsaufrufe für die beiden von uns definierten Funktionen.

Listing 6.1: Beispiel für Funktionsaufrufe

```
1 #include <stdio.h>
2 #include <time.h>
3
4 // Deklarationen der Funktionen
5
6 float addTwoFloatValues (float, float);
7 void wait(int);
8
9 int main()
10 {
11     int iWait = 10;
12     float fOperand1 = 50.0;
13     float fOperand2 = 21.1;
14     float fResult = 0.0;
15
```

6. Funktionen

```
16      // warte 10 Sekunden
17      wait(iWait);
18
19      //addiere die zwei Werte
20      fResult = addTwoFloatValues(f0operand1, f0operand2);
21
22      printf("\nDie Summe ist: %.2f\n", fResult);
23
24      return 0;
25 }
26
27 // Definition der Funktionen
28
29 float addTwoFloatValues(float f0p1, float f0p2)
30 {
31     float fResult;
32     fResult = f0p1 + f0p2;
33     return fResult;
34 }
35
36 void wait (int nSeconds)
37     // Die Funktion benoetigt time.h
38 {
39     int iElapsedTime = 0;
40     int iStartTime = 0;
41
42     iStartTime = time(NULL);
43     // warte nSeconds Sekunden
44
45     do {
46         iElapsedTime = time(NULL) - iStartTime;
47     }
48     while (iElapsedTime < nSeconds);
49     return;
50 }
51
52 /* oder kuerzere Varianten der beiden Funktionen
53
54 float addTwoFloatValues(float f0p1, float f0p2)
55 {
56     return f0p1 + f0p2;
57 }
58
59 void wait (int nSeconds)
60     // Die Funktion benoetigt time.h
61 {
62     int iStartTime = 0;
63
64     iStartTime = time(NULL);
```

```

65
66     // warte nSeconds Sekunden
67     do {
68     }
69     while ((time(NULL) - iStartTime) < nSeconds);
70     return;
71 }
72 */

```

Dieses nicht besonders sinnvolle Beispiellisting 6.1 wartet zunächst 10 Sekunden und gibt dann das Ergebnis aus:

Die Summe ist: 71.10

.

6.4. Sichtbarkeit von Variablen

Wir haben schon oft Variablen deklariert und verwendet. Mit der Einführung eigener definierter Funktionen wird es notwendig, die Sichtbarkeit von Variablen zu klären. Wir werden in diesem Abschnitt zwei wichtige Sichtbarkeiten von Variablen – local und global – kennenlernen. Dabei ist guter Programmierstil, dass Variablen nur solange sichtbar sind, wie sie benötigt werden.

6.4.1. Lokale Variablen

Bisher haben wir immer lokal in der Hauptfunktion `main()` definierte Variablen verwendet, die in der Hauptfunktion sichtbar sind, solange diese ausgeführt wird. Jedesmal, wenn eine Funktion startet, wird ihr Speicherplatz für die in ihr deklarierten Variablen zugeordnet und der Wert darin gespeichert. Wird die Funktion dann wieder verlassen, geht der Speicherplatz und damit der Wert verloren. Da lokale Variablen immer nur in der Funktion sichtbar sind, in der sie deklariert werden, dürfen verschiedene Funktionen denselben Variablennamen verwenden, ohne dass es zu Komplikationen kommt. Dieses Prinzip wird in dem Listing 6.2 ausgenutzt.

Listing 6.2: Beispiel für lokale Variablen

```

1 #include <stdio.h>
2
3 // Deklarationen der Funktion
4
5 int getSecondNumber();
6
7 int main()
8 {
9     int iNum1 = 0;
10
11     printf("\nEnter a number: ");

```

6. Funktionen

```
12     scanf("%d", &iNum1);
13
14     printf("\nYou entered %d and %d\n", iNum1, getSecondNumber());
15     return 0;
16 }
17
18 // Definition der Funktion
19
20 int getSecondNumber()
21 {
22     int iNum1 = 0;
23
24     printf("\nEnter a second number: ");
25     scanf("%d", &iNum1);
26
27     return iNum1;
28 }
```

Dieses führt zu der folgenden Bildschirmausgabe beim Ablauf des Programms

```
Enter a number: 15
Enter a second number: 21
You entered 15 and 21
```

Da die Variable `num1` lokal in jeder Funktion deklariert ist, wird ihr auch in jeder Funktion ein eigener Speicherort zugewiesen. Daher gibt es auch keine Probleme durch ein Wechselseitiges Überschreiben der Inhalte, wie man vermuten könnte.

Lokal sichtbare Variable können in jeder Funktion deklariert werden. Darüber hinaus ist es auch möglich, diese beschränkt auf Anweisungsblöcke `{...}` zu deklarieren, in denen sie dann auch nur sichtbar sind. Gleiches gilt für `for`-Schleifen.

Listing 6.3: Weiteres Beispiel für lokale Variablen

```
1 #include <stdio.h>
2
3 void myFunction(int, int);
4
5 int main()
6 {
7     int i = 1000;
8     int k = 5;
9
10    myFunction(i, k);
11
12    printf("\nNach myFunction ist i gleich: %d", i);
13    printf("\nNach myFunction ist k gleich: %d", k);
14    return 0;
15 }
16
17 void myFunction(int i, int k)
```

```

18 {
19     int n = 0;
20     // i, k, n sind lokal fuer die Funktion deklariert
21     printf("\ni_ist_gleich:%d", i);
22     printf("\nn_ist_gleich:%d", n);
23
24     for(int i = 1, i < k, i++)
25         // i ist nur innerhalb dieser Schleife sichtbar
26     {
27         int k = 100; //nur sichtbar in diesem Anweisungsblock
28
29         printf("\nDer_Schleifenzaehler_i_ist_gleich:%d", i);
30         printf("\nk_ist_gleich:%d", k);
31     }
32
33     printf("\nNach_der_Schleife_ist_i_gleich:%d", i);
34     printf("\nNach_der_Schleife_ist_k_gleich:%d", k);
35     return;
36 }

```

In dem Listing 6.3 wird dieses Prinzip der unterschiedlichen lokalen Sichtbarkeit auf die Spitze getrieben. Der ganz große Vorteil ist, dass die Variable jeweils nur dort deklariert ist, wo sie auch benötigt wird. Dieses führt üblicherweise zu besser verständlichen und fehlerfreieren Programmen. Ob allerdings die vielfache Verwendung desselben Variablennamens in dem beispielhaften Listing 6.3 die Lesbarkeit und Verständlichkeit fördert, mag bezweifelt werden.

6.4.2. Globale Variablen

Wir haben gesehen, dass lokale Variablen immer nur in den jeweiligen Funktionen oder Anweisungsblöcken sichtbar sind, in denen sie auch deklariert werden. Für ein strukturiertes Vorgehen beim Programmieren mit den Zielen der Wiederverwendbarkeit und Lesbarkeit von Programmcode sowie dem Verstecken nicht erforderlicher Informationen durch Blackboxes oder die temporäre Deklaration von Variablen für die Dauer ihrer Verwendung ist diese lokale Sichtbarkeit sehr sinnvoll. Allerdings gibt es Fälle, in denen Daten zwischen und über Funktionen hinweg geteilt werden müssen. Hier kommen sogenannte globale Variablen ins Spiel. Diese werden außerhalb aller Funktionen, dazu zählt auch die Hauptfunktion `main()` deklariert. Das folgende Listing 6.4 erläutert das Vorgehen bei der Nutzung globaler Variablen.

Listing 6.4: Beispiel für globale Variable

```

1 #include <stdio.h>
2
3 void printLuckyNumber();    // function prototype
4 int iLuckyNumber;          // global variable
5
6 int main()
7 {

```

6. Funktionen

```
8     printf("\nEnter your lucky number:");
9     scanf("%d", &iLuckyNumber);
10
11     printLuckyNumber();
12     return 0;
13 }
14
15 // function definition
16 void printLuckyNumber()
17 {
18     printf("\nYour lucky number is: %d\n", iLuckyNumber);
19 }
```

Die Variable `iLuckyNumber` ist global, da sie außerhalb aller Funktionen deklariert wurde. Daher kann auf diese Variable von jeder Funktion sowohl den Wert lesend als auch den Wert verändernd zugegriffen werden. Allerdings sollte genau aus diesem Grund sparsam mit globalen Variablen umgegangen werden. Die Fehlersuche bei einem versehentlichen Zugriff auf eine solche Variable kann sich sehr aufwändig gestalten. Daher sollte immer überlegt werden, welche Sichtbarkeit für die jeweilige Variable erforderlich ist, damit sie ausreichend gegen versehentliches Ändern geschützt ist. So kann in dem Beispiel 6.4 natürlich auch die auszugebende Glückszahl als Argument der Funktion übergeben werden. Damit wäre eine globale Deklaration der Variablen in diesem Beispiel nicht erforderlich.

6.5. Challenges

6.5.1. Konzentrationsspiel 2.0

Schreiben Sie das Konzentrationsspiel aus der Challenge 5.7.3 so um, dass wiederverwendbare Funktionen etwa für das Löschen des Bildschirms, das Warten, das Erzeugen von Zufallszahlen in einen vorgegeben Intervall, für ein einzelnes Spiel (Überlegen Sie welche Argumente dafür erforderlich sind?) definiert und genutzt werden.

6.5.2. Geldwechselautomat

Schreiben Sie ein Programm für einen Geldwechselautomat, der zunächst fragt, für welchen Betrag sie Wechselgeld haben wollen (z.B. 13.64 Euro)? Das Programm gibt ihnen dann aus, in welcher Form sie das Wechselgeld erhalten, wenn sie insgesamt möglichst wenig Münzen oder Scheine haben wollen. In dem Beispiel von 13.64 Euro sind das: ein 10 Euroschein, eine 2 Euromünze, eine 1 Euromünze, eine 50 Centmünze, eine 10 Centmünze und zwei 2 Centmünzen).

7. Arbeiten mit Arrays

Bisher haben wir die speicherorientierten Variablen auf einfache Datentypen beschränkt. So standen nur Datentypen wie Zeichen oder ganze Zahlen `char`, `int`, `long` oder Fließkommazahlen `float`, `double`, `long double` zur Verfügung. Darüber hinaus hätte man aber oft gerne zusammengesetzte Datentypen. Damit wollen wir in diesem Kapitel beginnen.

Mit Arrays haben wir die Möglichkeit eine geordnete Folge von Werten **eines** bestimmten Datentyps zu speichern und bearbeiten.

7.1. Deklarieren von Arrays

Die Deklaration solcher Variablen erfolgt durch:

```
int a[5];  
char x[100];  
float c[5000];
```

Was bedeutet das? Durch die Deklaration `int i[5]` werden 5 Variablen im Array mit dem Namen `a` vom Typ `int` erzeugt und der Speicherplatz reserviert. Mit `char x[100]` wird ein Array `x` von 100 Elementen oder Feldern des Types `char` deklariert.

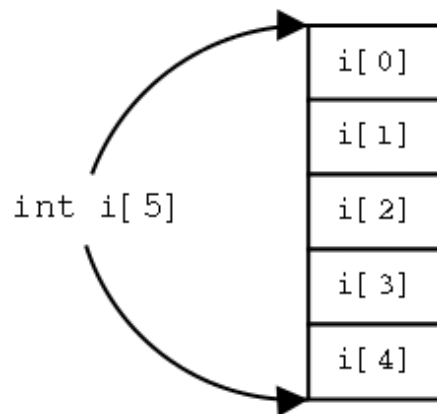


Abbildung 7.1.: Aufbau eines Arrays

Sehen wir uns `int a[5]` in Abbildung 7.1 genauer an: Mit der Deklaration wird ein zusammenhängender Speicherplatz zur Verfügung gestellt, der nacheinander die Integervariablen `a[0]` bis `a[4]` beinhaltet. Deren Inhalt ist noch nicht bekannt.

7. Arbeiten mit Arrays

Wichtig ist in C, dass das erste Element oder Feld eines Arrays immer mit dem Index 0 beginnt. Der höchste erlaubte Index ist damit immer eins weniger als die Anzahl der deklarierten Felder. In ganz gleicher Weise wird ein Array `x` deklariert, dass aus einhundert Feldern besteht. Hierfür wird ein zusammenhängender Speicherplatz reserviert. Ebenso geschieht dieses für das Array `c`, das aus 5000 Feldern des Typs `float` besteht. Hierfür wird ein zusammenhängender Speicherplatz von 5000 mal 4 bytes (Speicherbedarf des Datentyps `float`) reserviert. Soll das Programm in einer Umgebung mit nur wenig Speicherplatz laufen, so muss natürlich bei der Deklaration von Arrays darauf geachtet werden, dass diese nicht unnötig groß werden.

Wichtig: Die Größe eines Arrays muss bereits zum Zeitpunkt der Übersetzung durch den Compiler bekannt sein. Die Größe eines Arrays kann hier also nicht durch eine Variable angegeben werden.

Es gibt Methoden, die es erlauben, Arrays auch dynamisch zur Laufzeit zu erzeugen, diese werden wir aber erst in einem späteren Kapitel kennenlernen. Damit kann dann der Speicherbedarf eines Arrays an einen wirklichen Bedarf angepasst werden und so der Speicher effizient genutzt werden.

7.1.1. Verwenden von Arrays

In dem Listing 7.1 wird dargestellt, wie einfach mit einem Array umgegangen werden kann.

Listing 7.1: Zugriff auf einzelne Felder eines Arrays

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int a[5];          /*Array mit 5 int Elementen*/
6
7     /*Wertzuweisungen an den einzelnen Felder des Arrays*/
8
9     a[0] = 5;
10    a[1] = 100;
11    a[2] = 66;
12    a[3] = 77;
13    a[4] = 1500;
14
15    /*Ausgabe einzelner Felder des Arrays*/
16
17    printf("Array a[0]=\n",a[0]);
18    printf("Array a[3]=\n",a[3]);
19    printf("Array a[4]=\n",a[4]);
20
21    return 0;
22 }
```

Hier haben wir jetzt allen 5 Feldern einen Wert mit Hilfe eines Index `a[Index]` übergeben. Dabei ist zu beachten, daß C beim Zählen immer mit der Zahl 0 beginnt. Würden in dem obigen Program die folgenden Zeilen ergänzt,

```
a[5] = 111;
printf("\na[5] = %d", a[5]);
```

dann würde der Compiler einen Fehler generieren oder manchmal würde sich sogar der Rechner aufhängen. Es sind nur 5 Adressen vom Typ 'int' reserviert worden. Die meisten Compiler fangen den Fehler auf, und melden ihn. Sollte allerdings der Index etwa in einer Schleife hochgezählt werden, müssen Sie selber sicherstellen, dass der Indexwert sich immer im erlaubten Bereich von 0 bis zu **Anzahl Felder des Arrays - 1** bewegt. Sonst wird der Programmablauf unvorhersagbar. Da der Index eine Variable ist, kann der Compiler im vornherein nicht feststellen, welche Werte diese Variable im Verlauf des Programms alles annehmen kann.

Häufiger Fehler: Der Index mit dem auf ein Element eines Arrays muss immer geringer sein, als die Anzahl der Feldelemente des Arrays.

7.1.2. Initialisieren von Arrays

Ähnlich wie bei Variablen möchte man diese oft initialisieren. Allerdings müssen hierbei die Werte zwischen geschweiften Klammern stehen.

```
int iBeispiel1[5] = {1, 2, 5, 6, 10};
int iBeispiel2[500] = {0};
```

Im ersten Beispiel werden den 5 Feldern des Arrays `iBeispiel1` jeweils die Werte 1, 2, 5, 6 und 10 zugewiesen. Im zweiten Beispiel wird allen 500 Feldern des Arrays `iBeispiel2` der Wert 0 zugeordnet.

Hinweis: Leider ist es aber nicht möglich, den Inhalt eines Arrays mit einem konstanten Wert verschieden von 0 in der verkürzten Schreibweise des zweiten Beispiels zu initialisieren.

7.2. Mehrdimensionale Arrays

So wie wir die Arrays jetzt kennen gelernt haben, können wir sie uns als eine **Gerade** vorstellen. Es ist aber auch möglich Arrays in beliebig viele Dimensionen zu definieren.

```
int Matrix[4][4];    //zweidimensional
```

Hiermit haben wir ein zweidimensionales Array definiert. Wie wird ein solches Array indiziert und wie werden die Felder angesprochen? Dieses ist in Abbildung 7.2 dargestellt. Der erste Index steht für die Reihe des zweidimensionalen Arrays, der zweite Index für die Spalte.

7. Arbeiten mit Arrays

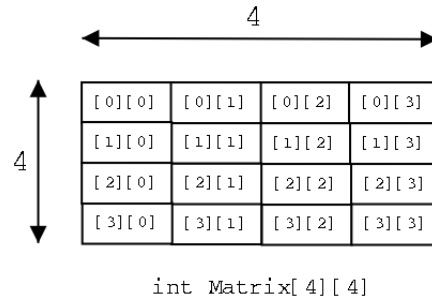


Abbildung 7.2.: Aufbau einen zweidimensionalen Arrays

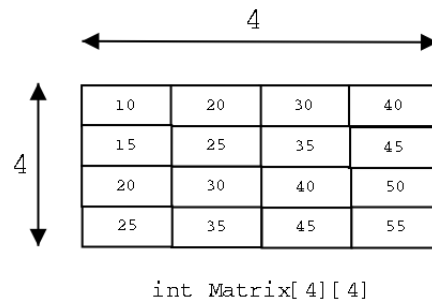


Abbildung 7.3.: Beispiel eines zweidimensionalen Arrays mit Werten

Die Werte in einem zweidimensionalen Array können Sie ähnlich initialisieren, wie bei einem eindimensionalen Array in Abschnitt 7.1.2 beschrieben. Auch hier werden die Werte reihenweise von geschweiften Klammern eingeschlossen, die dann wiederum von einer geschweiften Klammer umschlossen sind. Ähnlich gilt es für höhere Dimensionen.

