

*Ralf Salomon*

# Lösungsbuch zu den Übungen zur Einführung in die Praktische Informatik

Eine Einführung  
speziell für Elektrotechniker und andere Ingenieure

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

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

```
{
```

```
    printf( "Congratulations to this book. Before \n" );
```

```
    printf( "looking at a sample, make sure          \n" );
```

```
    printf( "    You have read the exercises          \n" );
```

```
    printf( "    You have solved them on your own    \n" );
```

```
    printf( "    You have tested your own programs \n" );
```

```
    printf( "Ok, now you are ready to study them! \n" );
```

```
}
```

## **Ralf Salomon**

Begleitendes Übungsmaterial zum Skript:

Praktische Informatik und die Programmiersprache C

Eine Einführung speziell für Elektrotechniker und andere Ingenieure

ISBN 978-3-00-047149-0

Copyright © Ralf Salomon, 18119 Rostock, 2014

Alle Rechte vorbehalten. Nachdruck, Übersetzung, Vortrag, Reproduktion, Vervielfältigung auf fotomechanischen oder anderen Wegen sowie Speicherung in elektronischen Medien auch auszugsweise nur mit ausdrücklicher Genehmigung des Autors gestattet.

Die in diesem Werk wiedergegebenen Gebrauchsmuster, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Haftungsausschluss: Weder die Autoren noch sonstige Stellen sind für etwaige Schäden, die aus der Verwendung der in diesem Dokument enthaltenen Informationen resultieren, verantwortlich.

Der Autor haftet nicht für die Inhalte der in diesem Buch angegebenen Web-Links und macht sich diese Inhalte nicht zu eigen.

Satz: Ralf Salomon

Druck und Verarbeitung: Westarp & Partner Digitaldruck Hohenwarsleben UG

Printed in Germany

Umwelthinweis: Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

Email: [ralf.salomon@uni-rostock.de](mailto:ralf.salomon@uni-rostock.de)

Web: [www.imd.uni-rostock.de/ma/rs](http://www.imd.uni-rostock.de/ma/rs)

## Danksagung und allgemeine Referenzen

Das hier vorliegende Manuskript beinhaltet 33 einzelne Übungspakete. Diese dienen der Vertiefung sowie dem praktischen Üben des Stoffes, der im vorlesungsbegleitenden Skript *Praktische Informatik und die Programmiersprache C* behandelt wird. Die Stoffauswahl, die Reihenfolge der einzelnen Aspekte und die gewählte Abstraktionsebene sind in erster Linie auf den Kenntnisstand und die späteren Programmierbedürfnisse unserer Elektrotechnikstudenten des ersten Semesters zugeschnitten.

Bei der Erstellung, dem Design und der Korrektur der Übungspakete haben viele Personen geholfen. In erster Linie sind die Mitarbeiter Enrico Heinrich, Matthias Hinkfoth, Ralf Joost, Ralf Warmuth und René Romann zu nennen, die durch ihre Anmerkungen, Vorschläge, Hinweise und teils sehr intensive und kontroverse Diskussionen zum Gelingen maßgeblich beigetragen haben. Also Jungs, vielen Dank!

Desweiteren sind alle Studenten zu nennen, die dem Fehlerteufel zu Leibe gerückt sind. Großen Dank an euch alle. Besonders hervorzuheben ist hier die Arbeit von Andrea Dorn, die das Manuskript noch einmal sehr gründlich gelesen und korrigiert hat.

Eine wesentliche Voraussetzung für das Erstellen der einzelnen Übungsaufgaben war die sehr inspirierende Arbeitsumgebung, die ich in den Sommermonaten der Jahre 2013 und 2014 auf Maui vorfand. Daher gebührt großer Dank all den anonymen Beschäftigten folgender Lokalitäten, die mich unwissentlich vor allem durch Kaffee, Musik und ihren Spirit unterstützt haben: Denny's at South Kihei, Paia Bay Cafe at Paia, Starbucks at South Kihei und Starbucks at Kahului. Thanks guys, your help is highly appreciated!

Keine der hier vorzufindenden Übungsaufgaben ist bewusst irgendwo abgeschrieben worden. Natürlich kann es bei aller Sorgfalt sein, dass es in der Literatur ähnliche Aufgaben, Quizfragen oder Programme gibt. Doch sind diese Ähnlichkeiten reiner Zufall oder Resultat von mehr als 25 Jahren Programmier- und Lehrerfahrung, in denen man viel programmiert hat und vor allem auch Open-Source Quellen sieht, an die man sich heute nicht mehr erinnert.

Natürlich haben auch einige Bücher ihre unverkennbaren Spuren hinterlassen. Hier sind insbesondere das Buch über Algorithmen und Datenstrukturen, von Niklaus Wirth [4], die Einführung in das Software Engineering von Kimm et al. [3], das Originalbuch über die Programmiersprache C von Kernighan und Ritchie [2] sowie das Buch über das Unix Betriebssystem von Maurice Bach [1] zu nennen.

# Inhaltsverzeichnis

## Vorwort

<b>1</b>	<b>Grundlagen: von der Hardware zum Programmieren</b>	
	Stoffwiederholung . . . . .	1-1
	Quiz . . . . .	1-4
	Fehlersuche . . . . .	1-5
	Anwendungen . . . . .	1-8
<b>2</b>	<b>Erste Kontakte mit dem PC</b>	
	Motivation: Warum das alles ...? . . . . .	2-1
	Kommandoeingabe unter Linux . . . . .	2-2
	Kommandoeingabe unter Windows . . . . .	2-8
	Die wichtigsten Befehle . . . . .	2-13
<b>3</b>	<b>Mein erstes Programm: Fläche eines Rechtecks</b>	
	Stoffwiederholung . . . . .	3-1
	Quiz . . . . .	3-3
	Fehlersuche . . . . .	3-4
	Anwendungen . . . . .	3-5
<b>4</b>	<b>Klassifikation von Dreiecken</b>	
	Stoffwiederholung . . . . .	4-1
	Quiz . . . . .	4-3
	Fehlersuche . . . . .	4-4
	Anwendungen . . . . .	4-5
<b>5</b>	<b>Abstrakte Programmierung</b>	
	Stoffwiederholung . . . . .	5-1
	Quiz . . . . .	5-3
	Fehlersuche . . . . .	5-4
	Anwendungen . . . . .	5-5
<b>6</b>	<b>Arbeiten mit Syntaxdiagrammen</b>	

	Stoffwiederholung . . . . .	6-1
	Quiz . . . . .	6-2
	Fehlersuche . . . . .	6-5
	Anwendungen . . . . .	6-6
<b>7</b>	<b>Angemessenes Formatieren von C-Programmen</b>	
	Stoffwiederholung . . . . .	7-1
	Quiz . . . . .	7-3
	Fehlersuche . . . . .	7-4
	Anwendungen . . . . .	7-5
<b>8</b>	<b>Datentyp int</b>	
	Stoffwiederholung . . . . .	8-1
	Quiz . . . . .	8-3
	Fehlersuche . . . . .	8-4
	Anwendungen . . . . .	8-5
<b>9</b>	<b>Logische Ausdrücke</b>	
	Stoffwiederholung . . . . .	9-1
	Quiz . . . . .	9-2
	Fehlersuche . . . . .	9-3
	Anwendungen . . . . .	9-4
<b>10</b>	<b>Fallunterscheidungen</b>	
	Stoffwiederholung . . . . .	10-1
	Quiz . . . . .	10-3
	Fehlersuche . . . . .	10-5
	Anwendungen . . . . .	10-7
<b>11</b>	<b>Schleifen</b>	
	Stoffwiederholung . . . . .	11-1
	Quiz . . . . .	11-3
	Fehlersuche . . . . .	11-6
	Anwendungen . . . . .	11-7
<b>12</b>	<b>Der Datentyp char</b>	
	Stoffwiederholung . . . . .	12-1
	Quiz . . . . .	12-2
	Fehlersuche . . . . .	12-3
	Anwendungen . . . . .	12-4
<b>13</b>	<b>Der Datentyp double</b>	
	Stoffwiederholung . . . . .	13-1
	Quiz . . . . .	13-2

	Fehlersuche . . . . .	13-3
	Anwendungen . . . . .	13-4
<b>14</b>	<b>Eindimensionale Arrays</b>	
	Stoffwiederholung . . . . .	14-1
	Quiz . . . . .	14-2
	Fehlersuche . . . . .	14-3
	Anwendungen . . . . .	14-4
<b>15</b>	<b>Einfaches Sortieren, Suchen und Finden</b>	
	Stoffwiederholung . . . . .	15-1
	Quiz . . . . .	15-3
	Fehlersuche . . . . .	15-4
	Anwendungen . . . . .	15-5
<b>16</b>	<b>Gemischte Datentypen</b>	
	Stoffwiederholung . . . . .	16-1
	Quiz . . . . .	16-2
	Fehlersuche . . . . .	16-3
	Anwendungen . . . . .	16-4
<b>17</b>	<b>Der gcc Compiler</b>	
	Stoffwiederholung . . . . .	17-1
	Quiz . . . . .	17-3
	Fehlersuche . . . . .	17-6
	Anwendungen . . . . .	17-7
<b>18</b>	<b>Ausdrücke</b>	
	Stoffwiederholung . . . . .	18-1
	Quiz . . . . .	18-3
	Fehlersuche . . . . .	18-6
	Anwendungen . . . . .	18-7
<b>19</b>	<b>Programmieren eigener Funktionen</b>	
	Stoffwiederholung . . . . .	19-1
	Quiz . . . . .	19-6
	Fehlersuche . . . . .	19-9
	Anwendungen . . . . .	19-10
<b>20</b>	<b>Zeiger und Zeigervariablen</b>	
	Stoffwiederholung . . . . .	20-1
	Quiz . . . . .	20-3
	Fehlersuche . . . . .	20-6
	Anwendungen . . . . .	20-7

<b>21 Funktionen mit Zeigern und Arrays als Parameter</b>	
Stoffwiederholung . . . . .	21-1
Quiz . . . . .	21-3
Fehlersuche . . . . .	21-4
Anwendungen . . . . .	21-5
<b>22 Rekursive Funktionsaufrufe</b>	
Stoffwiederholung . . . . .	22-1
Quiz . . . . .	22-2
Fehlersuche . . . . .	22-5
Anwendungen . . . . .	22-6
<b>23 Mehrdimensionale Arrays</b>	
Stoffwiederholung . . . . .	23-1
Quiz . . . . .	23-2
Fehlersuche . . . . .	23-3
Anwendungen . . . . .	23-4
<b>24 Zeichenketten</b>	
Stoffwiederholung . . . . .	24-1
Quiz . . . . .	24-3
Fehlersuche . . . . .	24-4
Anwendungen . . . . .	24-5
<b>25 Kommandozeilenargumente</b>	
Stoffwiederholung . . . . .	25-1
Quiz . . . . .	25-4
Fehlersuche . . . . .	25-5
Anwendungen . . . . .	25-6
<b>26 Der Datentyp struct</b>	
Stoffwiederholung . . . . .	26-1
Quiz . . . . .	26-4
Fehlersuche . . . . .	26-6
Anwendungen . . . . .	26-7
<b>27 Definition eigener Datentypen</b>	
Stoffwiederholung . . . . .	27-1
Quiz . . . . .	27-2
Fehlersuche . . . . .	27-5
Anwendungen . . . . .	27-6
<b>28 Module und getrenntes Übersetzen</b>	
Stoffwiederholung . . . . .	28-1

	Quiz . . . . .	28-2
	Fehlersuche . . . . .	28-3
	Anwendungen . . . . .	28-4
<b>29</b>	<b>Dynamische Speicherverwaltung: malloc() und free()</b>	
	Stoffwiederholung . . . . .	29-1
	Quiz . . . . .	29-3
	Fehlersuche . . . . .	29-4
	Anwendungen . . . . .	29-5
<b>30</b>	<b>Kopieren von Dateien</b>	
	Stoffwiederholung . . . . .	30-1
	Quiz . . . . .	30-3
	Fehlersuche . . . . .	30-6
	Anwendungen . . . . .	30-7
<b>31</b>	<b>Entwicklung eines einfachen Kellerspeichers (Stacks)</b>	
	Stoffwiederholung . . . . .	31-1
	Quiz . . . . .	31-3
	Fehlersuche . . . . .	31-5
	Anwendungen . . . . .	31-6
<b>32</b>	<b>Einfach verkettete, sortierte Liste</b>	
	Stoffwiederholung . . . . .	32-1
	Quiz . . . . .	32-3
	Fehlersuche . . . . .	32-9
	Anwendungen . . . . .	32-10
<b>33</b>	<b>Binäre Bäume</b>	
	Stoffwiederholung . . . . .	33-1
	Quiz . . . . .	33-2
	Fehlersuche . . . . .	33-3
	Anwendungen . . . . .	33-5
<b>Anhänge</b>		
	Anhang I: Anleitung zu den Übungspaketen . . . . .	iii
	Anhang II: Präzedenztabelle . . . . .	vi
	Anhang III: Literaturverzeichnis . . . . .	vii



# Vorwort

*Ahhh, endlich mal ein Lösungsbuch. Das macht die Arbeit einfach, so wünscht man sich das als Student. Weiter so :-)* Schon mal ein ganz falscher Start. *Wieso? Wir haben die Aufgaben und hier die Lösungen. Da können wir uns doch einfach letztere anschauen und wissen Bescheid. Bei dem ausführlichen Material ist dann auch die Prüfung ein Klacks. Oh, schon der zweite Fehlstart. Hä? Hat Deine Platte einen Sprung oder reden wir aneinander vorbei? Ist doch supi!?* Hier steht alles Wichtige. Schnell gelernt und schwups zur Prüfung. *Da habe ich endlich Zeit für die wichtigen Dinge wie Freizeit, Kneipe, Fußball, Party und so Sachen...* Ich sage ja, ein klassischer Fehlstart.