```
int Matrix[4][4] = { {10,20,30,40},  
                    {15,25,35,45},  
                    {20,30,40,50},  
                    {25,35,45,55} };
```

Wenn das zweidimensionale Array `Matrix` so initialisiert wurde, sieht es wie in Abb. 7.3 aus. Jetzt können Sie leicht auf einzelne Felder zugreifen, indem Sie die beiden Indices angeben. Wenn Sie jetzt zum Beispiel auf das Element mit dem Wert von 40 in dem Array `Matrix` in Zeile 2 und Spalte 2 auf unserer Tabelle zugreifen und den Wert auf 100 ändern wollen, dann reicht die folgende Anweisung:

```
Matrix[2][2]=100;
```

Auch hier müssen Sie in Erinnerung behalten, dass in C der Index mit 0 beginnt. In der Praxis werden mehrdimensionale Arrays sehr häufig für verschiedenen Arten von Berechnungen (Matrizen) benötigt oder bei 2D-Darstellungen von Grafiken, sowie Bildern. Das kleine Beispielprogramm 7.2 zeigt den Umgang mit zweidimensionalen Arrays.

Listing 7.2: Umgang mit zweidimensionalen Arrays

```

1 #include <stdio.h>
2 #define VOL1 3
3 #define VOL2 4
4
5 int main()
6 {
7     int i,j;
8     int myarray[VOL1][VOL2];      /*[3][4]*/
9
10    for(i=0; i<VOL1; i++) {
11        for(j=0; j<VOL2; j++) {
12            printf("Wert_eingeben_fuer_myarray[%d][%d]:",i,j);
13            scanf("%d",&myarray[i][j]);
14        }
15    }
16
17    printf("\nAusgabe_von_myarray[%d][%d].....\n\n",VOL1,VOL2);
18
19    for(i=0; i<VOL1; i++) {
20        for(j=0; j<VOL2; j++) {
21            printf("\t%4d",myarray[i][j]);
22        }
23        printf("\n\n");
24    }
25    return 0;
26 }

```

7.3. Beispielprogramm: Tic-Tac-Toe

Das Spiel Tic-Tac-Toe verwendet auf einfache Weise die bisher gelernten Techniken rund um Arrays, selbst definierte und deklarierte Funktionen und globale Variablen. Am Anfang des Programms finden Sie noch fortgeschrittene Techniken, die Möglichkeiten des Preprozessors verwenden, diese können Sie aber gerne überlesen.

Listing 7.3: Tic-Tac-Toe

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // define clearscreen for windows and unix systems
5 // enable the corresponding definition for unix or windows
6 // #define UNIX 1
7
8 #define WINDOWS 1
9
10 #ifdef WINDOWS
11 #define CLEARSCREEN    for (int i = 0; i < 24; i++) printf("\n")

```

7. Arbeiten mit Arrays

```
12 #else
13 #define CLEARSCREEN    system("clear")
14 #endif
15
16 // function prototypes
17 void displayBoard();
18 int verifySelection(int, int);
19 int isWinningBoard(char);
20 void checkForWin();
21 void clearScreen();
22
23 // global variables
24 char board[9];
25 char cWhoWon = ' ';
26 int  iCurrentPlayer = 0;
27
28 // begin main function
29 int main () {
30     int x;
31     int iSquareNum = 0;
32
33     for (x = 0; x < 9; x++)
34         board[x] = ' ';
35     displayBoard();
36
37     while (cWhoWon == ' ') {
38         printf("\n%c\n", cWhoWon);
39         if ( iCurrentPlayer == 1 || iCurrentPlayer == 0) {
40             printf("\nPlayer_X\n");
41             printf("Enter an available square number (1-9): ");
42             scanf("%d", &iSquareNum);
43             if (verifySelection(iSquareNum, iCurrentPlayer) == 1
44                 iCurrentPlayer = 1;
45             else
46                 iCurrentPlayer = 2;
47
48         }
49         else {
50             printf("\nPlayer_0\n");
51             printf("Enter an available square number (1-9): ");
52             scanf("%d", &iSquareNum);
53             if (verifySelection(iSquareNum, iCurrentPlayer) == 1
54                 iCurrentPlayer = 2;
55             else
56                 iCurrentPlayer = 1;
57
58         } // end if
59         displayBoard();
60         checkForWin();
```

```

61         } // end while
62     }
63
64 // function definitions
65
66 void displayBoard() {
67     CLEARSCREEN;
68     printf("\n\t|\t|");
69     printf("\n\t|\t|");
70     printf("\n%c\t|c\t|c", board[0], board[1], board[2]);
71     printf("\n-----|-----|-----");
72     printf("\n\t|\t|");
73     printf("\n%c\t|c\t|c", board[3], board[4], board[5]);
74     printf("\n-----|-----|-----");
75     printf("\n\t|\t|");
76     printf("\n%c\t|c\t|c", board[6], board[7], board[8]);
77     printf("\n\t|\t|");
78 }
79
80 int verifySelection(int iSquare, int iPlayer) {
81     if ( board[iSquare - 1] == '_' && (iPlayer == 1 || iPlayer == 0) ) {
82         board[iSquare - 1] = 'X';
83         return 0;
84     }
85     else if (board[iSquare - 1] == '_' && iPlayer == 2) {
86         board[iSquare - 1] = 'O';
87         return 0;
88     }
89     else
90         return 1;
91 }
92
93 int isWinningBoard(char cWin)
94 {
95     if (board[0] == cWin && board[1] == cWin && board[2] == cWin) {
96         cWhoWon = cWin;
97         return 1;
98     }
99     else if (board[3] == cWin && board[4] == cWin && board[5] == cWin) {
100         cWhoWon = cWin;
101         return 1;
102     }
103     else if (board[6] == cWin && board[7] == cWin && board[8] == cWin) {
104         cWhoWon = cWin;
105         return 1;
106     }
107     else if (board[0] == cWin && board[3] == cWin && board[6] == cWin) {
108         cWhoWon = cWin;
109         return 1;

```

7. Arbeiten mit Arrays

```
110     }
111     else if (board[1] == cWin && board[4] == cWin && board[7] == cWin) {
112         cWhoWon = cWin;
113         return 1;
114     }
115     else if (board[2] == cWin && board[5] == cWin && board[8] == cWin) {
116         cWhoWon = cWin;
117         return 1;
118     }
119     else if (board[0] == cWin && board[4] == cWin && board[8] == cWin) {
120         cWhoWon = cWin;
121         return 1;
122     }
123     else if (board[2] == cWin && board[4] == cWin && board[6] == cWin) {
124         cWhoWon = cWin;
125         return 1;
126     }
127     else
128     return 0;
129 }
130
131 void checkForWin() {
132     int catTotal = 0;
133     int x;
134
135     // check X
136     if (isWinningBoard('X')) {
137         printf("\nX_Wins!");
138         return;
139     }
140     else {
141         if (isWinningBoard('O')) {
142             printf("\nO_Wins!");
143             return;
144         }
145         else {
146             // check for draw game
147             for (x = 0; x < 9; x++) {
148                 if ( board[x] != ' ')
149                     catTotal += 1;
150             }
151             if ( catTotal == 9) {
152                 cWhoWon = 'C';
153                 printf("\nCAT_Game!\n");
154             }
155             return;
156         }
157     }
158 }
```


7.4. Challenges

7.4.1. Finde den Fehler

Das nachfolgende Programm kann sich unter Umständen aufhängen und lässt sich nicht ordentlich beenden. Erkennen Sie den Fehler? Sollten Sie es testen wollen, speichern Sie vorher alle wichtigen Dateien, so dass sie gegebenenfalls den Rechner neu starten können.

Listing 7.4: Finde den Fehler

```

1 #include <stdio.h>
2
3 int main()
4 {
5
6     int test[10];
7     int i;
8
9     /*
10      * Achtung das Programm kann den Computer abstuerzen lassen!
11      * Bitte vorher alle wichtigen Dateien speichern.
12      */
13
14     for(i=0; i<=10; i++)
15         test[i]=i;
16
17     for(i=0; i<=10; i++)
18         printf("%d, ", test[i]);
19
20     return 0;
21 }
```

7.4.2. Kumulative Summe

Schreiben Sie ein Programm, das ein Array `i1` mit den Werten 1 bis 10 füllt. Schreiben Sie dann in ein weiteres Array `iSum` die kumulativen Summen von `i1`, so dass

$$iSum[i] = \sum_{i=0}^i i1[i]$$

Geben Sie `i1` und `iSum` aus.

7.4.3. Zahlen sortieren

Schreiben sie ein Programm, das sie auffordert, Zahlen in ein Array der Größe ihrer Wahl einzugeben, und das diese dann der Größe nach sortiert ausgibt.

7.4.4. Funktion zur Berechnung der kumulativen Summe

Wandeln Sie Ihr Programm der Challenge 7.4.3 so ab, dass es die Summe der von Ihnen eingegebenen Zahlen berechnet, indem Sie eine Funktion zur Berechnung und Ausgabe der kumulativen Summe eines Arrays mit maximal einhundert Feldern definieren. Überlegen Sie sich, welche Argumente an die Funktion übergeben werden müssen.

7.4.5. Tic-Tac-Toe

Schreiben sie das Programm TicToc so um, dass ein zweidimensionales Array (3×3) anstelle des eindimensionalen Arrays `board[9]` verwendet wird. Wenn Sie wollen, können Sie durch einen Zufallszahlengenerator einen automatischen Gegenspieler programmieren oder diesen noch weiter optimieren.

8. Umgang mit Pointern

Pointer sind ein ganz wesentliche Elemente der Sprache C und können dazu verwendet werden, um auf Variablen, Datenstrukturen und Funktionen über deren Adresse im Speicher zuzugreifen. Pointer sind daher eigentlich nichts anderes als Variablen, die als Werte Speicheradressen aufnehmen können. Diese Speicheradressen verweisen etwa auf andere Variable oder Funktionen. Oder kurz gesagt, eine Pointer-Variable enthält eine Speicheradresse, die meist auf eine andere Variable verweist.

Nehmen wir an, wir haben eine Integer-Variable `iResult`, die den Wert 75 enthält und deren Speicheradresse `0x4523129` ist. Adressen werden meist als Hexadezimalzahlen angegeben. Und wir haben eine Pointer-Variable `myPointer`. Diese soll die Adresse `0x4523129` enthalten, an der sich die Variable `iResult` befindet. An dieser Stelle im Speicher steht der Wert 75. Das bedeutet, dass die Pointer-Variable `myPointer` indirekt auf den Wert 75 verweist.

Hinweis: Sie haben bereits unbewusst mit Pointern gearbeitet, im vorangegangenen Kapitel 7 waren die Namen der Arrays nichts anderes als Pointer auf den Beginn eines Arrays.

8.1. Deklarieren und Initialisieren von Pointer-Variablen

Auch Pointer-Variablen müssen deklariert werden, bevor sie verwendet werden können. wie der folgende Programmausschnitt zeigt:

```
int x = 0;
int iAge = 30;
int *ptrAge;
```

Um eine Pointer-Variable zu deklarieren, muss also lediglich der Verweisoperator `*` davor gesetzt werden. In diesem Beispiel wurden also zwei Integer-Variablen und eine Pointer-Variable deklariert.

Trick: Die Verwendung einer Namenskonvention, etwa die vorangestellte Verwendung von *ptr* bei Pointer-Variablen, hilft dabei den Datentyp und die Verwendung der Variablen im Programm zu erkennen.

8. Umgang mit Pointern

Als der Pointer `ptrAge` deklariert wurde, wurde C außerdem mitgeteilt, dass der Pointer indirekt auf einen Integer-Wert zeigt. Im Moment zeigt die Pointer-Variable aber auf noch gar nichts. Um also indirekt auf einen Wert zu verweisen, muss die Adresse des Wertes noch der Pointer-Variablen zugewiesen werden.

```
ptrAge = &iAge;
```

Damit weise ich die Speicheradresse der Variablen `iAge` der Pointer-Variablen zu. Den *Adresse von*-Operator haben wir schon im Zusammenhang mit der `scanf()` Funktion kennengelernt, im Sinne von schreibe den Werte an die Adresse der angegebenen Variablen. Es ist eigentlich nichts Neues.

Umgekehrt kann ich aber auch den Inhalt an der Adresse auf die eine Pointer-Variable verweist einer anderen Variablen zuweisen — wie jetzt gezeigt.

```
int iMyAge;
...
iMyAge = *ptrAge;
```

Jetzt enthält die Integer-Variable `iMyAge` den Wert 30.

Umgekehrt kann ich aber auch einer Variablen indirekt einen Wert zuordnen indem ich einen Pointer verwende. Nehmen wir den folgenden kleinen Programmausschnitt als Beispiel:

```
int x = 5;
int *iPtr;
...
iPtr = &x;      // iPtr hat als Wert die Adresse von x
*iPtr = 7;      // der Wert von x wird indirekt auf 7 geändert
```

Um diese indirekte Zuweisung zu überprüfen, geben wir einfach den Wert der Adresse mit dem Konversionspezifikator `%p` aus.

Listing 8.1: Beispiel für indirekte Zuweisung

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int x = 1;
6     int *iPtr;
7
8     iPtr = &x;
9     *iPtr = 5;
10
11     printf("\n*iPtr: %p\n&x: %p\nx: %d\n", iPtr, &x, x);
12     return 0;
13 }
```

Hier wird der Konversionsspezifikator `%p` verwendet, um die Speicheradresse des Pointers und der Integer-Variablen (in hexadezimaler Form) auszugeben.

Hier noch ein weiteres Beispiel für die indirekte Zuweisung unter Verwendung von Pointern.

Listing 8.2: Indirekte Zuweisung mit Pointern

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int x = 5;
6     int y = 10;
7     int *iPtr = NULL;
8
9     printf("\niPointer_points_to: %p\n", iPtr);
10
11     // assign memory address of y to pointer
12     iPtr = &y;
13     printf("\niPointer_points_now_to: %p\n", iPtr);
14
15     // change the value of x to the value of y
16     x = *iPtr;
17     printf("\niThe_value_of_x_is_now: %d", x);
18
19     // change the value of y to 15
20     *iPtr = 15;
21     printf("\niThe_value_of_y_is_now: %d\n", y);
22     return 0;

```

8.2. Funktionen und Pointer

Einer der großen Vorteile bei der Verwendung von Pointern als Argument bei Funktionsaufrufen, ist die Übergabe *by reference*. Üblicherweise werden Argumente in C als Wert (*passing by value*) übergeben. Es wird dann eine Kopie des Wertes generiert. Daher werden von der Funktion die Werte der übergebenden Variablen auch nicht geändert, und sie stehen nach Beendigung der Funktion weiterhin zur Verfügung, wie wir im Abschnitt 6.4 gesehen haben. Zur Demonstration des Konzepts des *passing by value* zu demonstrieren, dient das folgende Programm 8.3:

Listing 8.3: Übergabe von Argumenten durch *passing by value*

```

1 /* download passbyvalue.c */
2
3 #include <stdio.h>
4
5 void demoPassByValue(int);
6

```

8. Umgang mit Pointern

```
7 int main()
8 {
9     int x = 0;
10
11     printf("\nEnter a number: ");
12     scanf("%d", &x);
13
14     demoPassByValue(x);
15
16     printf("\nThe original value of x did not change: %d\n", x);
17     return 0;
18 }
19
20 void demoPassByValue(int x)
21 {
22     x += 5;
23     printf("\nThe value of x is: %d", x);
24 }
```

Dieses Programm macht deutlich, dass der Wert der Variablen `x` in der aufrufenden Hauptfunktion `main()` sich nicht ändert, obwohl diese Variable scheinbar an die Funktion `demoPassByValue()` übergeben wurde. Aber es wurde eben nur der Wert von `x`, nicht aber die Variable selber übergeben. Manchmal möchte man aber, dass der Inhalt der übergebenden Variablen in der Funktion verarbeitet und verändert wird. Dieses kann durch die Übergabe *by reference* geschehen. Dabei wird anstelle des Wertes der Variablen ihre Adresse übergeben. Die aufgerufene Funktion kann jetzt lesend und verändernd auf den Inhalt der Variablen `x` zugreifen, wie dieses in der Programm 8.4 gezeigt wird.

Listing 8.4: Übergabe von Argumenten durch *passing by reference*

```
1 /* download passbyreference.c */
2
3 #include <stdio.h>
4
5 void demoPassByReference(int *);
6
7 int main()
8 {
9     int x = 0;
10
11     printf("\nEnter a number: ");
12     scanf("%d", &x);
13
14     printf("\nThe original value of x is: %d\n", x);
15
16     demoPassByReference(&x);
17
18     printf("\nThe new value of x is: %d\n", x);
19     return 0;
20 }
```

```

21
22 void demoPassByReference(int *ptrX)
23 {
24     *ptrX += 5;
25     printf("\nThe value of x is now: %d", *ptrX);
26 }

```

Um Argumente *bei reference* an eine Funktion übergeben zu können, muss bei der Deklaration darauf geachtet werden, dass auch ein Pointer als Argument deklariert wird.

```
void demoPassByReference(int *);
```

Damit weiß der C Compiler, dass ein Pointer auf einen Integer-Wert erwartet wird. Beim Funktionsaufruf muss dann darauf geachtet werden, dass auch ein Pointer auf eine Integer-Variable übergeben wird.

```
demoPassByReference(&x);
```

Der Rest passiert dann in der Funktionsdefinition, die genau schreibt, was denn nun geschehen soll.

```
void demoPassByReference(int *ptr) { ...}
```

Dadurch, dass der Pointer übergeben wird, weiß der Compiler, wo der Wert eingetragen werden soll.

```
*ptrX +=5;
```

Der `*` Operator in der `printf()` Funktion weist die Funktion an, den Inhalt auf den der Pointer zeigt, auszugeben.

```
printf("\nThe value of x is now: %d\n", *ptrX);
```

Damit ist jetzt auch klar, warum die `scanf()` Funktion das `&` Zeichen vor der Variablen benötigt, wo der Wert hingeschrieben werden soll.

8.3. Übergabe von Arrays an Funktionen

In Kapitel 7 haben wir kennengelernt, dass Arrays einen zusammenhängenden Speicherplatz für eine Folge von Werten gleichen Datentyps darstellen. In C sind Pointer und Arrays sehr eng verwandt. Es ist sogar so, dass der Name des Arrays nichts anderes ist, als die Adresse des ersten Datenfeldes in dem Array. Diese Adresse kann genutzt werden, um auf die Werte des Arrays zuzugreifen und sie gegebenenfalls zu verändern. Das nachfolgende Listing 8.5 demonstriert, wie ein Array an eine Funktion übergeben wird, und diese die Werte des Arrays verändert.

Listing 8.5: Verändern der Inhalte eines Arrays durch eine Funktion

```
1 /* download squarennumbers.c */
2 #include <stdio.h>
3
4 void squareNumbers(int [], int);
5
6 int main()
7 {
8     int iNumbers[3] = {2, 4, 6};
9
10    // print the array
11    printf("\nThe current array values are: ");
12    for (int i = 0; i < 3; i++)
13        printf("%d ", iNumbers[i]);
14    printf("\n");
15
16    squareNumbers(iNumbers, 3);
17
18    // print the array
19    printf("\nThe modified array values are: ");
20    for (int i = 0; i < 3; i++)
21        printf("%d ", iNumbers[i]);
22    printf("\n");
23
24    return 0;
25 }
26
27 void squareNumbers(int num[], int nlen)
28 {
29    // the array is passed by reference
30    // the length of the array by value
31    // change the array contents for their squares
32    for (int i = 0; i < nlen; i++)
33        num[i] = num[i] * num[i];
34    return;
35 }
```

8.4. Die const Anweisung

Die Übergabe von Variablen *by reference* bietet eine sehr mächtige Möglichkeit die Werte einer Variablen oder eines Arrays zu ändern. Dahingegen verhindert die Methode der Übergabe *by value* genau dieses. Manchmal möchte man aber gerade, dass die Variablen *by reference* übergeben werden, aber der Wert nicht geändert werden kann. Dieses kann durch die Angabe **const** erreicht werden. Der Compiler prüft dann, ob es eine Zuweisung auf ein Feld oder eine Variable gibt, wodurch sich der Wert verändern kann. Das Konzept wird in dem nachfolgenden Listing 8.6 erläutert.

Listing 8.6: Versuch die Inhalte eines const Arrays zu ändern

```

1  /* download constarray.c */
2  #include <stdio.h>
3
4  void printArray(const int [], int);
5  void modifyArray(const int [], int);
6
7  int main()
8  {
9      int iNumbers[3] = {2, 4, 6};
10
11     printArray(iNumbers, 3);
12
13     modifyArray(iNumbers, 3);
14
15     printArray(iNumbers, 3)
16
17     return 0;
18 }
19
20 void printArray(const int num[], int nlen)
21 {
22     //pass by reference, but read only
23     printf("\nArray contents are:\n");
24     for (int i = 0; i < nlen; i++)
25         printf("\t%d", num[i]);
26     printf("\n");
27     return;
28 }
29
30 void modifyArray(const int num[], int nlen)
31 {
32     //pass by reference, but read only
33
34     for (int i = 0; i < nlen; i++)
35         num[i] = num[i] * num[i];
36     return;
37 }

```

Dieses kleine Programm kann nicht kompiliert werden, da in der Funktion `modifyArray()` das als konstant deklarierte Array verändert wird.

```
error: assignment of read-only location ...
```

```
||= Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) =||
```

Die `const` Deklaration dient also dazu, sicherzustellen, dass nicht versehentlich durch eine Funktion Werte von Variablen oder Arrays geändert werden, wenn die Adressen an Funktionen übergeben werden. Das Konzept ist an der kurzen Funktion `printArray()` dargestellt, welche die Werte eines Arrays nur ausgeben, nicht aber ändern soll.

8.5. Cryptogram

Auch in diesem Kapitel wollen wir die gelernten Prinzipien wieder mit einem kleinen Programm anwenden. Hierbei soll ein lesbarer Text, der sogenannte Klartext, zunächst so verschlüsselt werden, dass er wie eine geheime Nachricht versendet werden kann und keiner versteht was geschrieben ist. Allerdings ist das Programm so einfach, dass es leicht geknackt werden kann. In diesem Fall soll jeder Buchstabe einfach zwei Zeichen weiter geschoben werden. Also aus dem Klartext *Triff mich heute Abend* wird dann *Vtkhh okej jgwvg Cdgpff*. Nicht besonders sicher, aber auf Anhieb nicht lesbar. Natürlich braucht man dann auch die Möglichkeit den Klartext wieder herzustellen. Der Schlüssel ist bei diesem einfachen Programm die Anzahl der Zeichen, um die jedes Zeichen verschoben wurde. In diesem Fall also zwei. In dem Programm kann ein solcher einfacher Schlüssel zufällig generiert werden.

Listing 8.7: Programm um ein Wort zu verschlüsseln und entschlüsseln

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 // function prototypes
6 void encrypt(char [], int);
7 void decrypt(char [], int);
8
9 int main()
10 {
11     char myString[21] = {0};
12     int iSelection = 0;
13     int iRand;
14     srand(time(NULL));
15     iRand = (rand() % 4) + 1; // random number 1 ... 4
16
17     while (iSelection != 4) {
18         printf("\n\n1\tEncrypt\tClear\tText");
19         printf("\n2\tDecrypt\tCypher\tText");
20         printf("\n3\tGenerate\tNew\tKey");
21         printf("\n4\tQuit\n");
22         printf("\nSelect a Cryptography Option:");
23         scanf("%d", &iSelection);
24         switch (iSelection) {
25             case 1:
26                 printf("\nEnter one word as clear text to encrypt:");
27                 scanf("%s", myString);
28                 encrypt(myString, iRand);
29                 break;
30             case 2:
31                 printf("\nEnter cypher text to decrypt:");
32                 scanf("%s", myString);
33                 decrypt(myString, iRand);

```

```

34             break;
35         case 3:
36             iRand = (rand() % 4) + 1;
37             printf("\nNew_Key_Generated\n");
38             break;
39     }
40 }
41 return 0;
42 }
43
44 void encrypt(char sMessage[], int iKey)
45 {
46     int i = 0;
47     while (sMessage[i]) {
48         sMessage[i] += iKey;
49         i++;
50     }
51     i = 0;
52     printf("\nEncrypted_Message_is:");
53     while (sMessage[i]) {
54         printf("%c", sMessage[i]);
55         i++;
56     }
57 }
58
59 void decrypt(char sMessage[], int iKey)
60 {
61     int i = 0;
62     while (sMessage[i]) {
63         sMessage[i] -= iKey;
64         i++;
65     }
66
67     i = 0;
68     printf("\nDecrypted_Message_is:");
69     while (sMessage[i]) {
70         printf("%c", sMessage[i]);
71         i++;
72     }
73 }

```

8.6. Challenges

8.6.1. Variablen und ihre Pointer

Schreibe ein Programm, das die folgenden Operationen durchföhrt:

- Deklariere drei Pointer-Variablen iPtr vom Typ Integer, cPtr vom Typ Character und fPtr vom Typ Fließkommazahl

8. Umgang mit Pointern

- Deklariere drei passende Variablen iNumber vom Typ Integer, cCharacter vom Typ Character und fNumber vom Typ Fließkommazahl
- Weise die Adresse jeder Variablen der entsprechenden Pointer-Variablen zu
- Gebe den Wert jeder Nicht-Pointer-Variablen aus
- Gebe den Wert aus, auf den die jeweilige Pointer-Variable zeigt
- *Unterscheiden sich diese Werte?*
- Gebe die Adresse jeder Nicht-Pointer-Variablen aus
- Gebe die Adresse, die in jeder Pointer-Variablen steht, aus
- *Unterscheiden sich diese Adressen?*

8.6.2. Würfelspiel

Entwickle ein kleines Würfelspiel. Der Spieler kann 6 Würfel auf einmal werfen. Jeder Wurf wird in einem Integer-Array der Länge 6 gespeichert. Das Array wird in der `main()` Funktion angelegt und an die neue Funktion `tossDice()` übergeben, diese weist den 6 Feldern des Array neue zufällige Werte zu. Geben Sie das zufällige Würfelergebnis aus.

8.6.3. Cryptogramm

Modifizieren Sie das Programm so, dass ein Substitutionscode verwendet wird, bei dem jeder Buchstabe durch einen anderen ersetzt wird.

9. Arbeiten mit Strings

9.1. Strings – Einführung

Strings sind eigentlich nichts anderes als Arrays vom Typ `char` und dienen zur Darstellung und Verarbeitung von Zeichenfolgen, sogenannten *Strings*. Diese werden etwa bei der Programmierung von Benutzerschnittstellen sowie Verarbeitung von textuellen Daten oder Textdateien benötigt. Stringkonstanten haben Sie schon kennen gelernt in Form von beispielsweise

```
printf("Ich bin die Stringkonstante und stehe zwischen 2 Hochkommata");
```

Alles was zwischen 2 Hochkommata steht, ist eine Stringkonstante. Dabei kann man Stringkonstanten auf 2 verschieden Arten darstellen.

```
char hallo1[] = {"Hallo Welt\n"};
```

oder

```
char hallo2[]={ 'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '\n', '\0' };
```

Sie sehen, ein String ist eigentlich gar nichts anderes als ein Array von Zeichen. Wenn Sie den Inhalt des Arrays direkt bei der Deklaration übergeben, benötigen sie keinen Indexwert []. C deklariert die Stringvariable automatisch passend, um den Inhalt aufzunehmen.

Allerdings haben Strings noch eine ganz wesentliche Eigenschaft. Jeder String wird mit dem `'\0'`-Zeichen beendet, damit C erkennen kann, wie lang ein String ist. Dieses ist gut in Abb. 9.1 zu erkennen.

Achtung: Immer, wenn Sie eine Stringvariable deklarieren, muss diese groß genug sein, um auch das abschließende `'\0'` Zeichen speichern zu können.

H	a	l	l	o		W	e	l	t	\n	\0
---	---	---	---	---	--	---	---	---	---	----	----

Abbildung 9.1.: Darstellung eines Strings

9. Arbeiten mit Strings

Die Bedeutung des abschließenden `'\0'` Zeichens wird in dem nachfolgenden Listing 9.1 noch einmal erläutert.

Listing 9.1: Bedeutung des abschließenden `\0` Zeichens

```
1 /*Download:string1.c*/
2
3 #include <stdio.h>
4
5 int main()
6 {
7     char hello1[] = {"Hallo_Welt\n"};
8     char output[] = {"Ich_bin_lesbar_\0_Ich_nicht_mehr"};
9     char deznul[] = {"Mich_siehst_du,_0_und_die_Null_auch"};
10
11     printf("%s", hello1);
12     printf("%s\n", output);
13     printf("%s", deznul);
14     return 0;
15 }
```

Der Konversionsspezifikator `%s`. sorgt dafür, dass unsere Strings ausgegeben werden. Jetzt ist auch verständlich, warum jeder String ein Ende-Kennungszeichen benötigt. Denn anstatt, wie bei Arrays von Zahlen, wo mit mit Schleifen oder Indexfeldern auf die einzelnen Werte zugegriffen werden muss, wird der String auf eine Schlag ausgegeben. Dafür muss C wissen, wann Schluss ist. Bei dem String `output`

```
char output[] = {"Ich bin lesbar \0 Ich nicht mehr"};
```

werden korrekterweise nur die Zeichen bis `\0` ausgegeben. Die zweite Hälfte des Strings existiert zwar, aber wird niemals ausgegeben, da zuvor das Zeichen `'\0'` für das Ende des Strings steht.

In der nächsten Anweisung

```
char deznul[] = {"Mich siehst du, 0 und die Null auch"};
```

wird der ganze String ausgegeben, weil das Zeichen `'0'` oder das Wort `Null` nicht gleich dem Ende-Kennungszeichen `'\0'` ist. Natürlich ist es auch möglich, wie bei einem Array auf die einzelnen Zeichen über den Index zuzugreifen.

Das ein weiter Vorteil der Strings: Wenn wir Strings auf ihre Länge überprüfen wollen, etwa mit einer `for`-Schleife benötigen wir lediglich als Abbruchbedingung das Zeichen `'\0'`. So könnte man mit den folgenden Anweisungen die Länge eines Strings bestimmen und ausgeben:

```
for(i=0; hello1[i]!='\0'; i++);
printf("Länge von '%s' = %d Zeichen\n", hello1, i);
```

Hier passiert übrigens alles in der Schleifendefinition. Der Schleifeninhalt besteht nur aus der leeren Anweisung `;`.

9.2. Strings in Zahlen konvertieren

Man muss in C immer zwischen Zeichen und einer Zeichenkette, die aus Ziffern besteht, und Zahlen unterscheiden, die einem Wert entsprechen. Oft werden über die Tastatur Zeichen eingegeben, die aus Ziffern bestehen und einen Zahlenwert repräsentieren sollen. Sollen diese Zeichenketten in C Zahlenwerte repräsentieren, so müssen diese erst umgewandelt werden. Hierzu werden in `stdlib.h` zwei Funktionen deklariert, mit denen Strings in Zahlenwerte gewandelt werden können.

- `atoi()` konvertiert einen String in einen Integerwert
- `atof()` konvertiert einen String in eine Gleitkommazahl

Beide Funktionen nehmen einen String als Argument und geben die gefundene Zahl zurück wie in Listing 9.2 demonstriert wird.

Listing 9.2: Konvertieren eines Strings in eine Zahl

```

1  /* download atoi atof.c */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8      char *str1 = "123.79";
9      char *str2 = "55";
10     float f1;
11     int i1;
12
13     printf("\nString_1_is_%s", str1);
14     printf("\nString_2_is_%s\n", str2);
15
16     f1 = atof(str1);
17     i1 = atoi(str2);
18
19     printf("\nString_1_is_converted_to_a_float: %.2f", f1);
20     printf("\nString_2_is_converted_to_an_integer: %d\n", i1);
21
22     return 0;
23 }
```

Hier ist noch eine Kleinigkeit zu erwähnen: Mit der Anweisung in Zeile 8

```
char *str1 = "123.79";
```

wird kein String deklariert und dann initialisiert, sondern es wird ein Pointer auf einen `char` deklariert und dieser wird dann mit dem der Adresse des ersten Zeichens des konstanten Strings "123.79" initialisiert. Es ist daher nicht möglich, den Wert des konstanten Strings zu verändern. Damit hat diese Anweisung eine andere Bedeutung als die bisher bekannten Möglichkeiten zur Deklaration und Initialisierung eines Strings

9. Arbeiten mit Strings

```
char str[80] = "123.78";
```

oder

```
char str[] = "123.78";
```

Diese reservieren zunächst einen Speicherplatz der Länge von 80 Zeichen beziehungsweise einen zur Aufnahme des anschließenden Initialisierungsstrings ausreichend großen Speicherplatz und initialisieren diesen dann mit dem Wert "123.78". Die Adresse dieses Speicherplatzes ist dann `str` zugeordnet.

9.3. Stringverarbeitung

Wichtige Aufgaben im Zusammenhang mit Strings sind das Kopieren, das Verbinden von Strings, das Vergleichen oder das Kopieren. Dabei werden Sie in diesem Abschnitt einige in C häufig verwendete String-Funktionen kennenlernen. Diese sind zumeist in `string.h` deklariert.

9.3.1. `strlen()` Funktion

Oft benötigt man die Länge eines Strings in Zeichen. Die Funktion `strlen()` nimmt einen String als Argument und gibt die Anzahl von Zeichen in dem String zurück. Dabei wird das abschließende `'\0'` nicht mitgezählt.

```
int iLength;
char name[] = "Peter"

iLength = strlen(name);
// iLength hat jetzt den Wert 5
```

9.3.2. `tolower()` und `toupper()` Funktionen

Die in `ctype.h` deklarierten Funktionen bieten viele Möglichkeiten zum Umgang mit einzelnen Zeichen. Leider gibt es in C keine fertigen Funktionen, um einen String komplett in Großbuchstaben oder Kleinbuchstaben zu wandeln. Dieses kann mit selbst definierten Funktionen geschehen, wie das Listing 9.3 zeigt.

Listing 9.3: Umwandlung von Strings in Groß- oder Kleinbuchstaben

```
1 /* download 9-toupper.c */
2
3 #include <stdio.h>
4 #include <ctype.h>
5 #include <string.h>
6
7 // function prototypes
8 char *convertL(char *);
```

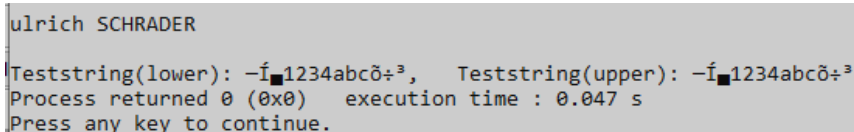


```

9 char *convertU(char *);
10
11 int main()
12 {
13     char strVorname[] = "Ulrich";
14     char strNachname[] = "Schrader";
15     char strTest[] = "ÄÖÜ1234ABCäöü";
16
17     convertL(strVorname);
18     convertU(strNachname);
19
20     printf("\n%s %s\n", strVorname, strNachname);
21     printf("\nTeststring(lower): %s, Teststring(upper): %s",
22           convertL(strTest), convertU(strTest));
23
24     return 0;
25 }
26
27 char *convertL(char *str)
28 {
29     for (int i = 0; i < strlen(str); i++)
30         str[i] = tolower(str[i]);
31     return str;
32 }
33
34 char *convertU(char *str)
35 {
36     for (int i = 0; i < strlen(str); i++)
37         str[i] = toupper(str[i]);
38     return str;
39 }

```

Wie man an dem Vornamen und dem Nachnamen sieht, hat die Konvertierung des Vornamens in Klein- und des Nachnamens in Großbuchstaben geklappt. Allerdings ist es bei dem Teststring mit Umlauten und Zahlen misslungen. Während die Zahlen korrekterweise nicht verändert worden sind, haben die Umlaute Probleme gemacht. Das liegt daran, dass C zunächst nur auf den Zeichensatz ASCII (American Standard Code for Information Interchange) siehe Tab. B.1 aufbaut, der nur Zeichenwerte von 0 bis 127 kennt. Dieser Bereich enthält gar keine Umlaute. Zu finden sind diese erst im nicht stan-



```

ulrich SCHRADER
Teststring(lower): -f_1234abcö÷³, Teststring(upper): -f_1234abcö÷³
Process returned 0 (0x0) execution time : 0.047 s
Press any key to continue.

```

Abbildung 9.2.: Ausgabe des Listing 9.3

9. Arbeiten mit Strings

dardisierten *extended ASCII*-Bereich mit Zeichenwerten von 128 bis 255. Die Funktionen `tolower()` und `toupper()` aber sind aber nur für den standardisierten ASCII-Bereich von einem Zeichenwert 0 bis 127 definiert. Daher müsste man die Funktionen `convertL()` und `convertU()` verändern, um auch die deutschsprachigen Sonderzeichen korrekt umzuwandeln. Wollen Sie sich alle 255 verfügbaren (druckbaren) Zeichen ansehen, so können Sie das mit dem Listing B.1.

9.3.3. `strcpy()` Funktion

Die `strcpy()` Funktion kopiert den Inhalt des einen Strings in einen anderen String. Wie man schon ahnen kann, nimmt diese Funktion zwei Strings als Argument, wie das Listing `reflststrcpy` zeigt. Das zweite Argument ist der Quellstring der kopiert werden soll, das erste Argument der Zielstring in den kopiert werden soll. Dabei muss immer beachtet werden, dass der Zielstring ausreichend groß deklariert ist, damit er den Quellstring einschließlich des String-Endezeichen `'\0'` aufnehmen kann. Die Verwendung der `strcpy()` Funktion wird in Listing 9.4 gezeigt.

Listing 9.4: Kopieren eines Strings

```
1 /* download 9-strcpy.c */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char strZiel[11];
8     char *strQuelle = "C_Language";
9
10    printf("\nstrZiel_enthaltet_jetzt_%s\n\n", strcpy(strZiel, strQuelle));
11    return 0;
12 }
```

Nach erfolgten Kopieren gibt die Funktion `strcpy()` zusätzlich die Adresse des Zielstrings zurück, so dass diese wie in dem Listing, direkt ausgedruckt werden kann.

9.3.4. `strcat()` Funktion

Eine weitere häufig verwendete Funktion ist die `strcat()` Funktion. Diese dient dazu, einen String an einen anderen anzufügen. Ganz gleich wie `strcpy()` nimmt diese Funktion zwei Argumente und gibt die Adresse des Ergebnisstrings zurück wie in Listing 9.5 beschrieben.

Listing 9.5: Anfügen eines Strings an einen anderen String

```
1 /* download 9-strcat.c */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
```

```

6 {
7     char str1[40] = "C_ist_eine_";
8     char str2[] = "tolle_Programmiersprache";
9
10    printf("\n%s\n\n", strcat(str1, str2));
11    return 0;
12 }

```

Dieses Listing zeigt das `str2` an `str1` angefügt wird. Da die Zeichen unmittelbar aufeinander folgen, musste `str1` am Ende noch ein Leerzeichen haben, damit die Worte korrekt voneinander getrennt ausgegeben werden.

9.4. Strings analysieren

In diesem Abschnitt sollen einige Funktionen aufgezeigt werden, die dazu dienen Strings näher zu untersuchen.

9.4.1. strcmp() Funktion

Die `strcmp()` Funktion dient primär dazu zwei Strings auf ihre Gleichheit zu untersuchen. Dabei werden die beiden Strings zeichenweise miteinander verglichen. Gemäss den ASCII-Werten der einzelnen Zeichen können die beiden Strings sogar auch für eine mögliche Sortierung verglichen werden, wie im Listing 9.6 dargestellt.

Listing 9.6: Vergleichen von Strings