Lass uns doch noch mal in aller Ruhe von vorne anfangen. Ja, das ist hier ein Lösungsbuch. Und um euch die Arbeit möglichst einfach zu machen, sind sogar noch alle Übungsaufgaben mit abgedruckt. *Sag' ich doch. Easy!* Eben nicht. Es geht darum, dass ihr etwas lernt. Dazu müsst ihr *selbstständig* üben. *Ja ja. Soll das heißen, dass in diesem Buch falsche Lösungen stehen?* Nein, natürlich nicht, sie sind korrekt. *Na also, dann reicht es doch, wenn ich mir die Lösungen anschau.* Anschließend weiß ich doch alles Wichtige. Eben nicht!

Das Wichtige sind nicht die Lösungen sondern der Weg dorthin. Bisher sind alle Studenten durchgefallen, die ganz „schlau“ mit den Musterlösungen angefangen haben. *Ja ja, das sagst du jetzt ja nur so. Oder vielleicht hatten sie auch alle Alzheimer, soll ja manchmal auch früher anfangen...* Nein, sie sind *alle* durchgefallen. Das ist keine Übertreibung sondern Realität. Der Grund hierfür ist ganz einfach: Jede Klausur ist neu, wir übernehmen keine Aufgabe aus früheren. Also muss man die benötigten Lösungswege selbst finden. Und genau das lernt man nicht, wenn man gleich in das Lösungsbuch schaut. Man muss üben, den richtigen Ansatz und den richtigen Weg zu finden. Daher muss man sich selbst durch das Übungsbuch arbeiten.

*Ach menno. Wozu gibt's denn dann das Lösungsbuch?* Ja, hierfür gibt es die folgenden Gründe: Erstens, die hier vorgestellten Lösungen sollen euch *einen* möglichen Lösungsweg zeigen, den ihr mit eurem vergleichen könnt. Zweitens, könnt ihr bei den Quizaufgaben sehen, ob ihr die Situation richtig analysiert habt. Und drittens, sollen andere Lehrer Anregungen erhalten, wie sie ihr eigenes Lehrkonzept gestalten können.

*Ok, ich sag's ja ungerne, aber ich gebe mich geschlagen und fange an richtig zu arbeiten.*

## **33** Übungspakete



# Übungspaket 1

## Grundlagen: von der Hardware zum Programmieren

---

### Übungsziele:

1. Die Bedeutung des Programmierens für mein Studium und meine spätere Berufstätigkeit
2. Was ist eine erfolgreiche Lernstrategie?
3. Die wesentlichen Komponenten des Prozessors

### Skript:

Kapitel: 1 bis 5

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket werden die wesentlichen Grundlagen kurz wiederholt, die für das Programmieren unbedingt benötigt werden. Dazu gehört neben der allgemeinen Motivation und der Bedeutung des Programmierens für das Studium und die spätere Berufstätigkeit auch das Verständnis der Funktionsweise der wesentlichen Hardware-Elemente eines Computers wie Arbeitsspeicher und Prozessor (CPU).

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Fragen zur Motivation

Aus welchen Bestandteilen bestehen die Lehrunterlagen?

Zu den Lehrunterlagen gehören das Skript, die Übungsblätter und sonstige Arbeitsblätter, die früher oder später verwendet werden. Auf den zweiten Blick umfassen die Lehrunterlagen aber ebenfalls die Literatur, die auf der Webseite der Lehrveranstaltung angegeben wird. Und auf den dritten Blick kann eigentlich jede zugängliche Dokumentation zur Programmiersprache C als Lehrunterlage aufgefasst werden, beispielsweise auch die **man-pages** auf den Linuxsystemen oder den Webseiten, die sich der Sprache C widmen. Natürlich fragt man sich als Student: „Wie, dass soll ich alles lesen und zur Klausur lernen?“ Nein, die Klausur prüft, ob der Student ein Verständnis von der Programmiersprache C hat. Sie prüft nicht diese oder jene Kapitel aus dieser oder jener Quelle.

Welche Bedeutung haben die Übungsaufgaben?

Die Aufgaben dienen der Anwendung und der Vertiefung des behandelten Lehrstoffs. Es geht nicht nur darum, die Aufgabe zu lösen, sondern auch darum, Alternativen zu probieren. Es geht darum, eine gewisse Neugier zu erzeugen: Warum funktioniert das so? Wie sehen mögliche Alternativen aus? Welche Vor-/Nachteile haben diese? Was passiert, wenn ich bestimmte Zeilen ändere?

Welchen Zweck verfolgen die Musterlösungen? Wofür sind sie nicht geeignet?

Der Zweck der Musterlösungen liegt darin, nur eine mögliche Lösungsvariante vorzustellen und eventuell hier und da noch auf ein paar Besonderheiten einzugehen. Niemand sollte auf die Idee kommen, Musterlösungen für die Klausur auswendig zu lernen. Niemand sollte auf die Idee kommen, dass man C anhand der Musterlösungen lernen kann. Richtig geht man vor, wenn man sich *nach* dem Erarbeiten einer eigenen Lösung die Musterlösung anschaut und mit der eigenen vergleicht. Was wurde wo und warum anders gemacht?

Wie sieht eine sinnvolle Lernstrategie aus?

Gemäß obiger Ausführungen sind die einzelnen Schritte wie folgt:

1. Die jeweils *nächste* Vorlesung durch Lesen der entsprechenden Skriptkapitel *vorbereiten*. Siehe dazu auch die entsprechenden Arbeitsblätter.
2. Zur Vorlesung gehen und gut zuhören *sowie mitarbeiten*.
3. Die Vorlesung durch nochmaliges Lesen der entsprechenden Kapitel *nachbereiten*.
4. Die Übungsaufgaben lösen soweit dies ohne Hilfe der Assistenten geht.
5. *Nach* erfolgreichem Lösen der Übungsaufgabe diese mit der Musterlösung vergleichen und dabei die Unterschiede finden, verstehen und beurteilen.

Warum lerne ich ausgerechnet die Programmiersprache C? Welche Bedeutung wird sie in meinem Studium sowie meinem späteren Berufsleben haben?

Grundsätzlich gehört C zu den wichtigsten Programmiersprachen überhaupt. Das ist dadurch begründet, dass C beispielsweise für nahezu alle Mikrokontroller verwendet werden kann. C ist eine hardwarenahe Abstraktion von Assembler (Maschinencode). Das macht C zu *der* Programmiersprache für Elektrotechniker. C deckt das gesamte Spektrum von der hardwarenahen Programmierung kleiner Mikrokontroller bis hin zu komplexen PC-basierten Anwendungen ab. Weiterhin erleichtert ein gutes Verständnis der Programmiersprache C auch den Umstieg auf nahezu jede andere Programmiersprache, wie beispielsweise Java oder C++. Wer diese Punkte verstanden hat, der wird auch die Bedeutung der Programmiersprache C für das Studium und das spätere Berufsleben einschätzen können.

## Aufgabe 2: Hardware

Erkläre in eigenen Worten die folgenden Speicher-Begriffe: Bit, Byte, Wort.

In der Elektronik bezeichnet das *Bit* die kleinste Dateneinheit. Ein Bit kann lediglich die Zustände Null oder Eins annehmen. Dies ist auch die Bedeutung des Bits in der Programmiersprache C. Alle anderen Datenformate wie Zahlen, Zeichen und so weiter sind lediglich definierte *Bitmengen*, die man je nach Verwendung entsprechend interpretiert. Viele Mikrokontroller und PCs betrachten eine Spannung von 0 V als den logischen Wert Null (oder auch „false“ oder „low“) und eine Spannung von beispielsweise 3.3 V als den logischen Wert Eins (alternativ auch „true“ oder „high“). Ein *Byte* ist eine Gruppe, die aus acht Bit besteht. Ein *Wort* ist eine Gruppe von Bytes. Die genaue Anzahl der Bytes ist dabei vom System abhängig. Üblich sind beispielsweise 2 Byte oder 4 Byte.

Benenne die wesentlichen Register eines Prozessors (einer CPU) und erkläre in eigenen Worten deren Bedeutung und Funktionsweise.

Unabhängig von Modell und Hersteller haben wohl alle CPUs die folgenden Register:

1. Im *Befehlsregister* (*instruction register*) steht, welchen Befehl die CPU gerade ausführt. Beispiel: Setze das Programm an Adresse xyz fort.
2. Der *Befehlszähler* (*program counter*, *instruction pointer*) gibt an, wo im Arbeitsspeicher sich der nächste Befehl befindet.
3. Die *Adressregister* (*address register*) enthalten Speicheradressen von Variablen, Zahlen und anderen Werten, auf die die CPU gleich oder in Kürze zugreifen möchte.
4. Das *Stapelregister* (*stack pointer*) ist ein weiteres Adressregister, das die CPU im Wesentlichen für eigene Zwecke benötigt.
5. Die *Datenregister* (*Akkumulatoren*) speichern die Daten und Resultate der arithmetischen und logischen Operationen, die die Recheneinheit (ALU) ausführt.

Die meisten CPUs besitzen noch weitere Register wie Status- und Interruptregister, die aber erst in weiterführenden Lehrveranstaltungen behandelt werden.

Was ist ein Programm und auf welchen „Abstraktionsebenen“ kann man programmieren?

Es lassen sich unter anderem die folgenden drei Ebenen identifizieren:

1. Das *direkte Programmieren* mit Nullen und Einsen im Arbeitsspeicher erfordert explizites Wissen, welche Nullen und Einsen welchen Befehl kodieren. Dies ist in der Regel *sehr* zeitraubend, fehleranfällig und sehr frustrierend.
2. Die Assemblerprogrammierung ist die direkte Programmierung mittels Maschinenbefehlen. Auch hier ist der Aufwand sehr hoch, da komplexe Operationen Schritt für Schritt eigenständig beschrieben werden müssen. Die Assemblerprogrammierung ist nicht plattformunabhängig, wodurch ein Assemblerprogramm in der Regel für jeden neuen Prozessortyp auch wieder neu geschrieben werden muss. Beispielhafte Assemblerbefehle sind: lade Wert 1 in Akkumulator, lade Wert 2 in Akkumulator, addiere Wert 1 und Wert 2, speichere Ergebnis an Adresse xyz.
3. Die Programmiersprache C gehört zu den sogenannten Hochsprachen, auch wenn es noch wesentlich „höhere“ Programmiersprachen wie beispielsweise Pascal, Ada, SQL etc. gibt. C ist weitestgehend plattformunabhängig und erlaubt verständliche Programmstrukturen. Die Sprache verfügt über Schleifen, Verzweigungen und den Aufruf eigener Funktionen.

## Teil II: Quiz

---

Beurteile aus eigener Sicht, wie zutreffend die folgenden Aussagen sind:

Es ist unwichtig, zur Übung zu gehen.

überhaupt nicht zutreffend

Das Ansehen der Musterlösungen ist völlig ausreichend.

überhaupt nicht zutreffend

In meinem späteren Berufsleben spielt das Programmieren keine große Rolle.

überhaupt nicht zutreffend

Die Assistenten korrigieren mir meine Programmierfehler.

überhaupt nicht zutreffend

Es ist völlig überflüssig, die Übungsaufgaben zu Hause vorzubereiten bzw. zu lösen.

überhaupt nicht zutreffend

Die Klausuraufgaben sind jedes Jahr gleich.

überhaupt nicht zutreffend

Es ist ganz besonders hilfreich und sinnvoll, die Aufgaben der letzten Klausuren (auswendig) zu lernen.