```

1 /* download 9-strcmp.c */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char *str1 = "Paul";
8     char *str2 = "Mary";
9     char *str3 = "Peter";
10    char *str4 = "Paul";
11
12    printf("\n\nstr1=_%s", str1);
13    printf("\nstr2=_%s", str2);
14    printf("\nstr3=_%s", str3);
15    printf("\nstr4=_%s\n", str4);
16
17    printf("\nstrcmp(str1,_str2)=_%d", strcmp(str1, str2));
18    printf("\nstrcmp(str1,_str3)=_%d", strcmp(str1, str3));
19    printf("\nstrcmp(str1,_str4)=_%d\n", strcmp(str1, str4));
20
21    return 0;
22 }

```

9. Arbeiten mit Strings

Funktion	Rückgabewert	Beschreibung
<code>strcmp(str1, str2);</code>	0	str1 ist identisch str2
<code>strcmp(str1, str2);</code>	<0	str1 ist kleiner str2
<code>strcmp(str1, str2);</code>	>0	str2 ist größer str2

Tabelle 9.1.: Stringvergleich mit `strcmp()`

Die `strcmp()` Funktion nimmt zwei Strings, die miteinander verglichen werden sollen als Argumente und gibt einen numerischen Wert zurück, der gemäß Tab. 9.1 angibt, ob die Strings identisch oder der ein größer oder kleiner als der andere ist. Dieses kann beispielsweise verwendet werden, datenbankähnliche Funktionalitäten zu realisieren.

9.4.2. `strstr()` Funktion

Die `strstr()` Funktion kann dazu verwendet werden, einen String nach einem anderen zu durchsuchen. Auch hier werden wieder zwei Strings als Argumente übergeben. Das erste Argument ist der String, der durchsucht werden soll. Das zweite Argument stellt den String dar, der gesucht werden soll. Als Rückgabewert wird ein Pointer auf die Stelle im ersten String gegeben, an der sich der gesuchte String befindet. Sollte er nicht gefunden werden, wird der `NULL` Pointer zurückgegeben. Es kann damit also ein langer String nach dem Auftauchen eines bestimmten anderen Strings durchsucht werden, wie in Listing 9.7 gezeigt.

Listing 9.7: Suchen eines Strings in einem anderen String

```
1 /* download 9-strstr.c */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main()
6 {
7     char *str1 = "Peter, Paul, Mary and Puff, the Little Dragon";
8     char *str2 = "Mary";
9     char *str3 = "ittle";
10    char *str4 = "und";
11
12    printf("\n\nString to be searched: %s", str1);
13    printf("\n\nSearching for %s: %s", str2, strstr(str1, str2));
14    printf("\n\nSearching for %s: %s", str3, strstr(str1, str3));
15    printf("\n\nSearching for %s: %s", str4, strstr(str1, str4));
16
17    return 0;
18 }
```

9.5. Beispielprogramm: Find Word

Find Word ist ein einfaches Programm, dass die in diesem Kapitel besprochenen String-funktionen und -eigenschaften aufgreift und umsetzt. Darüber hinaus werden auch die im Abschnitt 7 besprochenen Techniken angewandt. Das Ziel des Spiels besteht darin innerhalb einer kurzen Zeit ein Wort in einer scheinbar sinnlosen Buchstabensuppe zu finden.

Listing 9.8: Wortsuch-Spiel

```

1  /* download findword.c */
2
3  #include <stdio.h>
4  #include <string.h>
5  #include <time.h>
6  #include <stdlib.h>
7  #include <ctype.h>
8
9  #ifdef __UNIX__
10 #define CLEARSCREEN      system("clear")
11 #else
12 #define CLEARSCREEN      system("cls")
13 #endif // __UNIX__ defined
14
15 // function prototypes
16 void checkAnswer(char *, char []);
17
18 int main()
19 {
20     char *strGame[5] = {"ADELANGUAGEEFERVZOPIBMOU",
21                         "ZBPOINTERSKMLLOOPMNOTZUL",
22                         "PODSTRINGGDIWHIEEICERLUT",
23                         "YVCPROGRAMMERQWKNUITMDAT",
24                         "UKUNIXFIMWXIZEQZINPUTTEXT"};
25     char answer[80] = {0};
26     int displayed = 0;
27
28
29     int startTime = 0;
30     CLEARSCREEN;
31
32     printf("\n\nFind Word");
33     startTime = time(NULL);
34
35     for ( int i = 0; i < 5; i++) {
36         /* display text for a few seconds */
37         while (startTime + 3 > time(NULL)) {
38             if ( displayed == 0) {
39                 printf("\nFind a word in: \n\n");
40                 printf("%s\n\n", strGame[i]);

```

9. Arbeiten mit Strings

```
41         displayed = 1;
42     }
43 }
44
45     CLEARSCREEN;
46     printf("\nEnter word found: ");
47
48     scanf("%s", &answer);
49     checkAnswer(strGame[i], answer);
50
51     displayed = 0;
52     startTime= time(NULL);
53 }
54 return 0;
55 }
56
57 void checkAnswer(char *string1, char string2[])
58 {
59     // convert answer to upper case to perform the comparision
60     for ( int i = 0; i <= strlen(string2); i++)
61         string2[i] = toupper(string2[i]);
62
63     if (strstr( string1, string2) != 0 && string2[0] != 0)
64         printf("\nGreat job");
65     else
66         printf("\nSorry, you did not find the word");
67
68 }
```

9.6. Challenges

9.6.1. In Großbuchstaben umwandeln (1)

Schreiben sie ein Programm wie oben, dass den nachfolgenden Text in Großbuchstaben ausgibt. Schreiben Sie dafür eine eigene Funktion.

Diese Zeichenkette soll in Grossbuchstaben ausgegeben werden.

9.6.2. In Großbuchstaben umwandeln (2)

Schreiben sie ein Programm wie oben, dass den nachfolgenden Text in Großbuchstaben ausgibt. Schreiben Sie dafür eine eigene Funktion. Verwenden Sie nicht die Headerdatei `ctype.h`.

Diese Zeichenkette soll in Grossbuchstaben ausgegeben werden. Verwenden Sie nicht ctype.h

Tipp: Sehen Sie sich die Buchstabenwerte in der ASCII-Tabelle B.1 näher an.

9.6.3. Alphabetische Sortierung

Schreiben Sie ein Programm, das einen Array von Strings verwendet, um die folgenden Worte und Namen zu speichern:

- Peter
- Paul
- Mary
- Puff
- Magic
- Dragon

Schreiben Sie dann eine kleine Funktion, die mittels `strcmp()` die Worte alphabetisch sortiert ausgibt.

10. Dynamische Speicherverwaltung

10.1. Grundlagen

Immer wenn eine Variable oder gar ein Array deklariert wird, reserviert der Compiler ausreichend Speicherplatz entsprechend der Größe der Variablen. Dieses geschieht unabhängig davon, ob im Verlauf des Programms der Speicherplatz auch wirklich benötigt wird. Sollte es sich um kleine Arrays, einige Float oder Integer-Werte handeln, so ist das unproblematisch. Soll aber beispielsweise ein Programm zur Verwaltung von Bildern entwickelt werden, in dem auch die Bilddaten selber gespeichert werden, so kann es schon entscheidend sein, ob Speicherplatz für 500 oder gar 50.000 Bilder vorgehalten wird. Je nach verfügbaren Speicherplatz kann es dann schon eng werden. Insbesondere, wenn Anwendungen an Rechner mit nur geringem Hauptspeicher angepasst werden müssen, ist es oft notwendig, dafür zu sorgen, dass nur der momentan wirklich benötigte Speicher reserviert wird. Das bedeutet dann, dass der Speicher zur Programmlaufzeit dynamisch verwaltet werden muss. Bisher haben wir nur über unsere deklarierten Variablen mit statischen Speicherbereichen gearbeitet. Die von Programmen dynamischen benötigten Speicherbereiche werden aus dem sogenannten *Heap* gewonnen. Ein Heap ist ein großer zusammenhängender Speicherbereich, der allen Programmen zur Laufzeit als Speicherlieferant dient.

10.2. malloc() Funktion

Die `malloc()` Funktion ermöglicht es, während der Laufzeit unseres Programms für Variablen Speicherplatz zu reservieren (allozieren).

Bei Aufruf dieser Funktion wird auf dem Heap ein Hauptspeicherbereich der Größe `size Bytes` und liefert einen Zeiger auf das erste Byte des Speicherblocks.

Listing 10.1: Einfache Anwendung von `malloc()`

```
1 /* 10-malloc1.c */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     int *pInt;
9
10    pInt = (int *) malloc (sizeof(int));
11
```

```
12     if(pInt != NULL) {
13         *pInt = 99;
14         printf("\nAdresse_□pInt:□%x\nWert_□in_□pInt:□%d", pInt, *pInt);
15     }
16     else
17         fprintf(stderr, "Kein_□Speicherplatz_□vorhanden!!!\n");
18     return 0;
19 }
```

Erst haben wir einen int-Zeiger deklariert. Anschließend haben wir diesem Pointer durch

```
pInt = (int *) malloc(sizeof(int));
```

einen Speicherbereich der Größe eines Integers zugewiesen. Bei Erfolg zeigt `pInt` auf das 1. Byte des so reservierten (allozierten) Speichers. Die Funktion `malloc()` ist in `stdlib.h` deklariert. Bei Misserfolg hat der Pointer den Wert `NULL` und es wird eine entsprechende Fehlermeldung ausgegeben, dass kein Speicherplatz alloziert werden konnte. In diesem Beispiel wird die Ausgabe auf dem Gerät oder in der Datei festgehalten, auf die `stderr` verweist. Dieses ist in der Regel die Konsole. Allerdings kann es auch auf andere Geräte oder Dateien umgeleitet werden. Die Funktion `fprintf()` funktioniert dabei ganz identisch, wie die bereits genutzte `printf()` Funktion.

10.2.1. NULL-Pointer

Der `NULL`-Pointer wird zurückgeliefert, wenn `malloc()` nicht mehr genügend zusammenhängenden Speicher finden kann. Er ist ein vordefinierter Pointer, dessen Wert sich von regulären Pointern unterscheidet. Man verwendet den `NULL`-Pointer vorwiegend bei Funktionen, die einen Pointer als Rückgabewert liefern sollen, um einen Fehler anzuzeigen. Bei einem Aufruf von `malloc()` muss die Anzahl der zu allozierenden Bytes angegeben werden. Damit ist die Größe des Objekts gemeint, das durch den Zeiger referenziert werden soll. Sie können die Größe auch mit numerischen Werten selbst bestimmen. Insgesamt gibt es drei Arten, wie die Größe des benötigten Speicherplatzes angefordert werden kann:

10.2.2. Bestimmen des Speicherbedarfs

1. Numerische Konstante

```
pInt = (int *) malloc(sizeof(2));
```

Hier sind also 2 Bytes für einen int-Pointer alloziert worden. Aber was machen Sie, wenn Sie das Programm auf ein anderes System portieren wollen? Daher ist es oft besser die Größe des Zielobjektes vom Compilers selbst bestimmen zu lassen, wie wir es im Listing 10.1 gemacht haben.

2. Die Angabe des Typs der Variablen in `sizeof`

```
pInt = (int *) malloc(sizeof(int));
```

Diese Möglichkeit hat nur einen Nachteil. Was wenn Sie etwa anstatt eines Integers auf einmal Float-Werte wollen, dann müssen Sie mühsam alle Speicherzuweisungen umändern.

```
pFloat = (double *) malloc(sizeof(double));
```

3. Ebenfalls ist es möglich, den dereferenzierten Pointer für den **sizeof**-Operator zu verwenden, um die Anzahl der zu allozierenden Bytes zu bestimmen.

```
p = (double *) malloc(sizeof(*p));
```

10.2.3. Speicher freigeben

Nun wenn wir Speicher vom Heap holen, sollten wir ihn auch wieder zurückgeben können, wenn dieser nicht mehr benötigt wird. Dazu die die Funktion **free()**. Der Speicher wird übrigens auch freigegeben ohne **free()**, wenn das Programm beendet wird. Daher dient **free()** dazu, auch während des Programmablaufes wieder Speicher zurückzugeben. Ganz im Sinne einer wirklich dynamischen Speicherverwaltung, die nur solange (umfangreichen) Speicherplatz anfordert, wie er auch benötigt wird.

```
void free(void *p)
```

Die Verwendung von **free()** wird in Listing 10.2 gezeigt. Dabei ergibt sich aber ein potentielles Problem:

Listing 10.2: Anwendung von free()

```
1 /* 10-free.c */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     int *pInt;
9
10    pInt = (int *)malloc(10 * sizeof(int));
11
12    if (pInt != NULL) {
13        for (int i = 0; i < 10; i++) {
14            pInt[i] = 100 + i;
15        }
16    }
17    else
18        fprintf(stderr, "Kein_Speicherplatz_vorhanden!!!\n");
19
20    printf("\nvor_free()");
```

10. Dynamische Speicherverwaltung

```
21     for (int i = 0; i < 10; i++) {
22         printf("\n%d", pInt[i]);
23     }
24
25     free(pInt); // Speicher zurueckgeben
26
27     printf("\n\nnach_free()\n");
28
29     for (int i = 0; i < 10; i++) {
30         printf("\n%d", pInt[i]);
31     }
32     return 0;
33 }
```

Wie Sie sehen, wird hier der Speicherplatz für ein Array mit 10 Integer-Elementen alloziert. Ein Problem ergibt sich, da der Pointer auch nach dem Aufruf von `free()` immer noch auf den ursprünglichen Speicherplatz verweist. Es ist daher gute Praxis auch den Pointer auf `NULL` zu setzen.

```
free(pInt);
pInt = NULL;
```

Wenn jetzt erneut (fehlerhaft) auf den Speicherplatz über den Pointer zugegriffen wird, erscheint ein entsprechender Fehler. Diese Anweisungen könnte man auch gut in einer Präprozessor-Anweisung umsetzen:

```
#define FREE(x)  free(x); x=NULL
```

Falle: Falls Sie einen Speicher freigeben, den sie nicht zuvor mit `malloc()`, `calloc()` oder `realloc()` alloziert haben, kann dies katastrophale Folgen haben, da damit die ganze Speicherverwaltung aus dem Tritt gebracht werden könnte. Also achten sie darauf, dass wirklich nur Speicherplatz freigeben, den sie auch angefordert haben.

Was `malloc()` und die weiteren Funktionen für Speicherallokationen so besonders und wichtig macht, ist die einfache Möglichkeit von jedem X-Beliebigen Datentypen Speicher anzufordern - sind es nun einfache Datentypen wie Strings, Arrays oder komplexe Strukturen. Dabei sollte natürlich wie in Listing 10.3 immer darauf geachtet werden, nur soviel Speicher zu allozieren, wie auch tatsächlich benötigt wird.

Listing 10.3: Beispiel für bedarfsgerechtes Speicherallozieren

```
1 /* 10-malloc2.x */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 int main()
```

```

8 {
9     char *string;
10    string = (char *) malloc(sizeof("HALLO_dynamische_WELT\n"));
11
12    if(string == NULL)    {
13        fprintf(stderr, "\nKonnte_keinen_Speicher_reservieren\n");
14        exit(0); // beende das Programm
15    }
16
17    strcpy(string, "HALLO_dynamische_WELT\n");
18    printf("\n%s", string);
19    return 0;
20 }

```

Hinweis: Ein gerne gemachter Fehler in Verbindung mit Funktionen wie `malloc()` ist, dass nicht der Rückgabewert überprüft wird. Es ist nicht immer gesagt, dass auch wirklich Speicherplatz reserviert wurde. Alles was schief gehen KANN, wird auch mal schief gehen! Also vergessen Sie niemals, wenn sie Speicher allozieren, den Rückgabewert zu überprüfen, ob die allozierung erfolgreich verlief.

10.3. `calloc()` und `realloc()` Funktionen

Kommen wir zu weiteren Funktionen zum dynamischen Anfordern von Speicherplatz. Neben der Funktion `malloc()` sind in der Headerdatei `stdlib.h` noch folgende 2 Funktionen dafür vorhanden.

```

void *calloc(size_t anzahl, size_t groesse);
void *realloc(void *zgr, size_t neuegroesse);

```

Die Funktion `calloc()` ist der Funktion `malloc()` ähnlich. Nur das es bei der Funktion `calloc()` nicht einen, sondern 2 Parameter gibt. Im Gegensatz zu `malloc()` können sie bei `calloc()` noch die Anzahl von Datenobjekten angeben, die sie allozieren wollen.

Als Beispiel allozieren wir für 100 Datenobjekte vom Typ 'int' Speicherplatz.