überhaupt nicht zutreffend



## Teil III: Fehlersuche

---

Unser Newbie GRAND MASTER C hat unsere Unterlagen durchgearbeitet und sich folgenden „Arbeitsplan“ zurechtgelegt:

1. Programmieren ist cool. Am besten setze ich mich gleich an den Rechner, denn dann habe ich alles zusammen und kann loslegen.
2. Die Musterlösungen sind sehr gut. Statt selbst zu programmieren, arbeite ich diese lieber durch; bringt mehr und ist beides, effizienter und effektiver.
3. Der Besuch der Vorlesungen ist Pflicht, damit mich der Professor kennenlernt und mir eine gute Note gibt.
4. Der Besuch der Übungen ist fakultativ, denn dort sind nur Doktoranden, die keinen Einfluss auf die Note haben.
5. Ich besorge mir die alten Klausuren, denn auch Professoren sind bequem und denken sich keine neuen Aufgaben aus.
6. Ich lass es locker angehen, denn wir haben ja einen Freiversuch.
7. Ich soll das Skript lesen? Keine Zeit, die Dozenten erzählen doch alles in der Vorlesung. . .
8. Die Übungsblätter zu Hause oder gar in der Freizeit bearbeiten? No way! Ist auch quatsch, wir haben doch genügend viele Übungsstunden.
9. Wenn es (zeitlich) eng wird, nehme ich oder besorge mir die Musterlösungen und schreibe einfach ab. Die Doktoranden werden schon ihre eigenen Musterlösungen mögen.

Aus unserer Lehrkörpersicht können wir dieser Arbeitshaltung nicht so recht zustimmen. Finde, erläutere und korrigiere GRAND MASTER C's Missverständnisse auf den folgenden Seiten :-)

Nr.	Kommentierung	Korrigierte Aussage
1.	Einer der Hauptfehler ist es, sich gleich an den Rechner zu setzen. In diesem Fall weiß man noch gar nicht, was man eigentlich programmieren soll, sodass man den Wald vor lauter Bäumen nicht sieht. So scheitert man, obwohl man viele Stunden am Rechner verbringt.	Ich bereite mich mit meinen Studienkollegen gut vor, löse alle Aufgaben auf einem Zettel und setze mich erst dann an den Rechner, um zu programmieren.
2.	Das Abschreiben von Programmen und Musterlösungen aus dem Internet kann man sich sparen, da man dabei so rein gar nichts lernt! Derartige Übungen sind zu 100 % Zeitverschwendung.	Meine Freunde und ich lösen die Aufgaben zusammen. Erst wenn ich diese erfolgreich umgesetzt habe, vergleiche ich meine Lösungen mit den Musterlösungen, um auf Alternativen aufmerksam zu werden.
3.	Der Professor freut sich über jeden Studenten, der zur Vorlesung kommt und beantwortet auch gerne alle Fragen. Es nutzt aber rein gar nichts, sich beim Professor bekannt zu machen, denn es gibt keine Gummipunkte. Die Anwesenheit in der Vorlesung hat keinen Einfluss auf die Prüfungsnote, da diese <i>ausschließlich</i> von der Klausur abhängt.	Ich gehe zur Vorlesung, um die wesentlichen Inhalte nochmals zu hören und ggf. Fragen zu stellen, die ich dort beantwortet bekomme.
4.	Die Teilnahme am Übungsbetrieb ist weit aus wichtiger als das Zeitabsitzen in der Vorlesung. Bei den Übungen können die Doktoranden individuell auf meine Programmierfertigkeiten und -probleme eingehen.	Ich gehe zu den Übungsterminen, um die letzten Fehler meiner Programme zu beseitigen. Bei den Übungsterminen bekomme ich zwar keine fertigen Lösungen kann aber Fragen stellen und bekomme <i>individuelle</i> Hilfestellungen, Hinweise und Anregungen.
5.	Es mag sein, dass es Vorlesungen gibt, die so funktionieren. In der Veranstaltung <i>Einführung in die Praktische Informatik</i> denken sich die Doktoranden und der Professor <i>immer</i> neue Prüfungsfragen aus. Bisher hat es noch keinem Studenten geholfen, die alten Klausur auswendig zu lernen. Nur glauben etwa 50 % der Studenten dies nicht.	Um erfolgreich zu sein, lerne ich und bearbeite die Übungspakete <i>selbstständig</i> .

Nr.	Kommentierung	Korrigierte Aussage
6.	Ja, das erleben wir leider all zu oft. Ja, es gibt einen Freiversuch und ja, ich kann die Klausur öfters wiederholen. Aber, das Studium ist vollgepackt mit Lehrveranstaltungen. Und da bleibt keine Zeit, diese in späteren Semestern zu wiederholen.	Ich lass es nicht locker angehen sondern werde mich jede freie Minute mit dem Programmieren beschäftigen. Ich werde mich auch mit meinen Freunden in der Kneipe verabreden, um einzelne Übungsteile bei einem leckeren Bier gemeinsam zu bearbeiten.
7.	Das ist die ganz falsche Herangehensweise. In den Vorlesungen ist nur Zeit, einige ausgewählte Aspekte zu behandeln. Wenn man sich den Stoff nicht <i>vorher</i> intensiv angeschaut hat, kann man nur schwer folgen und bekommt wenig mit.	Ich werde also die relevanten Skriptteile <i>vor</i> jeder Vorlesung <i>durcharbeiten</i> , um vorbereitet zu sein und dem Dozenten folgen zu können. Ich werde jedesmal mindestens eine Frage vorbereiten; mal sehen, ob er sie beantworten kann ;-)
8.	Man mag es nicht glauben, aber Programmieren ist eine sehr zeitaufwändige Tätigkeit. Dabei vergeht die Zeit wie im Fluge. Wenn ich nicht schon alles zu Hause vorbereitet habe, reicht die Zeit bei weitem nicht aus. Ein Hauptproblem ist es, die Programmierfehler erst einmal zu finden. . .	Die Aufgabenblätter werde ich (zusammen mit meinen Freunden) zu Hause vorbereiten, um die Übungszeit zum Testen meiner Programme zu nutzen.
9.	Das Abschreiben irgendwelcher Lösungen bringt gar nichts! Die Doktoranden interessiert es auch so rein gar nicht, ihre eigenen Lösungen zu sehen, denn sie können schon programmieren. Ferner muss ich in der Klausur in der Lage sein, neue, mir unbekannte Aufgaben unter Zeitdruck selbstständig zu lösen; sonst falle ich <i>gnadenlos</i> durch.	Die vorhandenen Musterlösungen <i>arbeite</i> ich erst durch, wenn ich die Aufgaben (mit Hilfe der Doktoranden) selbst gelöst habe. Bei Unklarheiten frage ich die Doktoranden.

## Teil IV: Anwendungen

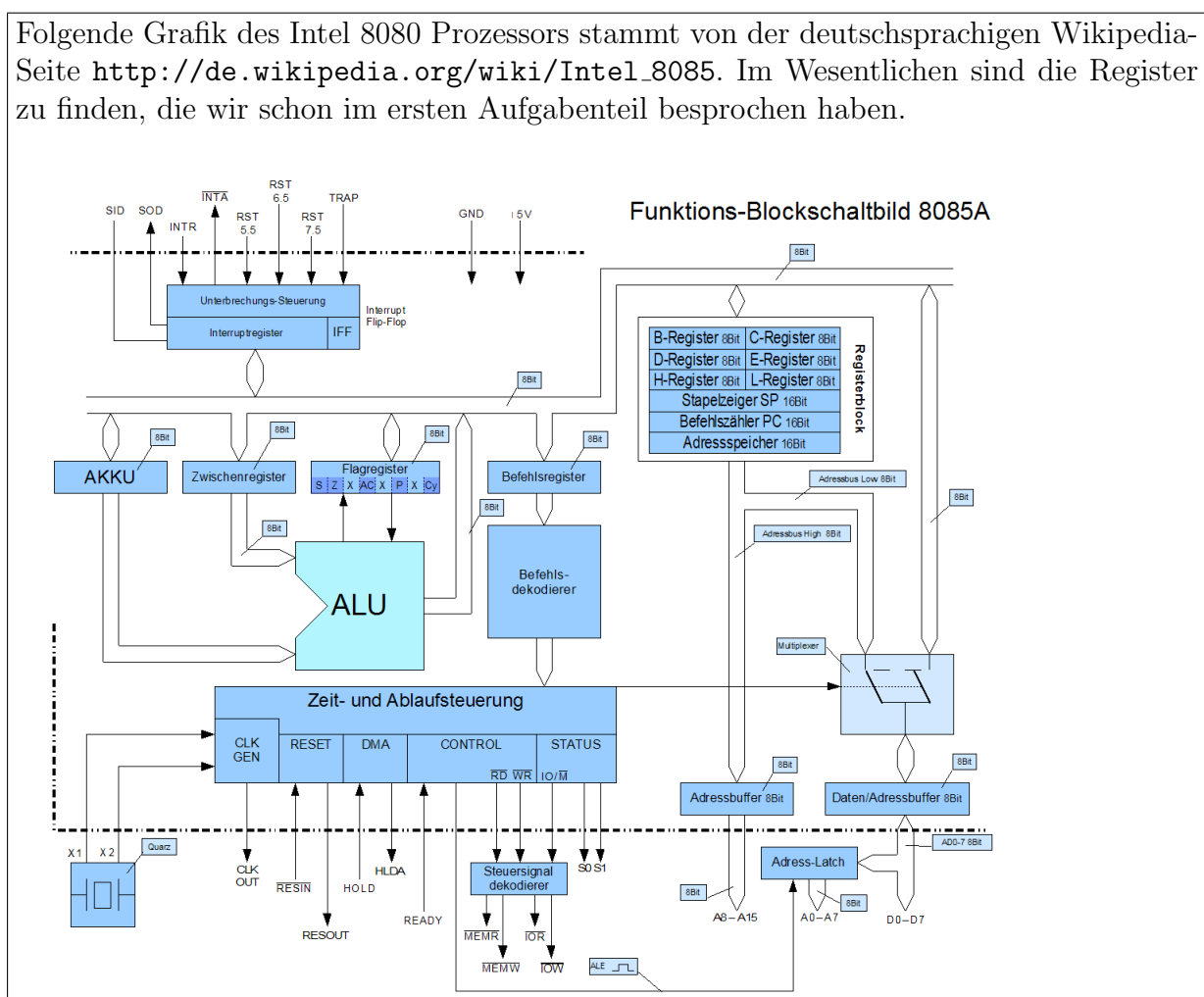
Die folgenden Aufgaben lassen sich mit Hilfe einer einfachen Internet-Recherche relativ einfach lösen.

### Aufgabe 1: CPU Register

In der Vorlesung und Skript (Kapitel 5) haben wir einige Register besprochen, die in *jeder* CPU in der einen oder anderen Form vorhanden sein *müssen*.

**Frage:** Welche Register hat der Intel Prozessor 8085? Recherchiere ein wenig im Internet und beschreibe oder illustriere kurz den Aufbau des Prozessors.

Folgende Grafik des Intel 8080 Prozessors stammt von der deutschsprachigen Wikipedia-Seite [http://de.wikipedia.org/wiki/Intel\\_8085](http://de.wikipedia.org/wiki/Intel_8085). Im Wesentlichen sind die Register zu finden, die wir schon im ersten Aufgabenteil besprochen haben.



## Aufgabe 2: Taktfrequenzen

Ein weiterer interessanter technischer Parameter beschreibt die Geschwindigkeit, mit der eine CPU arbeitet.

**Frage:** Mit welchen Taktfrequenzen arbeiten moderne Prozessoren und deren angeschlossenen Arbeitsspeicher (auch häufig als RAM bezeichnet).

Im April 2013 ergab eine kleine Recherche folgende Taktfrequenzen für handelsübliche Komponenten.

### Prozessoren:

- IBM POWER7 – Quad Core: 4,14 GHz
- Intel Xeon E3-1290 v2 – Quad Core: 3,7 GHz
- AMD FX-8150 – 8-Core: 3,6 GHz

### Übertaktete Prozessoren (Kühlung mittels flüssigem Stickstoff):

- AMD FX-8150 : 9,144 GHz ([www.radeon3d.org/forum/thread-2896.html](http://www.radeon3d.org/forum/thread-2896.html))
- Intel Celeron LGA775 352: 8,505 GHz ([www.hwbot.org](http://www.hwbot.org))
- AMD FX-8350: 8,502GHz ([www.hwbot.org](http://www.hwbot.org))

**Arbeitsspeicher:** Im Mai 2012 wurde DDR4 als neueste Speichertechnologie vorgestellt. DDR4 erlaubt Werte von bis zu 400 MHz für den Speichertakt. DDR4 arbeitet allerdings mit einem als „Prefetch“ bezeichnetem Verfahren. Dabei werden bei einer Leseanforderung acht mal mehr Daten an der Schnittstelle zwischen Speicher und System bereitgestellt, als eigentlich angefordert. Das beruht auf der Annahme, dass die folgenden Daten auch noch benötigt werden könnten. Dadurch ergibt sich ein effektiver Takt von 3,2 GHz. Als Systemtakt zwischen Speicher und CPU ist aber nur die halbe Taktrate, also 1,6 GHz notwendig, da Daten zwei mal pro Takt (sowohl bei der fallenden als auch bei der steigenden Taktflanke) übertragen werden.

## Aufgabe 3: Speichermedien

Wie groß ist heute üblicherweise der Arbeits- und Plattenspeicher moderner PCs?

Im Frühjahr 2013 hatte jeder Spiele-PC, der etwas auf sich hält, mindestens 16 GB Arbeitsspeicher. Als Größe für Plattenspeicher (Einzellaufwerke) haben sich Werte im unteren einstelligen Terrabyte (TB) etabliert, die Topmodelle (beispielsweise Seagate Constellation) bringen 4 TB mit.

# Übungspaket 2

## Erste Kontakte mit dem PC

---

### Übungsziele:

1. Sachgerechter Umgang mit dem PC und seinem Betriebssystem
2. Navigation im Dateisystem
3. Einrichten von Unterverzeichnissen (directories)
4. Erstellen von Dateien

### Skript:

Wird nicht im Skript besprochen

Weitere Hinweise und Unterlagen findet man wie folgt:

Linux:        Beispielsweise: Cameron Newham & Bill Rosenblatt, *Learning the bash*, O'Reilly Verlag

Windows: Die folgende Google-Anfrage bringt einige nützliche Hinweise: `windows kommandozeile einführung`

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket sollt ihr lernen, sachgerecht mit dem PC umzugehen. Dazu gehören insbesondere das Anlegen von Unterverzeichnissen und Dateien sowie das Navigieren im Dateisystem. Ferner zeigt dieses Übungspaket, wie man sich *online* Hilfe besorgen kann. All das machen wir *nicht* mit „Klickybunt“ sondern schön traditionell mit der Kommandoeingabe, die auch als Konsole bezeichnet wird.

# Teil I: Motivation: Warum das alles ...?

---

Jeder von euch hat sicher schon davon gehört, dass es Dinger gibt, die Rechner heißen und manchmal auf Neudeutsch auch Computer genannt werden :-)

Manche von euch werden auch schon mal einen gesehen haben. Diejenigen, die schon mal einen Rechner benutzt haben, werden gemerkt haben, dass der Rechner ziemlich schnell ist. Aber nicht nur das: Manchmal macht der Rechner (ziemlich schnell) nicht das, was er soll. Das hat schon manchen Nutzer zu Verzweiflungstaten getrieben, z.B.:

1. Programm neu starten,
2. Rechner neu starten,
3. Betriebssystem neu installieren,
4. Rechner aus dem Fenster werfen
5. und das Schlimmste: mit der Tastatur auf den Monitor einprügeln; die beiden können nun wirklich nichts dafür.

Doch das muss nicht sein! Auch wenn es noch so unglaublich klingen mag: Ein Rechner ist deterministisch. D.h. solange kein Hardware-Defekt vorliegt, macht er *immer genau das Richtige*. Daraus folgt eine tragische Offensichtlichkeit: Der „Fehler“ sitzt meist vor dem Rechner, entweder als Programmierer, oder als Nutzer. Für euch stellt sich also nicht die Frage: Wie programmiere ich so, dass der Rechner das Richtige macht? Denn das macht er sowieso. Die Frage lautet vielmehr: Wie muss ich programmieren, dass der Rechner tut, was ich erwarte?

Und nun kommen wir zu einem richtig großen, wenn nicht riesigen Problem: Woher wisst ihr, was ihr vom Rechner erwarten könnt? Er spricht ja nicht mit euch. Mitdenken ist ihm auch nicht gegeben. Mitfühlen schon gar nicht. Die schreckliche Wahrheit ist, dass ihr die grundsätzliche Funktionsweise eines Rechners verstehen müsst, um diesen Rechner sinnvoll programmieren zu können. Wir werden also erstmal ziemlich weit ausholen, und ganz unten anfangen.

*Jeder, der bis hierhin etwas nicht verstanden hat, oder etwas anders sieht, bespreche das jetzt mit dem Lehrpersonal. Das ist Voraussetzung für das Bearbeiten der Übungsaufgaben, und genau jetzt ist der richtige Zeitpunkt.*

In den beiden folgenden Abschnitten beschreiben wir einige wichtige Kommandos, getrennt für Linux und Windows. Der vierte Teil dieses Übungspakets fasst die wichtigsten Kommandos noch einmal zusammen.

# Teil II: Kommandoeingabe unter Linux

---

## 1 Arbeiten mit der Kommandozeile

Die Kommandozeile, auch als Eingabeaufforderung oder Shell (engl. für Schale bzw. Oberfläche) bekannt, stellt eine Möglichkeit des Umgangs mit dem Computer dar. Eine andere Möglichkeit hat zwei bis 20 Tasten und leuchtet hübsch an der Unterseite. Da ihr die Maus und grafische Oberflächen schon kennt, und wir euch nicht langweilen wollen, lernt ihr in diesem Übungspaket etwas Neues kennen: die Kommandozeile.

**Die Shell:** Um mit der Kommandozeile zu arbeiten, müsst ihr sie erstmal finden. Um nämlich dem normalen Nutzer das Leben zu vereinfachen, haben die Betriebssystementwickler die Shell meist gut versteckt. Eine Möglichkeit sieht wie folgt aus:

**Aufruf:** `Strg+Alt+T`

Es öffnet sich ein neues Fenster. Das sieht aber schon auf den ersten Blick ganz anders aus als alle anderen Fenster. In dem Fenster blinkt was rum, der Rest ist „Voodoo“.

Was sagt uns das? Dass da was blinkt ist gut. Was können wir machen? Wenn wir jetzt auf der Tastatur rumhacken, erscheinen die Zeichen genau dort wo es blinkt, und es blinkt rechts daneben. Tippen wir an dieser Stelle etwas Sinnvolles ein und drücken Enter, so passiert etwas Sinnvolles.

**Der Editor:** Der Editor ist ein (Hilfs-) Programm, mit dessen Hilfe wir Dateien erstellen und verändern können. Ein einfacher Editor heißt `pluma`.

**Aufruf:** `pluma`

Zu unserer Überraschung ist ein Editor aufgegangen. Wenn ihr das Programm beendet, könnt ihr hier weitermachen. Zuerst schauen wir mal, in welchem Ordner wir uns befinden.

**Navigation im Dateisystem:** Um sich im Dateisystem zurecht zu finden, benötigen wir eigentlich drei Kommandos.

**Wo sind wir? Kommando:** `pwd`

Dieser Ordner enthält vermutlich Dateien. Auch das gucken wir uns mal an.

**Was finden wir da, wo wir sind? Kommando:** `ls`

Natürlich seht ihr jetzt keine Dateien, sondern nur die Dateinamen. Alles andere wäre ziemlich unübersichtlich. Falls die Liste länger ist als der Bildschirm, können wir ein kleines Problem haben, da die Anzeige schnell verschwindet und man nicht alles



sieht. Da dieses Problem relativ häufig ist, gibt es eine Möglichkeit der seitenweisen Anzeige. Tippt nach dem Befehl einfach einen senkrechten Strich und `more` ein:

**Kommando:** `<befehl> | more`

**Beispiel:** `ls | more`

So, jetzt haben wir schon eine Menge Dateien gesehen. Sie befinden sich alle im aktuellen Verzeichnis. Lustigerweise gibt es noch mehrere Verzeichnisse. Eine Änderung des aktuellen Verzeichnisses ist mit `cd` möglich. Sucht euch einfach mal eines aus.

`cd <Verzeichnisname>`

Vielleicht wollt ihr wieder zurück in das vorherige Verzeichnis.

`cd ..`

**Dateien im Detail:** Jetzt, da wir Verzeichnisse beherrschen, können wir uns mal die Dateien genauer anschauen. Erstmal schauen wir, ob eine bestimmte Datei namens `datei.test` im aktuellen Ordner vorhanden ist.

**Kommando:** `ls datei.test`

Das ist sie nicht. Na dann erstellen wir sie doch einfach.

**Kommando:** `touch datei.test`

Jetzt gibt es eine Datei namens `datei.test`. Glaubt ihr nicht? Schaut doch nach!

**Kommando:** `ls datei.test`

Toll! Und was ist drin?

**Kommando:** `cat datei.test`

Natürlich nichts. Aber wie bekommen wir diese Datei jetzt wieder weg?

**Kommando:** `rm datei.test`

Und jetzt ist sie weg? Einfach mal nachschauen!

**Kommando:** `ls datei.test`

Diese Datei ist jetzt wirklich weg. So richtig. Nix Papierkorb oder so. Weg!

**Verzeichnisse:** Wir können jetzt Dateien erstellen. Geht das vielleicht auch mit Ordnern? Erstmal schauen, ob es so einen Ordner schon gibt.

**Kommando:** `ls ordner`

Gibt's nicht? Na dann erstellen wir ihn doch.

**Kommando:** `mkdir ordner`

So, wie können wir das jetzt überprüfen? Ganz einfach:

**Kommando:** `ls -d ordner`

Jetzt gibt es also einen leeren Ordner **ordner**. Um jetzt verschiedene Ordner gleichen Namens auseinander zu halten, lassen wir uns mal den gesamten Pfad anzeigen

**Kommando:** `pwd`

Weil wir jetzt einen Ordner erstellt haben, können wir in das neue Verzeichnis wechseln.

**Kommando:** `cd ordner`

Um zu gucken wo wir uns befinden:

**Kommando:** `pwd`

Jetzt können wir mal nachschauen, was in diesem Ordner drin ist.

**Kommando:** `ls`

Natürlich nix. Wir haben den Ordner ja auch gerade erst erstellt, und niemanden sonst an den Rechner rangelassen. Kommen wir eigentlich wieder aus dem Ordner raus?

**Kommando:** `cd ..`

Mal schauen, ob's geklappt hat:

**Kommando:** `pwd`

Unser Ordner ist aber noch da.

**Kommando:** `ls -d ordner`

Aber gleich nicht mehr.

**Kommando:** `rmdir ordner`

Jetzt ist er wirklich weg?

**Kommando:** `ls -d ordner`

Nix zu sehen. Können wir denn noch in das Verzeichnis wechseln?

**Kommando:** `cd ordner`

Wir können also in der Kommandozeile Ordner erstellen, in ihnen herumnavigieren, und Dateien erstellen und löschen. Das ist zwar schön, aber für sich genommen nicht besonders sinnvoll. Also schauen wir uns mal an, wie man Programme startet.

## 2 Sinnvolle Arbeiten mit der Kommandozeile

**Web-Browser:** Wenn wir eine Internet-Seite anschauen wollen, brauchen wir einen Browser. Starten kann man diesen, indem man einfach den Programmnamen in der Kommandozeile eintippt.

```
/usr/bin/x-www-browser
```

Den Browser kennt ihr, hier gibt es also nichts mehr zu erklären.

Da ihr im Browser häufig eine Internetseite anschauen wollt, könnt ihr sie auch mit angeben:

```
/usr/bin/x-www-browser "http://xkcd.com/"
```

Alles was ihr nach dem Programmnamen schreibt, sind Programmparameter. Mit diesen Parametern könnt ihr offensichtlich das Verhalten des Programms beeinflussen. Meist werden Parameter durch Leerzeichen getrennt.

**Variablen:** Wirklich schön ist der Aufruf aber erst, wenn man nicht den kompletten Programmpfad eintippen muss. Dafür gibt es verschiedene Mechanismen, unter Linux eine Umgebungsvariable PATH, unter Windows ebenfalls PATH und zusätzlich die Registry. Das Ergebnis ist das gleiche.

```
x-www-browser
```

Um ein eigenes Programm, beispielsweise `programm`, im aktuellen Verzeichnis zu starten, reicht folgendes:

```
./programm
```

Falls ihr mal ein Programm abbrechen wollt, das nichts sinnvolles mehr macht, funktioniert das mit der Tastenkombination **Strg+C**.

**Abbruch von Programmen:** Strg+C .

Das könnt ihr schnell mal ausprobieren. Startet ein Programm, Sobald ihr **Strg+C** drückt, wird es beendet.

**Mehr zu Dateien:** Da man zum Programmieren irgendwie Quelltext schreiben muss, gibt es Editoren, mit denen man das relativ bequem machen kann.

**Kommando:** `pluma &`

Jetzt könnt ihr fleißig Tasten drücken. Das Ergebnis lässt sich als Datei abspeichern. Gebt ihr einen schönen Namen, z.B. `text.txt`.

An der Kommandozeile könnt ihr euch die Datei mittels des Befehls `cat` anschauen. Interessanterweise kann das Programm auch mehrere Dateien hintereinander ausgeben:

**Kommando:** `cat datei1 datei2 datei3`

Ganz hübsch soweit. Der Inhalt der Dateien wird an der Kommandozeile angezeigt. Falls man diesen Inhalt in eine Datei schreiben möchte, kann man Folgendes eingeben.

**Kommando:** `cat datei1 datei2 datei3 > ziel`

So lassen sich einfach mehrere Dateien zusammenfügen. Der Mechanismus dahinter heißt Ausgabeumleitung. Das Zeichen `>` ist dafür da.