```

int *zahlen;
zahlen=(int *)calloc(100,sizeof(int));

```

Der zweite Unterschied zu `malloc()` ist, dass der allozierte Speicherbereich bei `calloc()` automatisch mit 0 initialisiert wird . Bei `malloc()` besitzt der reservierte Speicherplatz zu Beginn einen undefinierten Wert. Ansonsten funktionieren die beiden Funktionen im Prinzip identisch. `calloc()` gewinnt seine Bedeutung, wenn etwa Speicherplatz für Arrays von komplexen Strukturen alloziert werden soll.

Interessanter ist da schon die Speicherreservierung, die uns die Funktion `realloc()` bietet. Damit ist es nun wirklich möglich im laufenden Programm soviel Speicher zu reservieren wie wir benötigen und das gilt insbesondere für Arrays.

10. Dynamische Speicherverwaltung

Mit `realloc()` sind wir nun in der Lage wirklich dynamische Arrays zu programmieren. Die Anfangsadresse von unserem dynamisch zu vergrößerndem Array ist die, auf der unser Pointer `zgr` zeigt. Der Parameter 'neuegroesse' dient dazu, einen bereits zuvor allozierten Speicherplatz auf `neuegroesse` Bytes zu vergrößern. Apropos, mit `realloc()` ist es auch möglich, den Speicherplatz zu verkleinern. Dabei wird einfach der hintere Teil freigegeben, während der vordere Teil unverändert bleibt. Bei einer Vergrößerung des Speicherplatzes mit `realloc()` behält der vordere Teil auf jeden Fall seinen Wert und der hintere Teil wird einfach hinten angehängt. Dieser angehängte Wert ist aber wie bei `malloc()` undefiniert und nicht wie bei `calloc()` mit 0 initialisiert. Als Beispiel, wie man mit `realloc()` bedarfsgerecht Arrays wachsen lässt, die Listing 10.4.

Listing 10.4: Dynamisches Wachsen eines Arrays mit `realloc()`

```
1 /* 10-realloc.c */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     int n = 0;
9     int max = 5;
10    int z;
11    int i;
12    int *zahlen = NULL;
13
14    // Wir reservieren Speicher f r max int Werte mit calloc
15    if((zahlen=(int *)calloc(max, sizeof(int))) == NULL) {
16        fprintf(stderr, ".....Speichplatzmangel!!!!\n");
17        exit(0);
18    }
19    printf("Zahlen eingeben --- Beenden mit 0\n");
20
21    while (1) {
22        printf("Zahl (%d) eingeben: ", n+1);
23        scanf("%d", &z);
24        if (z == 0)
25            break;
26
27        /*Reservierung whrend der Laufzeit des Programms mit realloc*/
28        if (n >= max) {
29            max += max;
30            if ((zahlen = (int *) realloc (zahlen, max * sizeof(int))) == NU
31                fprintf(stderr, ".....Speicherplatzmangel!!!\n");
32                exit(1);
33            }
34            printf("\n Speicherplatz reserviert (%d Bytes)\n", sizeof(int) *
35        }
36
```

```

37     zahlen[n++] = z;
38 }
39
40 printf("Folgende Zahlen wurden eingegeben ->\n\n");
41 for(i=0; i<n; i++)
42     printf("%d ", zahlen[i]);
43
44 free(zahlen);
45 return 0;
46 }

```

Sicherlich hätten wir unseren Speicher auch einzeln für jeden Wert allokieren können. Allerdings muss die Funktion `realloc()` bei jedem Aufruf den kompletten alten Speicherbereich in einen größeren Speicherbereich umkopieren. In Listing 10.4 wird der Speicherplatz nach jedem erneuten Allokieren mit `realloc()` gleich verdoppelt (`max+=max`), was auch nicht immer ideal ist. Es kommt eben auf Ihr Programm an, wie viele Datenobjekte eines Typs Sie benötigen.

10.4. Challenges

10.4.1. Lieblingsfilm

Schreiben Sie ein Programm, das mittels `malloc()` Speicherplatz alloziert, um maximal 80 Zeichen aufzunehmen. Fragen Sie nach dem Lieblingsfilm des Nutzers unter Verwendung der Funktion `scanf()` und speichern Sie diesen in dem allozierten Speicher. Geben Sie dann den Film über die Konsole wieder aus.

10.4.2. Länge des Namens

Nutzen Sie die `calloc()` Funktion, und lesen Sie einen Namen in den allozierten Speicher. Schreiben Sie eine Schleife, die über diesen Speicherbereich läuft und die Anzahl der Zeichen im Namen zählt. Die Schleife sollte anhalten, wenn ein Speicherelement angehtroffen wird, dass nicht beim Lesen und Speichern des Namens verwendet wurde. Geben Sie die Anzahl der Buchstaben des Namens aus.

10.4.3. String aneinanderfügen

Schreiben Sie ein Programm, dass nacheinander zwei Strings `string1` und `string2` einliest und in allozierten passgenauen Speicherbereichen ablegt. Fügen Sie dann `string2` an `string1` an, so dass der Inhalt beider Strings in `string1` steht. Geben Sie `String1` aus.

11. Umgang mit Dateien

11.1. Einführung

Die bisher geschriebenen Programme haben Daten berechnet und die Ergebnisse auf der Konsole ausgegeben, oder haben Daten erhalten, die wir über die Tastatur eingegeben haben. In manchen Fällen ist das ausreichend. Verlangt ein Programm, das öfter aufgerufen wird aber eine Vielzahl an Eingabedaten, dann ist es mühsam diese immer wieder einzugeben. Gleiches gilt für die Ausgabe an der Konsole. Diese ist in der Regel flüchtig und muss wiederholt werden, falls man das Ergebnis zu einem späteren Zeitpunkt wieder benötigt. Das bedeutet, dass unsere Programme noch sehr eingeschränkt sind. Die meisten Programme benötigen, daher auch die Möglichkeit Daten aus Dateien zu lesen oder in Dateien zu schreiben. Hierzu bietet C eine Reihe von Funktionen in `stdio.h` an.

11.2. Binärdateien und Textdateien

C unterscheidet den Zugriff auf Dateien in zwei grundsätzliche Arten:

1. Textmodus
2. Binärmodus

Im Textmodus werden die Daten als Zeilen von Zeichen gespeichert, die jeweils mit dem Zeichen für neue Zeile `\n` bei UNIX-Systemen und mit den Zeichen für Wagenrücklauf `\r` unmittelbar gefolgt von `\n` bei Windows-basierten Systemen abgeschlossen werden. Dabei ist jedes Zeichen genau ein Byte groß. Die Zahl `123456` belegt also einen Speicherplatz von genau 6 Bytes, wenn sie im Textmodus in der Datei gespeichert wird.

Im Binärmodus werden die Daten in derselben Art und Weise abgespeichert, wie sie im Hauptspeicher des Rechners vorhanden sind. So kann dann die Zahl `123456` mit nur vier Byte gespeichert werden. Neben dem verringerten Speicherbedarf bringt eine solche Speicherung noch den Vorteil, dass beim Einlesen keine Konvertierung stattfinden muss, da die vier Byte direkt einer Integer-Variablen zugeordnet werden können. Im Textmodus müsste die Zeichenfolge erst in eine Zahl konvertiert werden. Dateien im Textmodus und im Binärmodus halten immer die Länge der Datei und ihr Dateiende fest. Hinzu kommt, dass Dateien im Textmodus zusätzlich noch ein Dateiendezeichen mit dem ASCII-Wert 26 am Ende haben. Das ist wichtig zu beachten, da alles Schreib- und Leseaktionen aufhören, wenn auf dieses Zeichen getroffen wird. Ein weiterer Vorteil von Textdateien besteht in ihrer Portabilität. So können Textdateien auf allen Systemen gelesen werden, da sie auf einem einheitlichen Zeichensatz basieren. Bei Binärdateien ist das nicht unbedingt der

11. Umgang mit Dateien

Fall, da hier die Länge der Datentypen in Bytes von einem System zum anderen variieren kann.

11.3. Dateipuffer

Da das Schreiben und Lesen in und von Dateien ein relativ langsamer Prozess ist, verglichen mit dem Schreiben und Lesen des Hauptspeichers, benutzen die Standardprozesse für die Ein- und Ausgabe einen sogenannten Pufferspeicher, der die Daten temporär aufnimmt und diese schnell zur Verfügung stellt.

Der Pufferspeicher oder kurz Puffer ist ein Speicherbereich, der Daten temporär speichert bevor diese in die Datei geschrieben werden. Wenn der Puffer voll ist, werden diese Daten automatisch in die Datei geschrieben (flush). Gleiches gilt, wenn der Zugriff auf die Datei geschlossen wird. In der Regel geschieht dieses automatisch, aber es ist manchmal erforderlich den Puffer eigenständig in die Datei zu schreiben. Dieses geschieht mit der Funktion `fflush()`.

11.4. Grundlegende Dateifunktionen

11.4.1. Öffnen einer Datei mit `fopen()`

Bevor Input- oder Outputoperationen auf eine Datei erfolgen können, muss diese Datei mit `fopen()` erst geöffnet werden.

```
FILE *fp;  
fp = fopen("textdatei.txt", "r");
```

Hier öffnen wir nun die Textdatei `textdatei.txt` auf die nun unser FILE-Pointer `fp` zeigt und verbinden damit einen sogenannten Stream. Als String, der die zu öffnende Datei benennt, ist jeder zulässige Pfadname erlaubt. Also auch Laufwerksangaben oder Verzeichnispfade. So könnte es zum Beispiel auch heißen:

```
FILE *fp;  
fp = fopen("/home/Texte/textdatei.txt", "r");
```

Das Argument `"r"` steht für den Modus, wie auf diesen Stream zugegriffen werden soll. In unserem Beispiel steht `"r"` für read, also nur lesen in diesem Falle.

Der FILE-Zeiger, oder man spricht auch vom FILE-Stream, ist eine Struktur, die in der Headerdatei `stdio.h` vordefiniert ist. Dieser beinhaltet alle Informationen, die wir für die höheren Dateizugriffsfunktionen benötigen.

- Puffer : Anfangsadresse, aktueller Zeiger, Größe
- Filedeskriptor (wird von low-level Dateifunktionen benötigt)
- Position von Schreib-und/oder-Lesezeiger

- Fehler- und End-of-File-Flags

Falls `fopen()` die Datei erfolgreich öffnen konnte, wird ein Pointer auf die File-Struktur (was das ist sehen wir in einem folgenden Kapitel) zurückgegeben. Dieser ist `NULL`, wenn ein Fehler aufgetreten ist. Meistens konnte die angegebene Datei nicht gefunden werden.

Mögliche Werte für den Modus mit dem eine Datei geöffnet werden kann sind:

- `"w"` - (write) - Dieser Modus dient dazu Daten in die Datei zu schreiben. Falls sie nicht existiert, wird sie neu angelegt. Wenn sie bereits existiert, werden allen bereits vorhandenen Daten gelöscht.
- `"a"` - (append) - In diesem Modus werden Daten am Ende der Datei angefügt (append). Wenn die Datei nicht existiert, wird sie neu angelegt. Ansonsten werden neue Daten am Ende angefügt.
- `"r"` - (read) - Dieser Modus erlaubt lediglich das Lesen der Daten der Datei. Sie muss daher bereits existieren. Bereits bestehende Daten können nicht verändert werden.
- `"w+"` - (write + read) - Der Modus ist genauso wie `"w"`, aber zusätzlich ist es möglich, die Daten zu lesen. Wenn die Datei nicht existiert, wird sie neu angelegt. Wenn die Datei bereits existiert, dann werden bereits bestehende Daten gelöscht, bevor neue Daten geschrieben werden.
- `"r+"` - (read and write) - Dieser Modus ist wie der `"r"`-Modus, aber es kann dennoch der Inhalt der Datei verändert werden. Die Datei muss allerdings bereits existieren. Die Daten können modifiziert werden, aber bereits existierende Daten werden nicht gelöscht. Dieser Modus wird dabei auch Update-Modus genannt.
- `"a+"` - (append + read) - Dieser Modus ist wie der `"a"`-Modus, aber man kann auch Daten der Datei lesen. Wenn die Datei nicht existiert, wird eine neue Datei angelegt. Wenn die Datei bereits existiert, dann werden neue Daten am Ende der Datei angefügt. In diesem Modus können Daten angefügt werden, aber bestehende nicht modifiziert werden.

Um eine Datei im Binärmodus zu öffnen, wird zusätzlich ein `"b"` angefügt, wie etwa in `"wb"` oder `"a+b"` oder auch `"ab+"`. Das `"t"` wird meist nicht spezifiziert, da dieses der Standardwert ist.

Sollen mehrere Dateien gleichzeitig geöffnet werden, so muss für jede Datei ein eigenständiger FILE-Pointer vorhanden sein.

Da es eine Reihe von Fehlermöglichkeiten beim Öffnen von Dateien gibt, sollte immer überprüft werden, ob das Öffnen auch erfolgreich war.

```
// Checking for errors
```

```
File *fp;
fp = fopen("somefile.txt", "r");
```

11. Umgang mit Dateien

```
if(fp == NULL)
{
    // if error opening the file show error message
    // and exit the program
    printf("Error opening a file");
    exit(1);
}
```

11.4.2. Schließen einer Datei mit `fclose()`

Wenn man eine Datei öffnet sollte diese auch wieder geschlossen werden. Hierzu dient die Funktion `fclose()`. Obwohl beim Beenden eines Programms alle geöffneten Dateien automatisch geschlossen werden, ist es gute Praxis Dateien zu explizit zu schließen, wenn diese nicht mehr benötigt werden. Damit werden dann auch im Puffer vorhandene Daten noch in die Datei geschrieben. Wenn das Schließen erfolgreich war, gibt `fclose()` eine 0 zurück, ansonsten ist der Rückgabewert EOF. Dieser Wert ist in `stdio.h` definiert und ist -1.

```
if (fclose(fp) != 0) {
    printf("Error closing file");
    exit(1);
}
```

Sollten mehrere Dateien geöffnet sein, so können diese auch pauschal mit `fcloseall()` geschlossen werden.

Damit kann der grundsätzliche Aufbau zur Arbeit mit einer Datei wie folgt beschrieben werden:

```
// Basic workflow of a file program in C

int main()
{
    FILE *fp; // declare file pointer variable
    fp = fopen("somefile.txt", "w"); // fopen() function called

    /*
        do something here
    */

    fclose(fp); // close the file
}
```

sectionLesen und Schreiben von Daten

Nehmen wir an, wir haben bereits eine Text-Datei geöffnet und der Datei-Pointer `fp` zeigt auf die zugehörige Dateistruktur, dann gibt es eine Reihe von Funktionen, die alle mit dem Buchstaben `f` beginnen, um den Dateibezug (file) kenntlich zu machen. Wir werden in diesem Skript nicht auf die wichtige Verwendung von Binärdateien eingehen und somit etwa die Funktionen `fwrite()` und `fread()` nicht näher beschreiben.

11.4.3. `fgets()`

Die Funktion `fgets()` zum Lesen eines Strings kennen wir bereits. Nur haben wir mit `stdin` einen speziellen Stream angesprochen und den Tastaturpuffer ausgelesen.

```
char *fgets(char *str, int n, FILE *fp);
```

Die Funktion `fgets()` liest einen String aus der Datei auf die `fp` zeigt. Die Funktion liest Zeichen solange bis entweder ein `'\n'` Zeichen gelesen wird oder `n-1` Zeichen gelesen wurden oder das Ende der Datei erreicht wurde. Danach wird `'\0'` an den String angefügt, um ihn korrekt abzuschließen. Wurde der Lesekorrek abgeschlossen, so gibt die Funktion einen Pointer auf den String zurück, ansonsten wird `NULL` zurückgegeben. Das Listing 11.1 gibt ein Beispiel für die Verwendung.

Listing 11.1: Verwendung von `fgets()`

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main()
5 {
6     char str[80];
7     FILE *fp;
8     fp = fopen("myfile2.txt", "r");
9
10    if(fp == NULL)
11    {
12        printf("Error opening file\n");
13        exit(1);
14    }
15
16    printf("Testing fgets() function:\n\n");
17    printf("Reading contents of myfile.txt:\n\n");
18
19    while( fgets(str, 80, fp) != NULL )
20    {
21        puts(str);
22    }
23
24    fclose(fp);
25
26    return 0;
27 }
```

11. Umgang mit Dateien

11.4.4. fputs()

Das Analogon zu `fgets()` ist die Funktion `fputs()`.

```
int fputs(const char *str, FILE *fp);
```

Diese Funktion schreibt einen String `str` in die Datei mit dem File-Pointer `fp`. Dabei wird das Null-Zeichen am Ende des Strings nicht geschrieben. War der Vorgang erfolgreich, so wird der Wert 0 zurückgegeben, ansonsten der Wert -1. Das nachfolgende Listing 11.2 erläutert die Verwendung.

Listing 11.2: Verwendung von `fputs()`

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main()
5 {
6     char str[50];
7     FILE *fp;
8     fp = fopen("myfile2.txt", "w");
9
10    if(fp == NULL)
11    {
12        printf("Error opening file\n");
13        exit(1);
14    }
15
16    printf("Testing fputs() function:\n\n");
17    printf("To stop reading press Ctrl+Z in windows and Ctrl+D in Linux:");
18
19    while( fgets(str, 50, stdin) != NULL )
20    {
21        fputs(str, fp);
22    }
23
24    fclose(fp);
25    return 0;
26 }
```

Wie Sie gesehen haben, kann auch die Funktion `puts()` verwendet werden, um einen String an die Konsole auszugeben. Ein wichtiger Unterschied zwischen `fputs()` und `puts()` besteht darin, dass `puts()` das `'\0'` Zeichen am Ende des Strings automatisch in ein `'\n'` umwandelt. Dieses macht `fputs()` nicht.

11.4.5. fprintf()

Mit den vorangegangenen Funktionen können wir Strings in Dateien schreiben und diese wieder auslesen. Allerdings gibt es noch viel mehr Datentypen und auch diese wollen wir in eine Textdatei schreiben oder aus einer solchen lesen. Dabei greifen wir mit `fprintf()`

auf eine Funktion zurück, die wir im Prinzip schon von der Funktion `printf()` her kennen.

```
int fprintf(FILE *fp, const char *format [, argument, ...] );
```

Dabei ist als erstes Argument der Funktion der Pointer auf die Datei, in die geschrieben werden soll, hinzugekommen. Ansonsten bleibt alles beim bereits bekannten. Wenn die Funktion korrekt gelaufen ist, gibt sie die Anzahl der geschriebenen Zeichen zurück, ansonsten den Wert EOF, der -1 entspricht. Ein kurzes Beispielprogramm zur Nutzung von `fprintf()` stellt Listing 11.3 dar.

Listing 11.3: Verwendung von `fprintf()`

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int myFlushStdin();
5
6 int main()
7 {
8     FILE *fp;
9     char name[50];
10    int roll_no, chars, i, n;
11    float marks;
12
13    fp = fopen("records.txt", "w");
14
15    if(fp == NULL)
16    {
17        printf("Error opening file\n");
18        exit(1);
19    }
20
21    printf("Testing fprintf() function:\n\n");
22
23    printf("Enter the number of records you want to enter:");
24    scanf("%d", &n);
25
26
27    for(i = 0; i < n; i++)
28    {
29        printf("\nEnter the details of student %d\n\n", i + 1);
30        // fflush(stdin) does not work for some compilers
31        myFlushStdin();
32
33        printf("Enter name of the student:");
34        fgets(name, 50, stdin);
35
36        printf("Enter roll no:");
37        scanf("%d", &roll_no);
```

11. Umgang mit Dateien

```
38
39     printf("Enter marks:");
40     scanf("%f", &marks);
41
42     chars = fprintf(fp, "%s%d%.2f\n", name, roll_no, marks);
43     printf("\n%d characters successfully written to the file\n\n", chars)
44 }
45
46 fclose(fp);
47 return 0;
48 }
49
50 int myFlushStdin()
51 {
52     int i;
53     while (getchar() != '\n')
54         i++;
55 }
```

11.4.6. fscanf()

Ebenfalls eine alte Bekannte ist die `fscanf()` Funktion, die formatierten Input aus einer Datei liest und Variablen unterschiedlicher Datentypen zuweist. Daher benötigt sie als zusätzlichen Parameter den Pointer auf die auszulesende Datei.

```
int fscanf(FILE *fp, const char *format [, argument, ...] );
```

Wenn die Funktion korrekt gelaufen ist, gibt sie die Anzahl der korrekt gelesenen Werte zurück, ansonsten den Wert EOF, der -1 entspricht. Die Verwendung wird in dem nachfolgenden Listing 11.4 beispielhaft dargestellt.

Listing 11.4: Verwendung von `fscanf()`

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main()
5 {
6     FILE *fp;
7     char strFirstName[50];
8     char strLastName[50];
9     int roll_no, chars;
10    float marks;
11
12    fp = fopen("records.txt", "r");
13
14    if(fp == NULL)
15    {
16        printf("Error opening file\n");
```


Wert	Beschreibung
SEEK_SET	Anfang der Datei
SEEK_CUR	Aktuelle Schreib-/Leseposition
SEEK_END	Ende der Datei

Tabelle 11.1.: Werte zur Angabe des Bezugspunkts bei `fseek()`

```

17     exit(1);
18 }
19
20 printf("Testing fscanf() function:\n\n");
21 printf("Name:\t\t\tRoll\t\tMarks\n");
22
23 while( fscanf(fp, "%s%s%d%f", strFirstName, strLastName, &roll_no, &marks) != EOF )
24 {
25     printf("%s%s\t\t%d\t\t%.2f\n", strFirstName, strLastName, roll_no, marks);
26 }
27
28 fclose(fp);
29 return 0;
30 }

```

11.4.7. `fseek()`

Wie bereits in 11.4.1 dargestellt, enthält die `FILE` Struktur auch eine Angabe, wo in der Datei aktuell gelesen oder geschrieben wird. Sehr oft beginnt das Lesen und Schreiben der Datei am Anfang der Datei und geht dann bis zum Ende. Was aber, wenn wir an einer ganz bestimmten Stelle etwas lesen wollen, dann müssen wir die Schreib- Leseposition erst an diese Stelle bringen. Dazu dient der Befehl `fseek()`.