Wenn man nicht immer einen grafischen Texteditor öffnen möchte, um einfache Veränderungen an seinen Texten durchzuführen, gibt es Programme, mit denen man das automatisieren kann. Um beispielsweise in einem Text alle Jahreszahlen 2011 durch 2012 zu ersetzen, genügt `sed 's/2011/2012/g' alt.txt > neu.txt`

**Ausschalten des Rechners:** Wenn es mal wieder so weit ist, den Rechner neu zu starten, dann genügt:

**Kommando:** `sudo shutdown -r now`

**Multimedia:** mp3-Dateien können im aktuellen Verzeichnis wie folgt angehört werden:

**Kommando:** `mpg123 *.mp3`

Ein Video kannst du mit dem Programm `mplayer` anschauen:

**Kommando:** `mplayer sonneborn.mp4`

### 3 Kommandozeile für den C-Programmierer

Jetzt könnt ihr mal den Quelltext aus der Vorlesung eintippen. Wenn ihr damit fertig seid, müsst ihr den Quelltext abspeichern. Genau genommen erstellt ihr eine Datei mit einem Namen, um später den Quelltext nochmal verwenden zu können. Die Dateiendung `.c` ist für C-Quelltext empfehlenswert, da der Compiler dann automatisch einige, meist richtige Annahmen trifft.

Zum Übersetzen des Quelltextes verwenden wir den Compiler GCC. Das Programm könnt ihr mal mit `gcc` ausführen. Es wird euch sagen, dass ihr keine Dateien angegeben habt. `gcc` ist also offensichtlich ein nichtinteraktives Programm. `gcc` erwartet als Argument eine beliebige Anzahl von C-Dateien, und erzeugt daraus ein Programm namens `a.out`. Also einfach mal ausprobieren, und das Programm mal ausführen. Herzlichen Glückwunsch, ihr habt soeben ein C-Programm geschrieben.

Der `gcc` kann natürlich noch viel mehr, und sein Verhalten lässt sich über weitere Parameter sehr detailliert steuern. Als Nachschlagewerk für den Compiler und die Funktionen der C-Standardbibliothek können die *manpages* verwendet werden, z.B. `man strlen`. Manchmal kommt dann zunächst die Beschreibung eines Konsolenkommandos, z.B. `man printf`. In

diesem Fall noch die Zahl 3 mit angeben (und in seltenen Ausnahmen auch die 2; müsst ihr ausprobieren), z.B. `man 3 printf`. Beenden kann man die *manpages* mit der Taste `q`. Mehr Informationen zu den *manpages* erhält man durch Eingabe von: `man man`.

# Teil III: Kommandoeingabe unter Windows

---

## 1 Arbeiten mit der Kommandozeile

Die Kommandozeile, auch als Eingabeaufforderung oder Shell (engl. für Schale bzw. Oberfläche) bekannt, stellt eine Möglichkeit des Umgangs mit dem Computer dar. Eine andere Möglichkeit hat zwei bis 20 Tasten und leuchtet hübsch an der Unterseite. Da ihr die Maus und grafische Oberflächen schon kennt, und wir euch nicht langweilen wollen, lernt ihr in diesem Übungspaket etwas Neues kennen: die Kommandozeile.

**Die Shell:** Um mit der Kommandozeile zu arbeiten, müsst ihr sie erstmal finden. Um nämlich dem normalen Nutzer das Leben zu vereinfachen, haben die Betriebssystementwickler die Shell meist gut versteckt. Eine Möglichkeit sieht wie folgt aus:

**Aufruf:** `Win+R` , dann `cmd` eintippen und `Enter` drücken

Es öffnet sich ein neues Fenster. Das sieht aber schon auf den ersten Blick ganz anders aus als alle anderen Fenster. In dem Fenster blinkt was rum, der Rest ist „Voodoo“.

Was sagt uns das? Dass da was blinkt ist gut. Was können wir machen? Wenn wir jetzt auf der Tastatur rumhacken, erscheinen die Zeichen genau dort wo es blinkt, und es blinkt rechts daneben. Tippen wir an dieser Stelle etwas Sinnvolles ein und drücken Enter, so passiert etwas Sinnvolles.

**Der Editor:** Der Editor ist ein (Hilfs-) Programm, mit dessen Hilfe wir Dateien erstellen und verändern können. Ein einfacher Editor heißt `notepad.exe`.

**Aufruf:** `notepad.exe`

Zu unserer Überraschung ist ein Editor aufgegangen. Wenn ihr das Programm beendet, könnt ihr hier weitermachen. Zuerst schauen wir mal, in welchem Ordner wir uns befinden.

**Navigation im Dateisystem:** Um sich im Dateisystem zurecht zu finden, benötigen wir eigentlich drei Kommandos.

**Wo sind wir? Kommando:** `cd`

Dieser Ordner enthält vermutlich Dateien. Auch das gucken wir uns mal an.

**Was finden wir da, wo wir sind? Kommando:** `dir`

Natürlich seht ihr jetzt keine Dateien, sondern nur die Dateinamen. Alles andere wäre ziemlich unübersichtlich. Falls die Liste länger ist als der Bildschirm, können wir ein kleines Problem haben, da die Anzeige schnell verschwindet und man nicht alles

sieht. Da dieses Problem relativ häufig ist, gibt es eine Möglichkeit der seitenweisen Anzeige. Tippt nach dem Befehl einfach einen senkrechten Strich und `more` ein:

**Kommando:** `<befehl> | more`

**Beispiel:** `dir | more`

So, jetzt haben wir schon eine Menge Dateien gesehen. Sie befinden sich alle im aktuellen Verzeichnis. Lustigerweise gibt es noch mehrere Verzeichnisse. Eine Änderung des aktuellen Verzeichnisses ist mit `cd` möglich. Sucht euch einfach mal eines aus.

`cd <Verzeichnisname>`

Vielleicht wollt ihr wieder zurück in das vorherige Verzeichnis.

`cd ..`

**Dateien im Detail:** Jetzt, da wir Verzeichnisse beherrschen, können wir uns mal die Dateien genauer anschauen. Erstmal schauen wir, ob eine bestimmte Datei namens `datei.test` im aktuellen Ordner vorhanden ist.

**Kommando:** `dir datei.test`

Das ist sie nicht. Na dann erstellen wir sie doch einfach.

**Kommando:** `type nul >> datei.test`

Jetzt gibt es eine Datei namens `datei.test`. Glaubt ihr nicht? Schaut doch nach!

**Kommando:** `dir datei.test`

Toll! Und was ist drin?

**Kommando:** `type datei.test`

Natürlich nichts. Aber wie bekommen wir diese Datei jetzt wieder weg?

**Kommando:** `del datei.test`

Und jetzt ist sie weg? Einfach mal nachschauen!

**Kommando:** `dir datei.test`

Diese Datei ist jetzt wirklich weg. So richtig. Nix Papierkorb oder so. Weg!

**Verzeichnisse:** Wir können jetzt Dateien erstellen. Geht das vielleicht auch mit Ordnern? Erstmal schauen, ob es so einen Ordner schon gibt.

**Kommando:** `dir ordner`

Gibt's nicht? Na dann erstellen wir ihn doch.

**Kommando:** `mkdir ordner`

So, wie können wir das jetzt überprüfen? Ganz einfach:

**Kommando:** `dir ordner`

Jetzt gibt es also einen leeren Ordner **ordner**. Um jetzt verschiedene Ordner gleichen Namens auseinander zu halten, lassen wir uns mal den gesamten Pfad anzeigen

**Kommando:** `cd`

Weil wir jetzt einen Ordner erstellt haben, können wir in das neue Verzeichnis wechseln.

**Kommando:** `cd ordner`

Um zu gucken wo wir uns befinden:

**Kommando:** `cd`

Jetzt können wir mal nachschauen, was in diesem Ordner drin ist.

**Kommando:** `dir`

Natürlich nix. Wir haben den Ordner ja auch gerade erst erstellt, und niemanden sonst an den Rechner rangelassen. Kommen wir eigentlich wieder aus dem Ordner raus?

**Kommando:** `cd ..`

Mal schauen, ob's geklappt hat:

**Kommando:** `cd`

Unser Ordner ist aber noch da.

**Kommando:** `dir ordner`

Aber gleich nicht mehr.

**Kommando:** `rmdir ordner`

Jetzt ist er wirklich weg?

**Kommando:** `dir ordner`

Nix zu sehen. Können wir denn noch in das Verzeichnis wechseln?

**Kommando:** `cd ordner`

Wir können also in der Kommandozeile Ordner erstellen, in ihnen herumnavigieren, und Dateien erstellen und löschen. Das ist zwar schön, aber für sich genommen nicht besonders sinnvoll. Also schauen wir uns mal an, wie man Programme startet.



## 2 Sinnvolle Arbeiten mit der Kommandozeile

**Web-Browser:** Wenn wir eine Internet-Seite anschauen wollen, brauchen wir einen Browser. Starten kann man diesen, indem man einfach den Programmnamen in der Kommandozeile eintippt.

```
"C:\Programme\Internet Explorer\iexplore.exe"
```

Den Browser kennt ihr, hier gibt es also nichts mehr zu erklären.

Da ihr im Browser häufig eine Internetseite anschauen wollt, könnt ihr sie auch mit angeben:

```
"C:\Programme\Internet Explorer\iexplore.exe" "http://xkcd.com/"
```

Alles was ihr nach dem Programmnamen schreibt, sind Programmparameter. Mit diesen Parametern könnt ihr offensichtlich das Verhalten des Programms beeinflussen. Meist werden Parameter durch Leerzeichen getrennt.

**Variablen:** Wirklich schön ist der Aufruf aber erst, wenn man nicht den kompletten Programmpfad eintippen muss. Dafür gibt es verschiedene Mechanismen, unter Linux eine Umgebungsvariable PATH, unter Windows ebenfalls PATH und zusätzlich die Registry. Das Ergebnis ist das gleiche.

```
start iexplore.exe
```

Um ein eigenes Programm, beispielsweise `programm.exe`, im aktuellen Verzeichnis zu starten, reicht folgendes:

```
programm.exe
```

Falls ihr mal ein Programm abbrechen wollt, das nichts sinnvolles mehr macht, funktioniert das mit der Tastenkombination **Strg+C**.

**Abbruch von Programmen:** Strg+C .

Das könnt ihr schnell mal ausprobieren. Startet ein Programm, z.B. `date.exe`. Sobald ihr **Strg+C** drückt, wird es beendet.

**Mehr zu Dateien:** Da man zum Programmieren irgendwie Quelltext schreiben muss, gibt es Editoren, mit denen man das relativ bequem machen kann.

**Kommando:** `notepad.exe`

Jetzt könnt ihr fleißig Tasten drücken. Das Ergebnis lässt sich als Datei abspeichern. Gebt ihr einen schönen Namen, z.B. `text.txt`.

An der Kommandozeile könnt ihr euch die Datei mittels des Befehls `type` anschauen. Interessanterweise kann das Programm auch mehrere Dateien hintereinander ausgeben:

**Kommando:** `type datei1 datei2 datei3`

Ganz hübsch soweit. Der Inhalt der Dateien wird an der Kommandozeile angezeigt. Falls man diesen Inhalt in eine Datei schreiben möchte, kann man Folgendes eingeben.

**Kommando:** `type datei1 datei2 datei3 > ziel`

So lassen sich einfach mehrere Dateien zusammenfügen. Der Mechanismus dahinter heißt Ausgabeumleitung. Das Zeichen > ist dafür da.

**Ausschalten des Rechners:** Wenn es mal wieder so weit ist, den Rechner neu zu starten, dann genügt:

**Kommando:** `shutdown -r -t 0`

**Multimedia:** Eine DVD kann man beispielsweise wie folgt anschauen:

**Kommando:** `start wmpayer /device:DVD /fullscreen`

### 3 Kommandozeile für den C-Programmierer

Da `notepad.exe` ein ziemlich gruseliger Editor ist, stellen wir für die Übung einen Editor mit Syntax-Highlighting zur Verfügung. Er befindet sich im Übungs-Framework im Ordner `Notepad++`. Da Windows die Pfadeinträge für den Compiler nicht enthält, findet ihr im Ordner `MinGW` das Programm `CommandPromptPortable.exe`, welches euch das Eintragen abnimmt.

Jetzt könnt ihr mal den Quelltext aus der Vorlesung eintippen. Wenn ihr damit fertig seid, müsst ihr den Quelltext abspeichern. Genau genommen erstellt ihr eine Datei mit einem Namen, um später den Quelltext nochmal verwenden zu können. Die Dateiendung `.c` ist für C-Quelltext empfehlenswert, da der Compiler dann automatisch einige, meist richtige Annahmen trifft.

Zum Übersetzen des Quelltextes verwenden wir den Compiler GCC. Das Programm könnt ihr mal mit `gcc` ausführen. Es wird euch sagen, dass ihr keine Dateien angegeben habt. `gcc` ist also offensichtlich ein nichtinteraktives Programm. `gcc` erwartet als Argument eine beliebige Anzahl von C-Dateien, und erzeugt daraus ein Programm namens `a.exe`. Also einfach mal ausprobieren, und das Programm mal ausführen. Herzlichen Glückwunsch, ihr habt soeben ein C-Programm geschrieben.

Der `gcc` kann natürlich noch viel mehr, und sein Verhalten lässt sich über weitere Parameter sehr detailliert steuern. Als Nachschlagewerk für den Compiler und die Funktionen der C-Standardbibliothek können die *manpages* verwendet werden. Da es für Windows keine brauchbare Implementierung des Programms `man` gibt, bleibt euch die Online-Recherche:

[http://www.kernel.org/doc/man-pages/online\\_pages.html](http://www.kernel.org/doc/man-pages/online_pages.html).

Hier ist Section 3 für euch die interessanteste.

## Teil IV: Die wichtigsten Befehle

---

### Die wichtigsten Linux Kommandos:

Aufgabe	Kommando
Anzeige des aktuellen Verzeichnisses	pwd
Ordnerinhalt anzeigen	ls
Datei erstellen	touch <dateiname>
Dateien anschauen	cat <dateiname1> <dateiname2> ...
Verzeichnis erstellen	mkdir ordner
Verzeichnis löschen	rmdir ordner
Wechsel in ein anderes Verzeichnis	cd <Ordnername>
Wechsel in das Oberverzeichnis	cd ..
Dateien löschen	rm <Dateiname1> <Dateiname2>
Umbenennen und verschieben von Dateien und Verzeichnissen	mv <alteredatei> <neuedatei>
Programme ausführen	<Dateiname> <Parameter>
Programm im aktuellen Verzeichnis ausführen	./<programm>
Ausgabe seitenweise umbrechen	<befehl>   more

### Die wichtigsten Windows Kommandos:

Aufgabe	Kommando
Anzeige des aktuellen Verzeichnisses	cd
Ordnerinhalt anzeigen	dir
Datei erstellen	type nul >> <dateiname>
Dateien anschauen	type <dateiname1> <dateiname2> ...
Verzeichnis erstellen	mkdir ordner
Verzeichnis löschen	rmdir ordner
Wechsel in ein anderes Verzeichnis	cd <Ordnername>
Wechsel in das Oberverzeichnis	cd ..
Dateien löschen	del <Dateiname>
Umbenennen und verschieben von Dateien und Verzeichnissen	rename <alteredatei> <neuedatei>
Programme ausführen	<Dateiname> <Parameter>
Programm im aktuellen Verzeichnis ausführen	<programm.exe>
Ausgabe seitenweise umbrechen	<befehl>   more

# Übungspaket 3

## Mein erstes Programm: Fläche eines Rechtecks

---

### Übungsziele:

1. Der Software Life Cycle im Überblick
2. Umgang mit Editor und Compiler
3. Editieren und Starten eines eigenen Programms

### Skript:

Kapitel: 6 bis 8 und 16

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Dieses Übungspaket soll euch ein wenig mit dem Computer und den für die Lehrveranstaltung notwendigen Werkzeugen vertraut machen. Die Aufgabe besteht darin, das in der Vorlesung (siehe auch Kapitel 7 und 8) entwickelte Programm zur Berechnung des Flächeninhalts eines Rechtecks einzugeben, zu übersetzen und zur Ausführung zu bringen. Im Zuge dieser Tätigkeiten sollt ihr euch auch nochmals etwas mit dem Entwicklungsprozess vertraut machen, der auch *Software Life Cycle* genannt wird.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Der Software Life Cycle im Überblick

In der Vorlesung haben wir erklärt, dass das Programmieren nicht ein monolytischer Prozess ist, sondern einer, der sich in einzelne Phasen unterteilen lässt. Jede Phase hat ihren eigenen Namen, erstellt unterschiedliche Dokumente und verwendet spezifische Werkzeuge. Erläutere kurz in eigenen Worten, was sich hinter dem Begriff „Software Life Cycle“ verbirgt und was das Ziel dieses Gesamtprozesses ist (Hinweis: die Beschreibung der einzelnen Phasen kommt in der nächsten Aufgabe):

Der Begriff Software Life Cycle beschreibt den Weg eines Programms von der Idee bis zu seiner Realisierung auf dem Rechner und die später notwendigen Wartungsarbeiten. Die typischen Phasen sind Aufgabenstellung, Problemanalyse, Entwurf, Implementierung, Codierung und Test. Im Regelfall merkt man in einer der späteren Phasen oder den zwischenzeitlich durchgeführten Tests, dass in früheren Phasen Dinge vergessen oder ungünstig entschieden wurden, sodass man teilweise auch wieder etwas zurück gehen muss. Der Prozess des Software Life Cycle enthält also Schleifen und ist selten irgendwann zu Ende.

## Aufgabe 2: Die Phasen des Software Life Cycles

Nenne jeweils die Namen der Phasen, der Dokumente und der Werkzeuge, sowie die wesentlichen Ziele der einzelnen Phasen:

### 1. Phase

Name:	Aufgabenstellung
Dokumente:	Aufgabenblatt
Werkzeuge:	Zettel und Stift
Ziele:	Formulierung der Aufgabe des Programms

### 2. Phase

Name:	Problemanalyse
Dokumente:	Pflichtenheft
Werkzeuge:	Zettel und Stift
Ziele:	Klarheit über die Aufgabenstellung. Was soll gemacht werden? Verbindliches Pflichtenheft mit Ziel-/Aufgabenstellung.

### 3. Phase

Name:	Entwurf
Dokumente:	Modulzeichnung
Werkzeuge:	Zettel und Stift
Ziele:	Aufteilung des Ganzen in mehrere möglichst eigenständige Komponenten. Erarbeiten einer Modulstruktur. Diese Phase entfällt in der ersten Semesterhälfte, da die Aufgaben ohnehin sehr klein und einfach sind.

#### 4. Phase

Name:	Implementierung
Dokumente:	Struktogramme / Schrittweise Verfeinerung
Werkzeuge:	Zettel und Stift
Ziele:	Algorithmische Beschreibungen der einzelnen Funktionalitäten. Diese sollen so sein, dass sie möglichst direkt in eine Programmiersprache überführt werden können.

#### 5. Phase

Name:	Codierung
Dokumente:	Quelltexte
Werkzeuge:	Texteditor und Compiler
Ziele:	Ein lauffähiges Programm

#### 6. Phase

Name:	Test
Dokumente:	Programm
Werkzeuge:	Testdaten
Ziele:	Durchlaufen aller Programmzweige zur Sicherstellung der korrekten Funktion des Programms gemäß der Aufgabenstellung.

## Teil II: Quiz

---

### Aufgabe 1: Schrittweise Verfeinerung

In Vorlesung und Skript haben wir die Methode der Schrittweisen Verfeinerung kennengelernt. Hierzu haben wir folgende Fragen, die ihr anhand eines Beispiels erklären sollt:

Wie werden Anweisungen formuliert?	Drucke Variable a
Wie werden Variablen definiert?	Variablen: Typ Integer: a, b, F
Wie werden Berechnungen ausgeführt?	Setze $F = a * b$
Wie werden Werte eingelesen?	Einlesen von Seite a
Wie werden Ergebnisse ausgegeben?	Ausgabe des Flächeninhalts F

### Aufgabe 2: Programmieren auf dem PC

Um auf einem Rechner (wie dem PC) programmieren zu können, benötigen wir einige Programme. Ergänze die folgende Tabelle unter der Annahme, dass sich der Quelltext unseres ersten Programms in der Datei `rechteck.c` befindet:

Funktion	Name	Programmaufruf
Editor	<code>notepad.exe</code>	<code>notepad.exe rechteck.c</code>
Compiler	<code>gcc.exe</code>	<code>gcc.exe -Wall -o rechteck.exe rechteck.c</code>
Beispielprogramm	<code>rechteck.exe</code>	<code>rechteck.exe</code>

# Teil III: Fehlersuche

---

## Aufgabe 1: Syntaxfehler

Ein wesentliches Ziel dieses Übungspaket ist es, das Beispielprogramm zur Berechnung der Fläche eines Rechtecks in den PC einzugeben und erfolgreich zur Ausführung zu bringen. Es ist ganz natürlich, dass bei diesem Arbeitsprozess verschiedene (Kopier-) Fehler auftreten. So erging es auch DR. PROFILING, wie ihr gleich sehen werdet.

```
1  include <stdio.h>
2
3  int MAIN( int argc, char ** argv )
4      {#
5          int a b, F;
6          printf( "Bitte Seite a eingeben: " )
7          input( "%d", & a );
8          output( "Bitte Seite b eingeben: " );
9          scanf( "%d", b );
10         F = A * b;
11         print( "Der Flaecheninhalt betraegt F=%d m*m\n", F );
12     ]
```

Mit Ausnahme von Zeile 2 enthält jede Programmzeile genau einen Fehler. Finde dies, erkläre ihre Ursache und korrigiere sie (im Zweifelsfalle hilft Skriptkapitel 7 weiter).

Zeile	Fehler	Erläuterung	Korrektur
1	include	Präprozessoranweisungen beginnen mit #	#include
3	MAIN	main() muss klein geschrieben werden	main
4	# zu viel	Das Zeichen # ist hier nicht erlaubt	# entfernen
5	a b	Variablen müssen mit Komma getrennt werden	a, b
6	; fehlt	Anweisungen müssen mit einem Semikolon enden	printf(...);
7	input	Die Eingabefunktion heißt scanf()	scanf(...);
8	output	Ausgabefunktion heißt printf()	printf(...);
9	b	scanf() erwartet die Adresse einer Variablen	& b
10	A	C unterscheidet Gross- und Kleinbuchstaben	a
11	print	Ausgabefunktion heißt printf()	printf(...);
12	]	Anweisungsblöcke werden mit {} zusammengefasst	}



### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int main( int argc, char ** argv )
4  {
5      int a, b, F;
6      printf( "Bitte Seite a eingeben: " );
7      scanf( "%d", & a );
8      printf( "Bitte Seite b eingeben: " );
9      scanf( "%d", & b );
10     F = a * b;
11     printf( "Der Flaecheninhalt betraegt F=%d m*m\n", F );
12 }
```

## Teil IV: Anwendungen

---

In den Kapiteln 7 und 8 haben wir ein einfaches Programm zur Berechnung des Flächeninhalts schrittweise entwickelt und in die Programmiersprache C umgesetzt. In diesem Anwendungsteil geht es nun darum, dieses erste C-Programm in den Rechner einzutippen und zum „laufen“ zu bekommen. Die folgenden Aufgaben beschreiben die einzelnen durchzuführenden Arbeitsschritte. Auf den folgenden Seiten zeigen wir beispielhaft ein typisches Sitzungsprotokoll mit häufig auftretenden Fehlern. Das Durchlesen dieses Protokolls hilft in vielen Fällen, den eigenen Fehlern auf die Spur zu kommen.

### Aufgabe 1: Eingabe des Programms

Die vier wesentlichen Arbeitsschritte sind:

1. Einrichten eines Unterverzeichnisses (`mkdir/md`), um den Überblick zu behalten.
2. Starten des Editors (mit einem vernünftigen Dateinamen), z.B. `gedit rec.c`.
3. Programmieren, d.h. in diesem Fall das Abtippen des vorbereiteten Programms.
4. Abspeichern.

### Aufgabe 2: Übersetzen und Eliminieren von Fehlern

Zunächst darf euer Programm keine Syntaxfehler enthalten, damit es vom Compiler übersetzt wird. Der normale Zyklus (auch bei Profis) ist: Abspeichern, Übersetzen, angezeigte Syntaxfehler im Editor korrigieren, Abspeichern, Übersetzen, angezeigte Syntaxfehler ...

In der Regel meldet der Compiler am Anfang sehr viele Fehler. Die meisten davon sind Folgefehler, die aus einem zuvor gemachten Fehler resultieren. Don't panic! Den ersten Fehler korrigieren und erneut schauen, was passiert.

### Aufgabe 3: Programm testen

Programm starten, Testdatensatz eingeben und das Ergebnis mit den erwarteten Ausgaben vergleichen. Gegebenenfalls den Editor starten, korrigieren, ...

**Wichtiger Hinweis:** Nach durchgeführten Programmänderungen sollte man das Testen unbedingt wieder von vorne anfangen, denn oft führen Korrekturen zu neuen Fehlern; das Programm wird häufig erst einmal „verschlimmbessert“.

Wenn alles fehlerfrei läuft, einen Kaffee mit den Kumpels trinken und tierisch freuen!

## Beispielhaftes Sitzungsprotokoll

Im folgenden Sitzungsprotokoll gehen wir davon aus, dass der Nutzer `hirsch` an seinem Rechner `super-PC` sitzt und vor jeder Eingabe den Prompt `hirsch@super-PC>` sieht. Nehmen wir nun an, dass `hirsch` das Programm aus Kapitel 7 abgetippt und unter dem Namen `rect.c` abgespeichert hat. Dabei hat er sich aber an einigen Stellen vertippt:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int ä = 0;          // Seite 1
6      int b = 0;          // Seite 2
7      int F = 0;          // Fläche
8
9      // einlesen von Seite 1
10     printf("Bitte Seite a eingeben: "); scanf("%d", &ä);
11
12     // einlesen von Seite 2
13     printf("Bitte Seite b eingeben: "); scanf("%d", b);
14
15     // Ergebnis berechnen und ausgeben
16     F = ä x b;
17     printf("Der Flächeninhalt beträgt F= %d m*m\n", F);
18
19     // Rückgabewert auf 0 setzen
20     return 0;
21 }
```