```
int fseek(FILE *fp, long offset, int origin);
```

Das erste Argument ist dabei der Pointer auf die Datei, dessen Schreib-/Leseposition verschoben werden soll. `origin` gibt den Bezugspunkt an von dem aus die Position um `offset` verschoben werden soll. Für diesen Bezugspunkt hat C drei feste Werte vordefiniert.

So wird dann etwa mit

```
fseek(fp, 0, SEEK_SET);
```

die Schreib-/Leseposition an den Anfang der Datei gesetzt, damit diese dann beispielsweise von Anfang an gelesen werden kann. Dasselbe könnte man natürlich auch erreichen, indem man die Datei mit `fclose()` schließt und anschließend etwa zum Lesen wieder öffnet.

11.5. Challenges

11.5.1. Anlegen einer Datei

Legen Sie im Arbeitsverzeichnis eine neue Datei **namen.dat** an. Lesen Sie von der Konsole Vornamen und Nachnamen von fünf verschiedenen Personen ein, und schreiben Sie diese in die Datei.

11.5.2. Auslesen einer Datei

Öffnen Sie die Datei **namen.dat** und geben Sie den Inhalt der Datei auf der Konsole aus.

11.5.3. Anfügen an eine Datei

Zeigen Sie den Inhalt der Datei **namen.dat** an, und ergänzen Sie diese um eine weitere Person, deren Namen Sie über die Konsole erfragen. Zeigen Sie im Anschluss den Inhalt der Datei auf der Konsole an.

12. Komplexe Datenstrukturen (struct)

12.1. Die struct Anweisung

Oft hat man Variablen oder Daten, die inhaltlich zusammengehören. Werden etwa in einem Programm Personendaten verarbeitet, so gehören der Vorname, der Nachname oder das Alter einer Person inhaltlich zusammen. Im Sinne der Lesbarkeit eines Programms wäre wünschenswert, die zugehörigen Variablen entsprechend gruppieren zu können. Dieses geschieht mittels des Schlüsselwortes **struct** und definiert damit die Definition einer Struktur, die aus zwei Strings und einem Integer besteht..

```
struct person {
    char strFirstName[40];
    char strLastName[40];
    int iAge;
}
```

Die Strukturanweisung reserviert noch keinen Speicherplatz, sondern definiert nur den Namen und den Aufbau einer Struktur. Möchte man jetzt eine Instanz dieser Struktur anlegen, in der Daten gespeichert werden können, so muss ebenfalls das **struct** Schlüsselwort verwendet werden.

```
struct person aPerson;
```

Wie können jetzt Werte dieser Instanz **aperson** der Struktur **person** zugewiesen werden, beziehungsweise wie kann auf die Werte zugegriffen werden?

```
//assigning values to a structure
strcpy(aPerson.strFirstName, "Anna");
strcpy(aPerson.strLastName, "Bolika");
aPerson.iAge = 31;

//using values in a structure
printf("\n%s %s is %d years old", aPerson.strFirstName, aPerson.strLastName, aPerson.iAge);
```

Möchte man häufiger eine bestimmte Struktur verwenden, so kann man diese mit dem **typedef** Schlüsselwort abkürzen.

```
typedef struct person {
    char strFirstName[40];
```

12. Komplexe Datenstrukturen (struct)

```
    char strLastName[40];
    int iAge;
} pers;

pers aPerson;    // Anlegen einer Instanz
```

Zuweisungen und Verwenden geschehen jetzt ganz genauso wie in dem obigen Beispiel. Es gibt es lediglich einen neuen, komplexen "Datentypnamens **person** der einfach in Analogie zu den bekannten Datentypen über das Alias **pers** verwendet werden kann. Natürlich können jetzt auch Arrays dieser Struktur angelegt werden.

```
pers AllPersons[5];
```

Hiermit wird ein Array angelegt, dass die Daten von 5 Personen verwalten kann.

12.2. Übergabe von Strukturen an Funktionen

12.2.1. Übergabe von Strukturen an Funktionen - passing by value

Normalerweise werden auch Strukturen als Werte an Funktionen übergeben. Also wird an die Funktionen eine Kopie der Struktur übergeben, so dass die Funktion die übergebenen Werte nicht ändern kann, wie wir das auch schon von der Übergabe von Variablen einfacher Datentypen her kennen. Das Listing 12.1 zeigt auf, dass die Funktion, die Werte der übergebenen Struktur *by value* erhält, diese nicht verändert.

Listing 12.1: Übergabe einer Struktur by value

```
1 /* 12-passbyvalue.c */
2
3 #include <stdio.h>
4 #include <string.h>
5
6 typedef struct employee {
7     int id;
8     char name[40];
9     float salary;
10 } e_type;
11
12 void processEmp(e_type);
13
14 int main()
15 {
16     e_type empl = {0,0,0};    //initialize
17     processEmp(empl);
18
19     printf("\n\nID: %d", empl.id );
20     printf("\nName: %s", empl.name);
21     printf("\nSalary: %.2f\n", empl.salary);
```

```

22     return 0;
23 }
24
25 void processEmp(e_type emp)    //receives a copy of the structure
26 {
27     emp.id = 123;
28     strcpy(emp.name, "Anna");
29     emp.salary = 65000.00;
30 }

```

12.2.2. Übergabe von Strukturen an Funktionen - passing by reference

Soll eine Funktion aber die Werte einer übergebenen Struktur verändern können, so muss auch hier wieder die Struktur *by reference*, also als Pointer, übergeben werden. In diesem Falle muss der Pointer mit dem Operator `->` dereferenziert werden, damit auf die Elemente der Struktur zugegriffen werden kann.

```

e_type *empl;
...
empl->salary = 60000.00;

```

Damit ändert sich das obige Programmbeispiel zu dem Listing 12.2. Jetzt kann der Inhalt der Struktur-Variablen geändert werden, da diese als Pointer an die Funktion übergeben wurde. Allerdings kann nicht mehr der Punkt-Operator zum Zugriff auf die Elemente der Struktur verwendet werden, sondern es muss der `->` verwendet werden, wie in dem Listing 12.2 gezeigt wird.

Listing 12.2: Übergabe einer Struktur by reference

```

1  /* 12-passbyreference.c */
2
3  #include <stdio.h>
4  #include <string.h>
5
6  typedef struct employee {
7      int id;
8      char name[40];
9      float salary;
10 } e_type;
11
12 void processEmp(e_type *);
13
14 int main()
15 {
16     e_type empl = {0,0,0};    //initialize
17
18     e_type *ptrEmp;
19     ptrEmp = &empl;
20     processEmp(ptrEmp);

```

12. Komplexe Datenstrukturen (struct)

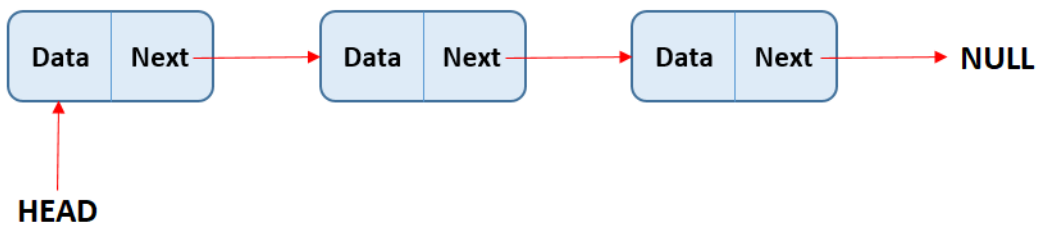


Abbildung 12.1.: Einfach verkettete Liste

```
21
22     printf("\n\nID: %d", ptrEmp->id );
23     printf("\nName: %s", ptrEmp->name);
24     printf("\nSalary: %.2f\n", ptrEmp->salary);
25     return 0;
26 }
27
28 void processEmp(e_type *emp)    //receives a copy of the structure
29 {
30     emp->id = 123;
31     strcpy(emp->name, "Anna");
32     emp->salary = 65000.00;
33 }
```

12.3. Anwendungsbeispiel: Verkettete Listen

In diesem Abschnitt soll als Beispiel für bisher gelernte Konzepte eine sogenannte verkettete Liste erstellt werden. Dabei werden sowohl Strukturen als auch Pointer benötigt. Verkettete Listen sind ein gutes Beispiel für dynamische Datenstrukturen unter Verwendung von Pointern und der Allokation von Speicher sowie der Nutzung von Strukturen.

Eine verkettete List ist, wie in Abbildung 12.1 dargestellt, eine Menge dynamisch allozierter Knoten, die jeweils aus Daten und einen Pointer bestehen. Der Pointer verweist dabei auf den nachfolgenden Knoten. Hat der Pointer den Wert `NULL`, dann befindet er sich am Ende der Liste und hat keinen Nachfolgerknoten mehr. Auf eine verkettete Liste wird mit einem Pointer auf den ersten Knoten (Head) verwiesen. Ist dieser Pointer `NULL`, dann ist die Liste leer.

Im Prinzip funktioniert eine verkettete Liste wie ein Array, das bei Bedarf an jeder Stelle wachsen und schrumpfen kann. Gegenüber Arrays haben verkettete Listen einige Vorteile:

- Elemente können in der Mitte der Liste hinzugefügt oder entfernt werden.
- Es muss keine anfängliche Größe der Liste angegeben werden.

Allerdings haben diese auch einige deutliche Nachteile:

- Auf Elemente kann nicht direkt zugegriffen werden, so ist es nicht möglich direkt auf das n-te Element zuzugreifen ohne über alle vorangegangenen Elemente zu gehen.
- Es ist eine dynamische Speicherallokation und die Verwendung von Pointern erforderlich, wodurch sich die Komplexität des Programms erhöht und Speicherlecks möglich werden.
- Verkettete Listen sind in der Speicherung ineffizienter als Arrays und benötigen zu den Daten noch Pointer auf die Folgeelemente.

Das Listing 12.3 ist ein einfaches Beispiel für eine verkettete Liste.

Listing 12.3: Beispiel für eine einfach verkettete Liste

```

1  /* 12-linkedlist */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  struct Data {
8      int day;
9      int month;
10     int year;
11     char event[80];
12 };
13
14 // Define structure of a node
15
16 struct LinkedListNode{
17     struct Data nodedata;
18     struct LinkedListNode *next;    //pointer to next node in list
19 };
20
21 // define NodePointer as a pointer to a node in the list
22 typedef struct LinkedListNode *NodePointer; //define NodePointer of
23                                           //data type struct LinkedList
24
25 NodePointer createNode(){
26     //creates a new empty node
27
28     NodePointer ptrNewNode;    // declare a node
29
30     // allocate memory for new node
31     ptrNewNode = (NodePointer)malloc(sizeof(struct LinkedListNode));
32
33     // check if allocation of memory was successful
34     if (ptrNewNode == NULL) {

```

12. Komplexe Datenstrukturen (struct)

```
35         fprintf(stderr, "\nUnable to allocate memory");
36         exit(-1);
37     }
38
39     ptrNewNode->next = NULL;    // make next point to NULL
40     return ptrNewNode;        //return the address of the new node
41 }
42
43 NodePointer addNodeEndOfList(NodePointer head,
44                             int year,
45                             int month,
46                             int day,
47                             char *event){
48
49     NodePointer ptrNewNode; //pointer to newly created node
50     NodePointer ptrNode;    //used as pointer to nodes
51
52     ptrNewNode = createNode(); //create new node and fill it with values
53     ptrNewNode->nodedata.year = year;    // fill with values
54     ptrNewNode->nodedata.month = month;
55     ptrNewNode->nodedata.day = day;
56     strcpy(ptrNewNode->nodedata.event, event);
57
58     // if list is empty make node the head of list
59     if(head == NULL){
60         head = ptrNewNode;    //when linked list is empty
61     }
62     else{
63         //go to end of list
64         ptrNode = head; //assign head to p
65         //now traverse the list until p is the last node.
66         //The last node always points to NULL.
67         while(ptrNode->next != NULL){
68             ptrNode = ptrNode->next;
69         }
70         //Point the previous last node to the new node created.
71         ptrNode->next = ptrNewNode;
72     }
73     return head;    // return the address of first node
74 }
75
76 void printList(NodePointer head)
77 {
78     NodePointer ptrNode;
79     ptrNode = head;
80     while(ptrNode != NULL){
81         // print data of node
82         printf("\n%d-%02d-%02d-_%s",
83             ptrNode->nodedata.year,
```



```

84         ptrNode->nodedata.month,
85         ptrNode->nodedata.day,
86         ptrNode->nodedata.event);
87     // traverse to next node
88     ptrNode = ptrNode->next;
89 }
90 }
91
92 int main()
93 {
94     NodePointer head = NULL; //pointer to first node of list
95
96     head = addNodeEndOfList(NULL, 2022, 1, 1, "New Year");
97     addNodeEndOfList(head, 2022, 1, 11, "C Programming Lecture");
98     addNodeEndOfList(head, 2022, 1, 11, "C Programming Exercise");
99     addNodeEndOfList(head, 2022, 1, 13, "C Programming Exercise");
100    addNodeEndOfList(head, 2022, 1, 10, "Restart of Lectures");
101    printList(head);
102 }

```

12.4. Challenges

12.4.1. Studierendendaten

Schreiben Sie die Listings 11.4 und 11.3 so um, dass alle Daten zu einem Studierenden, wie `strFirstName`, `strLastName`, `roll_no` und `marks`, in einer Struktur vereint werden.

12.4.2. Sortierte, verkettete Listen

Schreiben Sie die Funktion `addNode()` im Listing 12.3 so um, dass neue Knoten nach Datum sortiert eingefügt werden und nicht automatisch am Ende der Liste angehängt werden. So wird beispielsweise das hinzukommende Event für das Datum 2021-01-13 vor dem Event am 2021-01-14 eingefügt.

12.4.3. Löschen aus verketteten Listen

Schreiben Sie eine Funktion, die nach Einträgen für ein bestimmtes Datum sucht, und diese dann löscht. Achten Sie darauf, dass der allozierte Speicher auch wieder freigegeben wird.

13. Debugging und Umgang mit Fehlern

13.1. Debugging

13.1.1. Debugging mit printf()

Nachdem ein Programm korrekt übersetzt worden ist, können trotzdem zur Laufzeit noch Fehler auftreten, sei es, dass es etwa unverhofft abbricht oder falsche Werte berechnet werden. Häufige Praxis zur Fehlersuche, dem Debugging, ist es oft, sich durch eingefügte Ausgaben Informationen über den Programmablauf zu verschaffen. Dieses wird im folgenden an dem einfachen Listing 13.1 erläutert. In diesem Programm sind einige Fehler eingebaut, die erst zur Laufzeit auftreten werden.

Listing 13.1: Listing mit Laufzeitfehlern

```
1 #include <stdio.h>
2 #include <math.h>
3
4 float xdivision(float , float );
5 float xsqr(float);
6 float xsqrt(float);
7
8 int main()
9 {
10     float x;
11     float y;
12     float z;
13
14     x = 8.0;
15     y = 4.0;
16     z = xdivision(x, y);
17     printf("\nDivision: %f / %f = %f", x, y, z);
18     z = xsqr(x);
19     printf("\nsqr(%f) = %f", x, z);
20     z = xsqrt(x);
21     printf("\nsqrt(%f) = %f\n\n", x, z);
22
23     x = 8.0;
24     y = 0.0;
25     z = xdivision(x, y);
26     printf("\nDivision: %f / %f = %f", x, y, z);
27     z = xsqr(x);
28     printf("\nsqr(%f) = %f", x, z);
29     z = xsqrt(x);
```

13. Debugging und Umgang mit Fehlern

```
30     printf("\nsqrt(%f) = %f\n\n", x, z);
31
32     x = -8.0;
33     y = 4.0;
34     z = xdivision(x, y);
35     printf("\nDivision: %f / %f = %f", x, y, z);
36     z = xsqr(x);
37     printf("\nsqr(%f) = %f", x, z);
38     z = xsqrt(x);
39     printf("\nsqrt(%f) = %f\n\n", x, z);
40
41     return 0;
42 }
43
44 float xdivision(float x, float y) {
45     float z;
46
47     z = x / y;
48     return z;
49 }
50
51 float xsqr(float x) {
52     float z;
53
54     z = x * x;
55     return z;
56 }
57
58 float xsqrt(float x) {
59     float z;
60
61     z = sqrt(x);
62     return z;
63 }
```

So wird versucht durch 0 zu dividieren oder die Wurzel aus einer negativen Zahl zu ziehen. Um diese Fehler zu suchen, wird versucht durch `printf()` Anweisungen in den Funktionen das Auftreten des Fehlers, der nur bei bestimmten Werten passiert, zu erkennen, wie in Listing 13.2 gezeigt.

Listing 13.2: Fehlersuche mit `printf()`

```
1 #include <stdio.h>
2 #include <math.h>
3
4 float xdivision(float , float );
5 float xsqr(float);
6 float xsqrt(float);
7
8 int main()
```

```

9 {
10     float x;
11     float y;
12     float z;
13
14     x = 8.0;
15     y = 4.0;
16     z = xdivision(x, y);
17     printf("\nDivision: %f / %f = %f", x, y, z);
18     z = xsqr(x);
19     printf("\nsqr(%f) = %f", x, z);
20     z = xsqrt(x);
21     printf("\nsqrt(%f) = %f\n\n", x, z);
22
23     x = 8.0;
24     y = 0.0;
25     z = xdivision(x, y);
26     printf("\nDivision: %f / %f = %f", x, y, z);
27     z = xsqr(x);
28     printf("\nsqr(%f) = %f", x, z);
29     z = xsqrt(x);
30     printf("\nsqrt(%f) = %f\n\n", x, z);
31
32     x = -8.0;
33     y = 4.0;
34     z = xdivision(x, y);
35     printf("\nDivision: %f / %f = %f", x, y, z);
36     z = xsqr(x);
37     printf("\nsqr(%f) = %f", x, z);
38     z = xsqrt(x);
39     printf("\nsqrt(%f) = %f\n\n", x, z);
40
41     return 0;
42 }
43
44 float xdivision(float x, float y) {
45     float z;
46
47     printf("\n\t--- Inside xdivision\n\t--- y = %f", y);
48     z = x / y;
49     return z;
50 }
51
52 float xsqr(float x) {
53     float z;
54     printf("\n\t--- Inside xsqr\n\t--- x = %f", x);
55     z = x * x;
56     return z;
57 }

```

13. Debugging und Umgang mit Fehlern

```
58
59 float xsqrt(float x) {
60     float z;
61     printf("\n\t---_Inside_xsqr\n\t---_x=_%f", x);
62     z = sqrt(x);
63     return z;
64 }
```

Hier sind in den aufgerufenen Funktionen `printf()` Anweisungen eingefügt worden, die bei jedem Aufruf die übergebenen, kritischen Parameter ausgeben. So hilfreich das bei der Suche nach den Fehlern ist, führt es dazu, dass die Ausgabe sehr umfangreich wird, und es müssen spätestens bei der Fertigstellung des fehlerfreien Programms, die ganzen Anweisungen wieder gelöscht werden, damit die Ausgaben verschwinden und das Programm auch wieder schneller wird. Sollte der Fehler erneut auftreten müssten die Anweisungen erneut eingefügt werden, was einfach ärgerlich ist.

Dieses Problem kann mittels Präprozessoranweisungen umgangen werden. Solange ein Wert `DEBUG` definiert ist, werden alle Anweisungen, die zwischen `#ifdef DEBUG` und `#endif` stehen, mit kompiliert und ausgeführt. Werden diese Anweisungen nicht mehr benötigt, so muss einfach die Definition von `DEBUG` entfernt werden, und der Präprozessor entfernt alle Anweisungen zwischen `#ifdef DEBUG` und `#endif`. Diese werden wieder eingefügt, sobald `DEBUG` wieder definiert wird. Dieses Vorgehen wird in Listing 13.3 gezeigt.

Listing 13.3: Nutzung des Präprozessors zur Fehlersuche

```
1 #include <stdio.h>
2 #include <math.h>
3
4 #define DEBUG    1
5
6 float xdivision(float , float );
7 float xsqr(float);
8 float xsqrt(float);
9
10 int main()
11 {
12     float x;
13     float y;
14     float z;
15
16     x = 8.0;
17     y = 4.0;
18     z = xdivision(x, y);
19     printf("\nDivision:_%f/_%f=_%f", x, y, z);
20     z = xsqr(x);
21     printf("\nsqr(%f)_=%f", x, z);
22     z = xsqrt(x);
23     printf("\nsqrt(%f)_=%f\n\n", x, z);
24
25     x = 8.0;
```

```

26     y = 0.0;
27     z = xdivision(x, y);
28     printf("\nDivision: %f / %f = %f", x, y, z);
29     z = xsqr(x);
30     printf("\nsqr(%f) = %f", x, z);
31     z = xsqrt(x);
32     printf("\nsqrt(%f) = %f\n\n", x, z);
33
34     x = -8.0;
35     y = 4.0;
36     z = xdivision(x, y);
37     printf("\nDivision: %f / %f = %f", x, y, z);
38     z = xsqr(x);
39     printf("\nsqr(%f) = %f", x, z);
40     z = xsqrt(x);
41     printf("\nsqrt(%f) = %f\n\n", x, z);
42
43     return 0;
44 }
45
46 float xdivision(float x, float y) {
47     float z;
48
49     #ifdef DEBUG
50         printf("\n\t--- Inside xdivision\n\t--- y = %f", y);
51     #endif
52     z = x / y;
53     return z;
54 }
55
56 float xsqr(float x) {
57     float z;
58     #ifdef DEBUG
59         printf("\n\t--- Inside xsqr\n\t--- x = %f", x);
60     #endif
61     z = x * x;
62     return z;
63 }
64
65 float xsqrt(float x) {
66     float z;
67     #ifdef DEBUG
68         printf("\n\t--- Inside xsqrt\n\t--- x = %f", x);
69     #endif
70     z = sqrt(x);
71     return z;
72 }

```

13.1.2. Debugging mit `assert.h`

Eine andere Möglichkeit der Fehlersuche kann mittels der Include-Datei `assert.h` umgesetzt werden. Mit dem `assert`-Makro aus `assert.h` können wir Testpunkte zu unserem Programmen hinzufügen.

```
void assert(int expression);
```

Dabei kann der Ausdruck `expression` jede Form annehmen, die zu `TRUE` oder `FALSE` evaluiert werden kann. Solange der Ausdruck `expression` `TRUE` ist, passiert gar nichts. Wird der Ausdruck aber zu `FALSE` evaluiert, dann gibt das `assert`-Makro auf `stderr` ungefähr folgende Meldung aus:

```
Assertion failed: expression, file filename, line nnn
```

Im Anschluss wird die weitere Ausführung des Programms abgebrochen. An dem Dateinamen und der Angabe der Zeile lässt sich der Fehler gut nachvollziehen. Der Dateiname der Programmquelle sowie die Zeilennummer stammen von den Prozessor-Makros `__FILE__` und `__LINE__`.

Möchten wir nach erfolgreicher Fehlerkorrektur etwa zur Laufzeitverbesserung die `assert`-Makros wieder deaktivieren, um sie vielleicht später wiederverwenden zu können, dann muss lediglich beim Einfügen von `assert.h` der Wert `NDEBUG` definiert sein, und der Präprozessor ignoriert alle `assert`-Makros. Listing 13.3 zeigt die Verwendung des `assert`-Makros.

Listing 13.4: `assert` zur Fehlersuche

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <assert.h>
4
5 float xdivision(float , float );
6 float xsqr(float);
7 float xsqrt(float);
8
9 int main()
10 {
11     float x;
12     float y;
13     float z;
14
15     x = 8.0;
16     y = 4.0;
17     z = xdivision(x, y);
18     printf("\nDivision: %f / %f = %f", x, y, z);
19     z = xsqr(x);
20     printf("\nsqr(%f) = %f", x, z);
21     z = xsqrt(x);
22     printf("\nsqrt(%f) = %f\n\n", x, z);
```



```

23
24     x = 8.0;
25     y = 0.0;
26     z = xdivision(x, y);
27     printf("\nDivision: %f / %f = %f", x, y, z);
28     z = xsqr(x);
29     printf("\nsqr(%f) = %f", x, z);
30     z = xsqrt(x);
31     printf("\nsqrt(%f) = %f\n\n", x, z);
32
33     x = -8.0;
34     y = 4.0;
35     z = xdivision(x, y);
36     printf("\nDivision: %f / %f = %f", x, y, z);
37     z = xsqr(x);
38     printf("\nsqr(%f) = %f", x, z);
39     z = xsqrt(x);
40     printf("\nsqrt(%f) = %f\n\n", x, z);
41
42     return 0;
43 }
44
45 float xdivision(float x, float y) {
46     float z;
47
48     assert(y != 0);
49     z = x / y;
50     return z;
51 }
52
53 float xsqr(float x) {
54     float z;
55
56     z = x * x;
57     return z;
58 }
59
60 float xsqrt(float x) {
61     float z;
62
63     assert(x >= 0);
64     z = sqrt(x);
65     return z;
66 }

```

13.2. Umgang mit Fehlern

C bietet nur sehr rudimentäre Möglichkeiten des Umgangs mit Fehlern zur Programmlaufzeit. In der Regel werden Fehler in Form von Rückgabewerten von Funktionen behandelt. So geben viele Funktionen etwa die Werte -1 oder NULL im Fehlerfall zurück und setzen einen Fehlercode, der nähere Angaben über die Fehlerart macht, in einer globalen Variablen namens `errno`. Die zugehörigen Fehlerbeschreibungen zu den Werten in `errno` werden in `error.h` definiert.

Damit kann ein C Programm den zurückgegebenen Wert überprüfen und geeignete Maßnahmen ergreifen. Es ist gute Praxis, die globale Variable `errno` bei Programmstart mit 0 zu initialisieren, sofern diese zur Fehlerbehandlung verwendet wird. Ein Wert von 0 bedeutet dann, dass bisher kein Fehler aufgetreten ist.

13.2.1. `errno`, `perror()` und `strerror()`

Die Funktionen `perror()` und `strerror()` übersetzen den numerischen Fehlercode in `errno` in eine Fehlnachricht.

- Die `perror()` Funktion zeigt einen String des Entwicklers, der von einem Komma und einem Leerzeichen gefolgt wird sowie die Fehlnachricht, die dem Wert von `errno` entspricht, an.
- Die `strerror()` Funktion gibt einen Pointer auf die Fehlnachricht, die dem Wert von `errno` entspricht, zurück.

In dem Listing 13.5 wird versucht, eine nicht existierende Datei zu öffnen. In dem Beispiel werden beide Funktionen gezeigt. Üblicherweise wird nur eine von beiden verwendet. Ebenfalls wichtig ist, dass in dem Beispiel `stderr` für die Fehlerausgabe verwendet wird. Damit lassen sich leicht Fehlermeldungen in etwa eine Datei, als Fehlerprotokolldatei, umleiten. Dieses ist insbesondere bei umfangreichen, komplexen Programmen hilfreich, die viele Fehler programmtechnisch behandeln, aber dennoch sollen die Fehler dokumentiert werden.

Listing 13.5: Fehlerbehandlung mittel `strerror()` und `perror()`

```
1 #include <stdio.h>
2 #include <errno.h>
3 #include <string.h>
4
5 // errno is a global variable declared by errno.h
6
7 int main () {
8
9     FILE *fp;
10    errno = 0;
11
12    fp = fopen ("unexist.txt", "rb");
```

```

13
14     if (fp == NULL) {
15         fprintf(stderr, "Value_of_errno:_%d\n", errno);
16         perror("Error_printed_by_perror");
17         fprintf(stderr, "Error_opening_file:_%s\n", strerror(errno));
18     }
19     else {
20         fclose (fp);
21     }
22     return 0;
23 }

```

Ein anderes, nicht unübliches Problem sind Divisionen durch 0, diese werden oft in den Programmen nicht berücksichtigt und verursachen zur Laufzeit Fehler. Auch hier ist es eine gute Praxis solche Fehler abzufangen, wie in Listing 13.6 gezeigt.

Listing 13.6: Fehlerbehandlung der Division durch 0

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int dividend = 20;
7     int divisor = 5;
8     int quotient;
9
10    if( divisor == 0) {
11        fprintf(stderr, "Division_by_zero!_Exiting...\n");
12        exit(EXIT_FAILURE);
13    }
14
15    quotient = dividend / divisor;
16    fprintf(stderr, "Value_of_quotient:_%d\n", quotient );
17
18    return(EXIT_SUCCESS);
19 }

```

Wird eine Programm erfolgreich beendet und mittels `exit()` oder `return` verlassen, sollte durch den Returnwert `EXIT_SUCCESS` verwenden, damit dieses gegebenenfalls überprüft werden kann. Dabei ist `EXIT_SUCCESS` in `stdlib.h` mit dem Wert 0 definiert.

Wird das Programm hingegen mit einem Fehler beendet, so sollte der Wert `EXIT_FAILURE` verwendet werden. Dieser ist als -1 definiert. Diese Praxis ist in dem Listing 13.6 ebenfalls gezeigt.

14. Rekursion

14.1. Einleitung

Üblicherweise stellt man sich die meisten Programme so vor, dass von einer Funktion aus weitere Funktionen aufgerufen werden, die wiederum andere Funktionen aufrufen. Warum soll also eine Funktion sich nicht selber aufrufen? Dieses kann indirekt erfolgen, indem eine andere Funktion dazwischen liegt, oder aber direkt, indem die Funktion sich selber aufruft. Einen solchen Vorgang nennt man *Rekursion*. Interessanterweise kann man damit einige Probleme in sehr einfacher und eleganter Weise lösen, wie wir sehen werden.

14.2. Beispiele für Rekursionen

14.2.1. Fakultät

Die Fakultät $n!$ einer natürlichen Zahl n ist ein einfaches Beispiel für eine Funktion, die sowohl herkömmlich und auch als Rekursion programmiert werden kann. Dabei ist $n!$ das Produkt der natürlichen Zahlen von 1 bis n . Also

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Diese Zahl lässt sich herkömmlich iterativ sehr einfach in einer Funktion, wie in Listing 14.1 gezeigt, berechnen.

Listing 14.1: Iterative Berechnung der Fakultät

```
1 /* 14-facult_iterativ */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int fakultaet_iterativ(int n) {
6     int fak;
7     for (fak = 1; n > 1; n--)
8         fak *= n;
9     return fak;
10 }
11
12 int main() {
13     int n;
14     for (n = 0; n < 10; n++)
15         printf("\n%d!=_%d", n, fakultaet_iterativ(n));
16     return EXIT_SUCCESS;
17 }
```

14. Rekursion

Es gibt aber auch eine andere Definition der Fakultät, die hier bereits rekursiv definiert wird:

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{falls } n > 1 \end{cases}$$

Diese Definition bietet sich an, sie direkt in eine rekursiv arbeitende Funktion umzuwandeln, die sich selber mit dem nächstkleineren Argument aufruft. Dieses wurde im Listing 14.2 in C umgesetzt:

Listing 14.2: Rekursive Berechnung der Fakultät

```
1 /* 14-facult_rekursiv */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int fakultaet_rekursiv(int n) {
6     if (n <= 1)
7         return 1;
8     return n * fakultaet_rekursiv(n - 1);
9 }
10
11 int main() {
12     int n;
13     for (n = 0; n < 10; n++)
14         printf("\n%d! = %d", n, fakultaet_rekursiv(n));
15     return EXIT_SUCCESS;
16 }
```

Die Funktion `fakultaet_rekursiv()` ruft sich selber dabei solange auf, bis das übergebene Argument `n` den Wert 1 erreicht hat. Jetzt ruft sich die Funktion nicht mehr selber auf, sondern gibt einfach den Wert 1 zurück. Hieran kann man deutlich erkennen, dass jede rekursiv definierte Funktion auch eine Bedingung beinhalten lässt, die dafür sorgt, dass sie sich nicht endlos selber aufruft. Iterative und rekursive Funktionen unterscheiden sich oft nicht sehr und liefern natürlich dieselben Ergebnisse, betrachtet man aber den Bedarf an Speicherplatz und Laufzeit, so schneiden rekursive Funktionen in der Regel deutlich schlechter ab, da jeder Aufruf einer Funktion auch wieder Ressourcen benötigt. Allerdings lassen sich manche Probleme sehr elegant lösen, wie der folgende Abschnitt 14.2.2 über das Problem der Türme von Hanoi zeigt. Das Prinzip der Rekursion besteht nämlich darin, ein großes Problem auf ein kleineres Problem derselben Art zurückzuführen.

14.2.2. Türme von Hanoi

Das Problem der Türme von Hanoi wurde 1883 von dem Mathematiker Edouard Lucas beschrieben. Das Ziel des Spieles besteht darin, alle Scheiben des Startturms A auf dem Zielturm C zu sammeln. Dabei kann der temporäre Turm B genutzt werden (siehe Abb. 14.1). Dabei gelten die folgenden Regeln:

- Es kann ein temporärer Hilfsturm auf B aufgebaut werden.

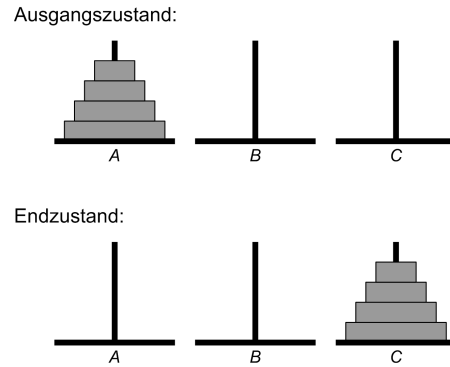


Abbildung 14.1.: Aufgabenstellung der Türme von Hanoi

- Es darf immer nur ein Ring zur Zeit bewegt werden.
- Es darf nie ein größerer Ring auf einem kleineren Ring zu liegen kommen. Das heißt, die Türme müssen immer der Größe nach sortiert sein.

Überlegen Sie einmal, wie man ein Programm schreiben könnte, dass die einzelnen erforderlichen Schritte eines Stapels von 5 Ringen berechnet. Nicht einfach? Sehen wir uns das folgende Listing 14.3 an, das genau dieses macht:

Listing 14.3: Lösung der Türme von Hanoi

```

1 /* 14-hanoi.c */
2
3 #include <stdio.h>
4
5 void hanoi(int n, char start, char tmp, char ziel) {
6     if (n > 1) {
7         hanoi(n - 1, start, ziel, tmp);
8         printf("\nRing %d: %c->%c", n, start, ziel);
9         hanoi(n - 1, tmp, start, ziel);
10    }
11    else {
12        printf("\nRing %d: %c->%c", n, start, ziel);
13    }
14 }
15
16 int main() {
17     hanoi(5, 'S', 'T', 'Z');
18 }

```

Das Problem ist in nur 14 Programmzeilen rekursiv gelöst! Sind Sie darauf gekommen? Die Funktion `hanoi` bewegt dabei `n` Ringe von `start` (A) über `tmp` (B) zum `ziel`. Wenn dabei mehr als ein Ring zu bewegen ist, dann sollen die folgenden Schritte ausgeführt werden:

14. Rekursion

- Bewege n-1 Ringe von **start** über **ziel** nach **tmp**
- Bewege den n-ten Ring nach **ziel**
- Bewege n-1 Ringe von **tmp** über **start** nach **ziel**
- Wenn nur ein Ring zu bewegen ist, bewege ihn direkt von **start** nach **ziel**

Aus der Beschreibung erkennt man, dass das Problem von **n** Scheiben auf ein Problem von **n-1** Scheiben zurückgeführt worden ist. Kann man es also für eine Scheibe lösen, dann kann man es für zwei Scheiben, und dann drei Scheiben usw. lösen. Das Prinzip der vollständigen Induktion.

14.3. Verkettete Listen reloaded

In Abschnitt 12.3 wurden die einfach verketteten Listen eingeführt. Deren Ziel war es, dynamisch wachsende oder schrumpfende Listen zu verwalten. Mittels eine Funktion zum Vergleich der Daten zweier Knoten können neue Knoten der Liste so an- oder eingefügt werden, dass die Daten der Liste der Größe nach sortiert sind. Das dazugehörige Programm ist in Listing 14.4 dargestellt. Insbesondere wird in der Hauptfunktion **main** versucht, die Funktionen systematisch zu testen.

Listing 14.4: Sortierte, verkettete Liste

```
1 /* 12-linkedlist */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 // define structure of data
8 struct Data {
9     int day;
10    int month;
11    int year;
12    char event[80];
13 };
14
15 // Define structure of a node
16 struct LinkedListNode{
17     struct Data nodedata;
18     struct LinkedListNode *next;    //pointer to next node in list
19 };
20
21 // define NodePointer as a pointer to a node in the list
22 typedef struct LinkedListNode *NodePointer;
23
24 // declare functions
25 int compareNode(NodePointer, NodePointer);
```



```

26 int compareNodeToDate(NodePointer, int, int , int);
27 void printNode(NodePointer);
28 NodePointer createNode();
29 NodePointer addNodeEndOfList(NodePointer, int, int, int, char *);
30 NodePointer findNode(NodePointer, int, int, int);
31 NodePointer deleteNode(int, int, int);
32 long convertDate(int, int, int);
33
34 long convertDate(int year, int month, int day) {
35     long lResult;
36     lResult = year * 100000 + month * 100 + day;
37     return lResult;
38 }
39
40 int compareNodeToDate(NodePointer ptrNode, int year, int month, int day) {
41     // return -1 if ptr > date
42     // return 0  if ptr = date
43     // return 1  if prt < date
44     long lTest1, lTest2;
45
46     lTest1 = convertDate(ptrNode->nodedata.year,
47                          ptrNode->nodedata.month,
48                          ptrNode->nodedata.day);
49
50     lTest2 = convertDate(year, month, day);
51
52     if (lTest1 < lTest2)
53         return 1;
54     else {
55         if (lTest1 > lTest2)
56             return -1;
57         else
58             return 0;
59     }
60 }
61
62
63
64 int compareNode(NodePointer ptrNode1, NodePointer ptrNode2) {
65     // return -1 if ptr1 > ptr2
66     // return 0  if ptr1 = ptr2
67     // return 1  if prt1 < ptr2
68     long lTest1, lTest2;
69
70     lTest1 = convertDate(ptrNode1->nodedata.year,
71                          ptrNode1->nodedata.month,
72                          ptrNode1->nodedata.day);
73
74     lTest2 = convertDate(ptrNode2->nodedata.year,

```

14. Rekursion