`hirsch` übersetzt mittels `gcc -o rect rect.c` und sieht folgende Fehlermeldungen:

```
hirsch@super-PC> gcc -o rect rect.c
rect.c: In function 'main':
rect.c:5: error: stray '\303' in program
rect.c:5: error: stray '\244' in program
rect.c:5: error: expected identifier or '(' before '=' token
rect.c:10: error: stray '\303' in program
rect.c:10: error: stray '\244' in program
rect.c:10: error: expected expression before ')' token
rect.c:16: error: stray '\303' in program
rect.c:16: error: stray '\244' in program
rect.c:16: error: 'x' undeclared (first use in this function)
rect.c:16: error: (Each undeclared identifier is reported only once
rect.c:16: error: for each function it appears in.)
rect.c:16: error: expected ';' before 'b'
```

Auf den ersten Blick sind derartig viele Fehler überwältigend und frustrierend. Aber das ist gar nicht so schlimm, denn viele Fehler sind eigentlich gar keine richtigen sondern nur einfache Folgefehler. Der Compiler hilft einem jedenfalls enorm, wenn man nur seine Ausgaben liest. Zur Wiederholung: *Man muss die Ausgaben (Fehlermeldungen) des Compilers lesen, dann bekommt man gute Hinweise darauf, was falsch ist! Also nicht ignorieren!*

Fangen wir nun einfach gemeinsam vorne an. Die erste wirkliche Ausgabe besagt, dass der Compiler in Zeile 5 auf ein unbekanntes Zeichen gestoßen ist, mit dem er nichts anfangen kann. Sofern das verwendete System die UTF-8-Kodierung verwendet, ist die Kombination `\303\244` ein `ä`. Umlaute sind aber keine Zeichen aus dem englischen Zeichensatz und damit in C nicht erlaubt. Also dürfen wir sie nicht verwenden. Daher tauschen wir innerhalb des gesamten Programms das `ä` gegen ein `ae` aus. Das gleiche machen wir natürlich mit den anderen Umlauten.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 0;          // Seite 1
6      int b = 0;          // Seite 2
7      int F = 0;          // Flaeche
8
9      // einlesen von Seite 1
10     printf("Bitte Seite a eingeben: "); scanf("%d", &a);
11
12     // einlesen von Seite 2
13     printf("Bitte Seite b eingeben: "); scanf("%d", &b);
14
15     // Ergebnis berechnen und ausgeben
16     F = a x b;
17     printf("Der Flaecheninhalt betraegt F= %d m*m\n", F);
18
19     // Rueckgabewert auf 0 setzen
20     return 0;
21 }
```

Und schon tritt nur noch ein einziger Fehler auf:

```
hirsch@super-PC> gcc -o rect rect.c
rectc: In function 'main':
rectc:16: error: expected ';' before 'x'
```

Auf den ersten Blick scheint ihm nun das `x` nicht zu gefallen. Im verwendeten Fall ist das `x` der Name einer Variablen. Aber wir wollen ja eigentlich multiplizieren, wofür wir den Operator `*` verwenden *müssen*. Dies haben wir im folgenden Programm entsprechend berücksichtigt:

```

1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 0;          // Seite 1
6      int b = 0;          // Seite 2
7      int F = 0;          // Flaeche
8
9      // einlesen von Seite 1
10     printf("Bitte Seite a eingeben: "); scanf("%d", & a);
11
12     // einlesen von Seite 2
13     printf("Bitte Seite b eingeben: "); scanf("%d", b);
14
15     // Ergebnis berechnen und ausgeben
16     F = a * b;
17     printf("Der Flaecheninhalt betraegt F= %d m*m\n", F);
18
19     // Rueckgabewert auf 0 setzen
20     return 0;
21 }

```

Nun tritt plötzlich ein neuer Fehler auf:

```

hirsch@super-PC> gcc -o rect rect.c
rect.c:(.text+0x29): undefined reference to 'printf'
collect2: ld returned 1 exit status

```

Diese Fehlermeldung besagt, dass der Compiler etwas verwenden soll, das `printf` heißt, er es aber nirgends finden kann. Wir erinnern uns: die Ausgabefunktion heißt `printf()`:

```

1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 0;          // Seite 1
6      int b = 0;          // Seite 2
7      int F = 0;          // Flaeche
8
9      // einlesen von Seite 1
10     printf("Bitte Seite a eingeben: "); scanf("%d", & a);
11
12     // einlesen von Seite 2
13     printf("Bitte Seite b eingeben: "); scanf("%d", b);
14
15     // Ergebnis berechnen und ausgeben

```

```

16         F = a * b;
17         printf("Der Flaecheninhalt betraegt F= %d m*m\n", F);
18
19         // Rueckgabewert auf 0 setzen
20         return 0;
21     }

```

Der Compiler kann jetzt unser Programm ohne Fehlermeldungen übersetzen, da syntaktisch alles richtig ist. Dies ist aber erst die halbe Miete, denn wir müssen jetzt *alleine* überprüfen, ob das Programm auch das macht, was es machen soll. Schauen wir, was passiert:

```

hirsch@super-PC> gcc -o rect rect.c hirsch@super-PC> ./rect
Bitte Seite a eingeben: 2
Bitte Seite b eingeben: 3
Segmentation fault

```

Das Programm stürzt leider ab :-(, es muss also irgendwo noch mindestens ein Fehler sein. Auch hier hilft es enorm, sich die Ausgaben genau anzuschauen und einen Blick in das Programm zu werfen. Was sehen wir? Zunächst sehen wir die Ausgabe **Bitte Seite a eingeben:** und wir gaben eine 2 ein. Damit sind wir schon mal mindestens bis zur Zeile 10 gekommen. Anschließend sehen wir noch die Ausgabe **Bitte Seite b eingeben:** und konnten eine 3 eingeben. Also hat mindestens auch das zweite `printf()` geklappt. Der Fehler kann nur beim Einlesen der Zahl 3 *oder* später (bei der Berechnung) passiert sein.

Also schauen wir in das Programm. Dort sehen wir in Zeile 13 nach dem `printf()` die Anweisung `scanf("%d", b);` Ein Vergleich mit Programmzeile 10 verrät uns, dass dort ein `&`-Zeichen fehlt; wir hätten vielleicht doch nicht die Warnung

```

rect.c: In function 'main':
rect.c:13: warning: format '%d' expects type 'int *', but argument 2 has
type 'int'

```

ignorieren sollen, die der Compiler auf manchen Systemen ausgibt. Die Bedeutung und den Hintergrund dieser Meldung werden wir erst viel später behandeln und verstehen. In Zeile 13 muss also `scanf("%d", &b);` stehen. Wir merken uns aber: Warnungen des Compilers *keinesfalls ignorieren*, sondern die entsprechenden Programmzeilen genau anschauen und ggf. die Betreuer um Rat fragen! Nach dieser letzten Korrektur sehen wir Folgendes:

```

hirsch@super-PC> gcc -o rect rect.c
hirsch@super-PC> ./rect
Bitte Seite a eingeben: 2
Bitte Seite b eingeben: 3
Der Flaecheninhalt betraegt F= 6 m*m

```

*Yes, that's it!*

Richtig schön wird das Programm, wenn man unsinnige Eingaben als solche erkennt, und den Programmablauf entsprechend sinnvoll gestaltet. Hier wird beispielsweise bei negativen Seitenlängen das Programm ohne weitere Berechnungen beendet.

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int a = 0;          // Seite 1
6      int b = 0;          // Seite 2
7      int F = 0;          // Flaeche
8
9      // einlesen von Seite 1
10     printf("Bitte Seite a eingeben: "); scanf("%d", &a);
11
12     if ( a > 0 )
13     {
14         // einlesen von Seite 2
15         printf("Bitte Seite b eingeben: "); scanf("%d", &b);
16
17         if ( b > 0 )
18         {
19             // Ergebnis berechnen und ausgeben
20             F = a * b;
21             printf("Der Flaecheninhalt betraegt %d m*m\n",F);
22         }
23         else printf("Fehler: Seite b muss positiv sein\n" );
24     }
25     else printf("Fehler: Seite a muss positiv sein\n" );
26
27     // Rueckgabewert auf 0 setzen
28     return 0;
29 }
```

# Übungspaket 4

## Klassifikation von Dreiecken

---

### Übungsziele:

1. Selbstständiges Entwickeln eines ersten Programms
2. Anwenden der Methoden des Software Life Cycles
3. Programmentwurf durch Anwendung der Methode der Schrittweisen Verfeinerung
4. Erstellen eines ersten, eigenen C-Programms
5. Finden von Programmierfehlern

### Skript:

Kapitel: 6 bis 9 und 16

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Nach dem ihr im 3. Übungspaket ein in der Vorlesung entwickeltes Programm auf dem Rechner zum Laufen gebracht habt, müsst ihr nun euer erstes eigenes Programm selber entwickeln. Im Rahmen der Programm-entwicklung sollt ihr alle Phasen des Software Life Cycles anwenden.



# Teil I: Stoffwiederholung

---

## Aufgabe 1: Das Konzept der Fallunterscheidung

In Kapitel 9 des Skriptes haben wir die *Fallunterscheidung* eingeführt. Erkläre mit eigenen Worten, wofür sie verwendet werden kann und wie sie mittels der Methode der Schrittweisen Verfeinerung dargestellt wird!

Die Fallunterscheidung dient dazu, einen logischen Ausdruck auszuwerten und in Abhängigkeit des Ergebnisses etwas Bestimmtes zu tun. Man spricht auch von einer bedingten Programmverzweigung. Die einfachste Form der Fallunterscheidung kann man als **wenn-dann** Konstruktion beschreiben:

```
wenn ein logischer Ausdruck wahr ist  
dann tue etwas Sinnvolles
```

Diese Konstruktion lässt sich erweitern:

```
wenn ein logischer Ausdruck wahr ist  
dann tue etwas Sinnvolles  
sonst tue etwas anderes
```

## Aufgabe 2: Die Fallunterscheidung in C

Erkläre kurz, wie die Fallunterscheidung in der Programmiersprache C implementiert wird! Gib zusätzlich mindestens vier Beispiele für eine „sinnvolle“ Fallunterscheidung in C an!

Die einfache **wenn-dann** Konstruktion wird in C wie folgt umgesetzt:

```
if ( logischer Ausdruck ) Anweisung;
```

Das Schlüsselwort **if** wird mit der Bedeutung **wenn** verwendet. C kennt kein Schlüsselwort für **dann**; es ist implizit enthalten. Es kann *nur* eine Anweisung folgen; benötigt man mehr als eine Anweisung, müssen diese mit geschweiften Klammern eingeschlossen werden (aus Platzgründen ohne Einrückungen):

```
1 if ( logischer Ausdruck )  
2 { Anweisung_1; Anweisung_2; }
```

Die erweiterte Fallunterscheidung sieht wie folgt aus (wieder ohne Einrückungen):

```
1 if ( logischer Ausdruck )  
2 { Anweisung_1; Anweisung_2; }  
3 else { Anweisung_3; Anweisung_4; }
```

Die Anweisungen 1 und 2 werden ausgeführt, wenn die Bedingung erfüllt ist (der logische Ausdruck wahr ist), sonst die Anweisungen 3 und 4.

Weitere Beispiele (Anmerkung: i und x sind vom Typ int):

```
1  if ( i == 0 )
2      printf( "i ist null\n" );
3
4  if( i == 0 )
5      printf( "i ist null\n" );
6  else printf( "i ist nicht null\n" );
7
8  if( i == 0 && x < 3 )
9  {
10     printf( "i ist null\n" );
11     printf( "x ist kleiner als 3\n" );
12 }
13 else {
14     printf( "entweder ist i ungleich null" );
15     printf( "oder x ist groesser oder gleich 3\n" );
16 }
17
18 if( i == 0 && x < 3 )
19 {
20     printf( "i ist null\n" );
21     if(x>0)
22     {
23         printf( "x ist entweder 1 oder 2\n" );
24     }
25     else {
26         printf( "x ist negativ oder gleich null\n" );
27     }
28 }
29 else {
30     printf( "entweder ist i ungleich null" );
31     printf( "oder x ist groesser oder gleich 3\n" );
32 }
```

## Aufgabe 3: Detailfragen zur Speicherorganisation

Wie kann man die Speicheradressen von Variablen und Funktionen ausgeben?

Variablen: `printf( "%p", & name_der_variablen );`

Funktionen: `printf( "%p", & name_der_funktion );`

## Teil II: Quiz

---

### Aufgabe 1: Phasen des Software Life Cycle

Wie heißen die einzelnen Phasen des Software Life Cycle und worauf zielen sie ab?

1. Phase

Name:	Aufgabenstellung
Ziele:	Erfassung des Problems, des Problemumfangs und der notwendigen Randbedingungen, vage Beschreibung des gewünschten Ergebnisses.

2. Phase

Name:	Problemanalyse
Ziele:	Konkretisierung aus Programmiersicht: welche Daten, wie wird verarbeitet, welches Ergebnis, Bedienung, Reaktion auf Fehler, welche Testdaten?