```
75         ptrNode2->nodedata.month,
76         ptrNode2->nodedata.day);
77
78     if (lTest1 < lTest2)
79         return 1;
80     else {
81         if (lTest1 > lTest2)
82             return -1;
83         else
84             return 0;
85     }
86 }
87
88 NodePointer findNode(NodePointer head, int year, int month, int day) {
89     NodePointer ptrNode;
90
91     ptrNode = head;
92     while(ptrNode != NULL){
93         // if node equals date, return address of node
94         if (compareNodeToDate(ptrNode, year, month, day) == 0) {
95             return ptrNode;
96         }
97
98         ptrNode = ptrNode->next;
99     }
100     printf("\n");
101     return ptrNode;
102 }
103
104 void printNode(NodePointer ptrNode) {
105     printf("\n%d-%02d-%02d-□□s",
106         ptrNode->nodedata.year,
107         ptrNode->nodedata.month,
108         ptrNode->nodedata.day,
109         ptrNode->nodedata.event);
110 }
111
112 NodePointer createNode(){
113     //creates a new empty node
114
115     NodePointer ptrNewNode;    // declare a node
116
117     // allocate memory for new node
118     ptrNewNode = (NodePointer)malloc(sizeof(struct LinkedListNode));
119
120     // check if allocation of memory was successful
121     if (ptrNewNode == NULL) {
122         fprintf(stderr, "\nUnable to allocate memory");
123         exit(EXIT_FAILURE);
124     }
```

```

124     }
125
126     ptrNewNode->next = NULL; // make next point to NULL
127     return ptrNewNode;      //return address of new node
128 }
129
130 NodePointer addNodeEndOfList(NodePointer head,
131                             int year,
132                             int month,
133                             int day,
134                             char *event){
135
136     NodePointer ptrNewNode; //pointer to newly created node
137     NodePointer ptrNode;    //used as pointer to nodes
138
139     //create new node and fill it with values
140     ptrNewNode = createNode();
141     ptrNewNode->nodedata.year = year;    // fill with values
142     ptrNewNode->nodedata.month = month;
143     ptrNewNode->nodedata.day = day;
144     strcpy(ptrNewNode->nodedata.event, event);
145
146     // if list is empty make node the head of list
147     if(head == NULL){
148         head = ptrNewNode;    //when linked list is empty
149         return head;
150     }
151
152     //go to end of list
153     ptrNode = head; //assign head to p
154     //now traverse the list until p is the last node.
155     //The last node always points to NULL.
156     while(ptrNode->next != NULL){
157         ptrNode = ptrNode->next;
158     }
159     //Point the previous last node to the new node created.
160     ptrNode->next = ptrNewNode;
161
162     return head;    // return the address of first node
163 }
164
165 NodePointer addNodeSorted(NodePointer head,
166                           int year,
167                           int month,
168                           int day,
169                           char *event){
170
171     NodePointer ptrNewNode; //pointer to newly created node
172     NodePointer ptrNode;    //used as pointer to nodes

```

14. Rekursion

```
173
174     //create new node and fill it with values
175     ptrNewNode = createNode();
176     ptrNewNode->nodedata.year = year;      // fill with values
177     ptrNewNode->nodedata.month = month;
178     ptrNewNode->nodedata.day = day;
179     strcpy(ptrNewNode->nodedata.event, event);
180
181     // Case 1:
182     // if list is empty make node the head of list
183     if(head == NULL){
184         head = ptrNewNode;      //when linked list is empty
185         return head;
186     }
187
188     // Case 2: vor dem ersten Element einfuegen
189
190     // Wenn Knoten 1 der Liste > ptrNewNode
191     if (compareNode(head, ptrNewNode) == -1) {
192         ptrNewNode->next = head;
193         head = ptrNewNode;
194         return head;
195     }
196
197     // Case 3: in der Mitte einfuegen
198
199     ptrNode = head;
200     while ((ptrNode->next != NULL)
201           && (compareNode(ptrNewNode, ptrNode->next) == -1)) {
202
203         ptrNode = ptrNode->next;
204     }
205
206     // case 4 am Ende einfuegen
207     if (ptrNode->next == NULL) { //bereits am Ende der Liste)
208         ptrNode->next = ptrNewNode;
209         return head;
210     }
211
212     // case 3 in der Mitte einfügen
213     ptrNewNode->next = ptrNode->next;
214     ptrNode->next = ptrNewNode;
215     return head;
216     // return the address of first node
217 }
218
219 void printList(NodePointer head)
220 {
221     NodePointer ptrNode;
```

```

222     ptrNode = head;
223     while(ptrNode != NULL){
224         // print data of node
225         printNode(ptrNode);
226         // traverse to next node
227         ptrNode = ptrNode->next;
228     }
229     printf("\n\n");
230 }
231
232 int main()
233 {
234     NodePointer ptrNode;
235     NodePointer head = NULL; //pointer to first node of list
236
237     //test addNodeSorted
238     printf("\n\nCase_1: Testing empty file");
239     head = addNodeSorted(head, 2022, 1, 2, "1. Element");
240     printList(head);
241
242     printf("\nCase_2: Insert in front of first node");
243     head = addNodeSorted(head, 2022, 1, 1, "2. Element");
244     printList(head);
245
246     printf("\nCase_3: Insert after last node");
247     head = addNodeSorted(head, 2022, 1, 4, "3. Element");
248     printList(head);
249
250     printf("\nCase_3: Insert between node_2 and_4");
251     head = addNodeSorted(head, 2022, 1, 3, "4. Element");
252     printList(head);
253
254     // test findNode
255     printf("\nTest findNode: existing node");
256     ptrNode = findNode(head, 2022, 1, 2);
257     if (ptrNode == NULL)
258         printf("\nNode not found!");
259     else
260         printNode(ptrNode);
261     printf("\n");
262
263     printf("\nTest findNode: not existing node");
264     ptrNode = findNode(head, 2022, 1, 15);
265     if (ptrNode == NULL)
266         printf("\nNode not found!");
267     else
268         printNode(ptrNode);
269     printf("\n");
270 }

```

14.4. Challenges

14.4.1. Löschen in einer verketteten Liste

Erweitern Sie die Funktionen in dem Listing 14.4 um eine Funktion, die ein Element der Liste zu einem vorgegebenen Datum findet und es dann löscht. Diese Funktion sollte einen Pointer auf den Beginn der Liste zurückgeben, oder `NULL`, falls die Liste leer ist. Denken Sie daran, den Speicherplatz des Elements wieder freizugeben.

14.4.2. Löschen einer verketteten Liste

Erweitern Sie die Funktionen in dem Listing 14.4 um eine Funktion, welche die gesamte Liste löscht und dabei allen allokierten Speicher freigibt.

14.4.3. Fibonacci-Folge

Schreiben Sie ein Programm, dass die ersten 20 Werte der Fibonacci-Folge berechnet. Lösen Sie das Problem einmal iterativ und einmal rekursiv. Die Fibonacci-Reihe ist definiert durch:

Die Fibonacci-Folge f_1, f_2, f_3, \dots ist durch das rekursive Bildungsgesetz

$$f_n = f_{n-1} + f_{n-2}, \quad \text{wenn } n > 2$$

mit den Anfangswerten $f_1 = f_2 = 1$ definiert.

A. Häufig verwendete Bibliotheksfunktionen

Es werden nicht alle Funktionen aufgelistet, sondern nur oft verwendete.

A. Häufig verwendete Bibliotheksfunktionen

ctype.h	
<i>Name</i>	<i>Beschreibung</i>
isalnum()	isalpha(c) oder isdigit(c) ist wahr
isalpha()	isupper(c) oder islower(c) ist wahr
iscntrl()	Steuerzeichen
isdigit()	dezimale Ziffer
isgraph()	sichtbares Zeichen, kein Leerzeichen
islower()	Kleinbuchstabe (aber kein Umlaut oder ß)
isprint()	sichtbares Zeichen, auch Leerzeichen
ispunct()	sichtbares Zeichen, mit Ausnahme von Leerzeichen, Buchstabe oder Ziffer
isspace()	Leerzeichen, Seitenvorschub (\f), Zeilentrenner (\n), Wagenrücklauf (\r), Tabulatorzeichen (\t), Vertikal-Tabulator (\v)
isupper()	Großbuchstabe (aber kein Umlaut)
isxdigit()	hexadezimale Ziffer
tolower()	wandelt Buchstaben in Kleinbuchstaben um
toupper()	wandelt Buchstaben in Großbuchstaben um

Tabelle A.1.: ctype.h

math.h	
<i>Name</i>	<i>Beschreibung</i>
cos()	Kosinus von x
sin()	Sinus von x
tan()	Tangens von x
acos()	arccos(x)
asin()	arcsin(x)
atan()	arctan(x)
cosh()	Cosinus Hyperbolicus von x
sinh()	Sinus Hyperbolicus von x
tanh()	Tangens Hyperbolicus von x
exp()	Exponentialfunktion (e hoch x)
log()	natürlicher Logarithmus (Basis e)
log10()	dekadischer Logarithmus (Basis 10)
sqrt()	Quadratwurzel von x
pow()	Berechnet x^y

Tabelle A.2.: math.h

time.h	
<i>Name</i>	<i>Beschreibung</i>
asctime()	wandelt die Zeit in einen String
clock()	ungefähre Prozessorzeit, die vom Programm verwendet wird
ctime()	wandelt einen Zeitwert in einen String analog asctime()
difftime()	gibt die Zeit in Sekunden zwischen zwei Zeiten
gmtime()	konvertiert Zeit in UTC
localtime()	konvertiert Zeit in Ortszeit
mktime()	konvertiert Zeit zu einem Zeitwert
strftime()	formatiert Datum und Zeit
time()	gibt die Zeit in Sekunden

Tabelle A.3.: time.h

string.h	
<i>Name</i>	<i>Beschreibung</i>
memcpy()	kopiert n Zeichen
memmove()	wie memcpy, aber auch bei überlappenden Speicherbereichen
memcmp()	vergleicht die Speicherbereiche
memchr()	liefert Pointer auf das erste Vorkommen eines Zeichens
memset()	setzt die ersten n Zeichen auf einen Wert
strcpy()	kopiert einen String
strncpy()	kopiert n Zeichen eines Strings
strcat()	fügt einen String an einen String an
strncat()	fügt n Zeichen an einen String an
strcmp()	vergleicht zwei Strings
strncmp()	vergleicht n Zeichen eines Strings mit einem Anderen
strchr()	liefert Pointer auf das erste Auftreten eines Zeichens
strrchr()	liefert Pointer auf das letzte Auftreten eines Zeichens
strspn()	liefert Anzahl der Zeichen am Anfang eines String, die alle in anderem String enthalten sind
strcspn()	liefert Anzahl der Zeichen am Anfang von String, die alle nicht in anderem String enthalten sind
strpbrk()	liefert Zeiger auf die erste Stelle eines Strings, an dem ein Zeichen eines anderen String erstmals vorkommt
strstr()	liefert Zeiger auf das erste Auftretens eines Strings in einem anderen
strlen()	liefert Länge eines Strings
strerror()	liefert Zeiger auf String, der für einen bestimmten Fehler definiert ist
strtok()	durchsucht einen String nach Zeichenfolgen, dabei enthält ein anderer String die Zeichen, die als Begrenzer verwendet werden sollen

Tabelle A.4.: string.h

A. Häufig verwendete Bibliotheksfunktionen

stdio.h	
<i>Name</i>	<i>Beschreibung</i>

Tabelle A.5.: stdio.h

stdlib.h	
<i>Name</i>	<i>Beschreibung</i>

Tabelle A.6.: stdlib.h

B. ASCII Tabelle - druckbare Zeichen

Der sogenannte *extended ASCII Bereich* befindet sich oberhalb des Zeichenwertes von 127. Wenn Sie die auf Ihrem Rechner verfügbaren Zeichen sehen wollen, so können Sie das mit dem folgenden Listing B.1 erreichen. Es gibt auch die im unteren Bereich bis 32 verfügbaren Zeichen aus.

Listing B.1: Ausgebbare extended ASCII-Zeichen

```
1 /* download print_ascii.c */
2
3 #include <stdio.h>
4
5 void main()
6 {
7     char ch;
8
9     for (int i = 0; i < 256; i++) {
10        if ((i % 4) == 0)
11            printf("\n");
12        switch (i) {
13            case 0:
14                printf("%3i, \u0000", i);
15                break;
16            case 7:
17                printf("%3i, \u0007", i);
18                break;
19            case 8:
20                printf("%3i, \u0008", i);
21                break;
22            case 9:
23                printf("%3i, \u0009", i);
24                break;
25            case 10:
26                printf("%3i, \u000a", i);
27                break;
28            case 13:
29                printf("%3i, \u000d", i);
30                break;
31            default:
32                printf("%3i, \u00%02x", i, i);
33                break;
34        }
```

B. ASCII Tabelle - druckbare Zeichen

Nr.	hex	char	Nr.	hex	char	Nr.	hex	char	Nr.	hex	char
0	0x 0	NUL	1	0x 1		2	0x 2		3	0x 3	
4	0x 4		5	0x 5		6	0x 6		7	0x 7	BEL
8	0x 8	BS	9	0x 9	TAB	10	0x a	LF	11	0x b	
12	0x c		13	0x d	CR	14	0x e		15	0x f	
16	0x10		17	0x11		18	0x12		19	0x13	
20	0x14		21	0x15		22	0x16		23	0x17	
24	0x18		25	0x19		26	0x1a		27	0x1b	
28	0x1c		29	0x1d		30	0x1e		31	0x1f	
32	0x20	SPC	33	0x21	!	34	0x22	"	35	0x23	#
36	0x24	\$	37	0x25	%	38	0x26	&	39	0x27	'
40	0x28	(41	0x29)	42	0x2a	*	43	0x2b	+
44	0x2c	,	45	0x2d	-	46	0x2e	.	47	0x2f	/
48	0x30	0	49	0x31	1	50	0x32	2	51	0x33	3
52	0x34	4	53	0x35	5	54	0x36	6	55	0x37	7
56	0x38	8	57	0x39	9	58	0x3a	:	59	0x3b	;
60	0x3c	<	61	0x3d	=	62	0x3e	>	63	0x3f	?
64	0x40	@	65	0x41	A	66	0x42	B	67	0x43	C
68	0x44	D	69	0x45	E	70	0x46	F	71	0x47	G
72	0x48	H	73	0x49	I	74	0x4a	J	75	0x4b	K
76	0x4c	L	77	0x4d	M	78	0x4e	N	79	0x4f	O
80	0x50	P	81	0x51	Q	82	0x52	R	83	0x53	S
84	0x54	T	85	0x55	U	86	0x56	V	87	0x57	W
88	0x58	X	89	0x59	Y	90	0x5a	Z	91	0x5b	[
92	0x5c	\	93	0x5d]	94	0x5e		95	0x5f	_
96	0x60	`	97	0x61	a	98	0x62	b	99	0x63	c
100	0x64	d	101	0x65	e	102	0x66	f	103	0x67	g
104	0x68	h	105	0x69	i	106	0x6a	j	107	0x6b	k
108	0x6c	l	109	0x6d	m	110	0x6e	n	111	0x6f	o
112	0x70	p	113	0x71	q	114	0x72	r	115	0x73	s
116	0x74	t	117	0x75	u	118	0x76	v	119	0x77	w
120	0x78	x	121	0x79	y	122	0x7a	z	123	0x7b	{
124	0x7c		125	0x7d	}	126	0x7e	~	127	0x7f	DEL

Tabelle B.1.: ASCII-Codes für druckbare Zeichen

```
35         ch++;
36     }
37     return;
38 }
```


Listings

4.1. Einfaches Eingabemenü	32
4.2. Aufbau der switch-Anweisung	34
4.3. Beispiel für isdigit() und isalpha()	35
4.4. Zahlenratespiel	36
5.1. Beispiel für prefix- und postfix-Inkrement	40
5.2. Beispiel für eine while-Schleife	41
5.3. Beispiel für eine kontrollierte Endlosschleife	42
5.4. Auswahlmenü mit do...while	43
5.5. Schleife mit Namensausgabe mittels while	44
5.6. Schleife mit Namensausgabe mittels for	45
5.7. Konzentrationsspiel	45
6.1. Beispiel für Funktionsaufrufe	51
6.2. Beispiel für lokale Variablen	53
6.3. Weiteres Beispiel für lokale Variablen	54
6.4. Beispiel für globale Variable	55
7.1. Zugriff auf einzelne Felder eines Arrays	58
7.2. Umgang mit zweidimensionalen Arrays	61
7.3. Tic-Tac-Toe	61
7.4. Finde den Fehler	65
8.1. Beispiel für indirekte Zuweisung	68
8.2. Indirekte Zuweisung mit Pointern	69
8.3. Übergabe von Argumenten durch <i>passing by value</i>	69
8.4. Übergabe von Argumenten durch <i>passing by reference</i>	70
8.5. Verändern der Inhalte eines Arrays durch eine Funktion	72
8.6. Versuch die Inhalte eines const Arrays zu ändern	72
8.7. Programm um ein Wort zu verschlüsseln und entschlüsseln	74
9.1. Bedeutung des abschließenden \0 Zeichens	78
9.2. Konvertieren eines Strings in eine Zahl	79
9.3. Umwandlung von Strings in Groß- oder Kleinbuchstaben	80
9.4. Kopieren eines Strings	82
9.5. Anfügen eines Strings an einen anderen String	82
9.6. Vergleichen von Strings	83

9.7. Suchen eines Strings in einem anderen String	84
9.8. Wortsuch-Spiel	85
10.1. Einfache Anwendung von malloc()	89
10.2. Anwendung von free()	91
10.3. Beispiel für bedarfsgerechtes Speicherallokieren	92
10.4. Dynamisches Wachsen eines Arrays mit realloc()	94
11.1. Verwendung von fgets()	101
11.2. Verwendung von fputs()	102
11.3. Verwendung von fprintf()	103
11.4. Verwendung von fscanf()	104
12.1. Übergabe einer Struktur by value	108
12.2. Übergabe einer Struktur by reference	109
12.3. Beispiel für eine einfach verkettete Liste	111
13.1. Listing mit Laufzeitfehlern	115
13.2. Fehlersuche mit printf()	116
13.3. Nutzung des Präprozessors zur Fehlersuche	118
13.4. assert zur Fehlersuche	120
13.5. Fehlerbehandlung mittel strerror() und perror()	122
13.6. Fehlerbehandlung der Division durch 0	123
14.1. Iterative Berechnung der Fakultät	125
14.2. Rekursive Berechnung der Fakultät	126
14.3. Lösung der Türme von Hanoi	127
14.4. Sortierte, verkettete Liste	128
B.1. Ausgebbare extended ASCII-Zeichen	139