3. Phase

Name:	Entwurf
Ziele:	Aufteilung komplexer Aufgabenstellungen in mehrere kleinere Komponenten. Wie werden Daten zwischen den Komponenten vermittelt?

4. Phase

Name:	Implementierung
Ziele:	Detaillierte Beschreibung der Algorithmen der einzelnen Module: Struktogramm/schrittweise Verfeinerung (kein C-Code).

5. Phase

Name:	Kodierung
Ziele:	Umsetzen der Implementierung in C-Code.

6. Phase

Name:	Test
Ziele:	Test aller Programmteile mit gültigen und ungültigen Daten, Fehlererkennung und -behebung.

# Teil III: Fehlersuche

---

## Aufgabe 1: Syntaxfehler

Unser Programmiererteam, die TRI ANGELS haben in folgendes Programm je Zeile einen Fehler eingebaut. Finde, erkläre und korrigiere sie.

```
1 #include <stdio.h>
2 INT main( int argc, char **argv )
3     [
4         int i, j k, l;
5         i = 1; j = 2; k = 3; l = 2:
6         if ( i == 1
7             printf( "i ist 1\n" );
8         if ( i == 1 &&& j == 3 )
9             printf( "i ist 1 und j ist 3\n" );
10        if ( k == 1 && [j == 3 || l < 5))
11            print( "k ist 1 und j ist 3 oder l kleiner 5\n" );
12        if ( i => 2 * (j + k)*l || l == 2 )
13            printf "der komplizierte ausdruck ist wahr\n";
14    $}
```

Zeile	Fehler	Erläuterung	Korrektur
1	großes D	C unterscheidet Groß/Kleinbuchstaben	#include
2	großes INT	C unterscheidet Groß/Kleinbuchstaben	int
3	[	In C werden Blöcke durch { und } gebildet	{
4	, fehlt	Variablen müssen mit Komma getrennt werden	j, k
5	:	Abschluss einer Anweisung mittels ;	l = 2;
6	) fehlt	Die Bedingung muss mit einer runden Klammer enden	if (i == 1)
7	" fehlt	Zeichenketten müssen innerhalb von "..." stehen	"i ... 1\n"
8	&&&	Das logische und besteht aus nur zwei &-Zeichen	&&
9	\n	Der Zeilenumbruch heißt \n. Bei \" will der Compiler ein \" ausgeben. Folgefehler: er erkennt das Ende der Zeichenkette nicht	j ist 3\n"
10	[	Die einzelnen Ausdrücke müssen in () stehen	(j == 3...)
11	f fehlt	Die Ausgabeanweisung heißt printf()	printf

Zeile	Fehler	Erläuterung	Korrektur
12	=>	Der Vergleich $\geq$ heißt in C >=	i >= 2 ...
13	() fehlen	Die Argumente von printf müssen in () stehen	printf()
14	\$ zu viel	Dort gehört kein \$-Zeichen hin	}

### Programm mit Korrekturen:

```

1  #include <stdio.h>
2  int main( int argc, char **argv )
3  {
4      int i, j, k, l;
5      i = 1; j = 2; k = 3; l = 2;
6      if ( i == 1 )
7          printf( "i ist 1\n" );
8      if ( i == 1 && j == 3 )
9          printf( "i ist 1 und j ist 3\n" );
10     if ( k == 1 && (j == 3 || l < 5))
11         printf( "k ist 1 und j ist 3 oder l kleiner 5\n" );
12     if ( i >= 2 * (j + k)*l || l == 2 )
13         printf( "der komplizierte ausdruck ist wahr\n" );
14 }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Vorüberlegung: Klassen von Dreiecken

In der Mathematik werden Dreiecke, die durch die drei Seiten  $a$ ,  $b$  und  $c$  beschrieben werden, in die folgenden fünf Klassen eingeteilt. Beschreibe jede Klasse mittels einer Formel ( $a$ ,  $b$  und  $c$ ) sowie mindestens einem Zahlenbeispiel.

1. Kein Dreieck: Die Dreiecksungleichung besagt, dass die Summe zweier Seiten immer echt größer sein muss als die dritte Seite:

Die drei Seiten  $a$ ,  $b$ ,  $c$  können kein Dreieck bilden, wenn folgende Aussage wahr ist:  $a + b \leq c$  oder  $b + c \leq a$  oder  $a + c \leq b$ . Beispiel:  $a = 5$ ,  $b = 1$ ,  $c = 1$

Umgekehrt bilden die drei Seiten  $a$ ,  $b$ ,  $c$  ein Dreieck, wenn folgende Aussage wahr ist:  $a + b > c$  und  $a + c > b$  und  $b + c > a$

2. Gleichseitiges Dreieck:

Alle drei Seiten  $a$ ,  $b$ ,  $c$  müssen gleich lang sein:  $a = b$  und  $b = c$ .

Beispiel:  $a = 4$ ,  $b = 4$ ,  $c = 4$

3. Gleichschenkliges Dreieck:

Zwei Seiten des Dreiecks müssen gleich lang sein, folgende Aussage muss also wahr sein:  $a = b$  oder  $a = c$  oder  $b = c$

Beispiel:  $a = 6$ ,  $b = 6$ ,  $c = 3$

4. Rechtwinkliges Dreieck:

Im rechtwinkligen Dreieck muss die Summe der Quadrate der beiden Seiten, die den rechten Winkel einschließen, genauso groß sein, wie das Quadrat der dritten Seite:  $a^2 + b^2 = c^2$  oder  $b^2 + c^2 = a^2$  oder  $a^2 + c^2 = b^2$

Beispiel:  $a = 3$ ,  $b = 4$ ,  $c = 5$

5. Gewöhnliches Dreieck:

Jede Kombination der Seiten  $a$ ,  $b$ ,  $c$ , die gemäß Bedingung 1 ein Dreieck bilden, aber keine der Bedingungen 2 bis 4 erfüllen, bilden ein gewöhnliches Dreieck.

Beispiel:  $a = 3$ ,  $b = 4$ ,  $c = 6$

## Aufgabe 2: Entwicklung eines Dreiecksprogramms

In dieser Aufgabe geht es um die eigenständige Entwicklung eines Programms zur Klassifikation von Dreiecken. Dabei sollt ihr alle Phasen des Software Life Cycle nacheinander durchlaufen (siehe auch Skriptkapitel 7), was durch die folgende Gliederung unterstützt werden soll. Die Klassifikation der einzelnen Dreiecke kann gemäß Aufgabe 1 erfolgen. Innerhalb dieser Aufgabe bezeichnen wir die drei Seiten des Dreiecks mit  $a$ ,  $b$  und  $c$ .

### 1. Aufgabenstellung

Entwickle ein Programm, das drei Zahlen für die Seiten  $a$ ,  $b$  und  $c$  einliest. Das Programm soll überprüfen, ob es sich um ein Dreieck handelt und ggf. die zutreffenden Dreiecksklassen ausgeben.

**Beispiel:** Eingabe: 3, 4, 5 Ausgabe: rechtwinkliges Dreieck

Das Programm soll auch erkennen, falls getätigte Eingaben unsinnig sind. Hierzu zählt beispielsweise die Eingabe negativer Seitenlängen.

**Hinweis:** Bei der Programmentwicklung kann davon ausgegangen werden, dass immer nur ganze Zahlen vorkommen, sodass der Datentyp `int` verwendet werden kann.

### 2. Pflichtenheft

Aufgabe : Klassifikation von Dreiecken  
Eingabe : drei Zahlen für die Seiten  $a$ ,  $b$  und  $c$ , Datentyp `int`  
Ausgabe : Die jeweilige(n) Klasse(n) des Dreiecks  
Sonderfälle: Zahlenwerte  $\leq 0$  für eine oder mehrere Seiten

### 3. Testdaten

Testfall	$a$	$b$	$c$	Dreiecksklasse(n)
1	1	2	4	kein Dreieck
2	9	9	9	gleichseitig, gleichschenkelig
3	4	4	6	gleichschenkelig
4	4	3	3	gleichschenkelig
5	3	4	5	rechtwinklig
6	3	4	6	gewöhnlich
7	10	11	12	gewöhnlich
8	3	-1	5	fehlerhafte Eingabe
9	0	1	7	fehlerhafte Eingabe

#### 4. Implementierung

```
Klassifikation von Dreiecken
Variablen: Integer: a, b, c
Einlesen von Seite a, Einlesen von Seite b, Einlesen von Seite c
wenn a > 0 und b > 0 und c > 0
dann wenn a+b > c und a+c > b und b+c > a
    dann Ausgabe des Textes: reguläres Dreieck
        Test auf gleichseitiges Dreieck
        Test auf gleichschenkliges Dreieck
        Test auf rechtwinkliges Dreieck
    sonst Ausgabe des Textes: kein Dreieck
sonst Ausgabe des Textes: Mindestens eine Eingabe ist fehlerhaft

Test auf gleichseitiges Dreieck
    wenn a = b und b = c
    dann Ausgabe des Textes: Gleichseitiges Dreieck

Test auf gleichschenkliges Dreieck
    wenn a = b oder b = c oder a = c
    dann Ausgabe des Textes: Gleichschenkliges Dreieck

Test auf rechtwinkliges Dreieck
    wenn  $a^2 + b^2 = c^2$  oder  $b^2 + c^2 = a^2$  oder  $a^2 + c^2 = b^2$ 
    dann Ausgabe des Textes: Rechtwinkliges Dreieck
```

**Handsimulation:** Trage in nachfolgender Tabelle ein, welche Ausdrücke ausgewertet und welche Ausgaben getätigt werden.

Eingabe: 2 3 4

Ausdrücke und Variablen	Zuweisung	Auswertung	Ausgabe
Variable a: Wert: undef.	2		
Variable b: Wert: undef.	3		
Variable c: Wert: undef.	4		
$a > 0$ und $b > 0$ und $c > 0$		wahr	regulär
$a = b$ und $b = c$		falsch	
$a = b$ oder $a = c$ oder $b = c$		falsch	
$a^2 + b^2 = c^2$ oder $b^2 + c^2 = a^2$ oder $a^2 + c^2 = b^2$		falsch	



## 5. Kodierung

Unsere Kodierung sieht wie folgt aus (teilweise etwas komprimiert):

```
1  #include <stdio.h>
2
3  int main ( int argc, char** argv)
4  {
5      // fuer jede seite eine variable nebst initialisierung
6      int a = -1, b = -1, c = -1;
7
8      // eingabeaufforderung und die seiten einlesen
9      printf( "Bitte Seite a eingeben: " );scanf( "%d",&a );
10     printf( "Bitte Seite b eingeben: " );scanf( "%d",&b );
11     printf( "Bitte Seite c eingeben: " );scanf( "%d",&c );
12
13     // damit wir wissen, was vor sich geht
14     printf( "\nEingabe: a=%d b=%d c=%d\n\n", a, b, c );
15
16     // pruefung auf korrekte eingabe
17     if ((a > 0) && (b > 0) && (c > 0))
18     {
19         // alles OK; erster test: dreieck oder keines
20         if ((a+b > c) && (a+c > b) && (b+c > a))
21         {
22             printf("a, b und c bilden ein Dreieck\n");
23
24             // zweiter test: gleichseitiges dreieck
25             if ((a == b) && (b == c))
26                 printf("Das Dreieck ist gleichseitig\n");
27
28             // dritter test: gleichschenkliges dreieck
29             if ((a == b) || (b == c) || (a == c))
30                 printf("Das Dreieck ist gleichschenkelig\n");
31
32             // vierter test: rechtwinkliges dreieck
33             if ((a*a + b*b == c*c) || (a*a + c*c == b*b)
34                 || (b*b + c*c == a*a))
35                 printf("Das Dreieck ist rechtwinklig\n");
36         }
37         else printf("a, b und c bilden kein Dreieck!\n");
38     }
39     else printf("Mindestens eine Seite ist fehlerhaft\n");
40
41     // fertig
42     return 0;
43 }
```

**Handsimulation:** Im Sinne eines effizienten Arbeitens ist es sehr lohnenswert, das auf Papier entwickelte C-Programm *vor* dem Eintippen mittels einer Handsimulation für einige ausgewählte Testdaten zu überprüfen. Trage hierzu in nachfolgender Tabelle (von oben nach unten) ein, welche Ausdrücke ausgewertet und welche Ausgaben getätigt werden. Eine mögliche Testeingabe lautet: 3, 4, 5.

Eingabe: 3, 4, 5

Zeile	Variablen	Aktion	Resultat/Effekt
6	.....	Definition <code>int a,b,c</code>	Anlegen und Initialisieren
9	<code>a=-1 b=-1 c=-1</code>	<code>scanf( "%d", &amp; a )</code>	Eingabe 3: <code>a=3</code>
10	<code>a= 3 b=-1 c=-1</code>	<code>scanf( "%d", &amp; a )</code>	Eingabe 4: <code>b=4</code>
11	<code>a= 3 b= 4 c=-1</code>	<code>scanf( "%d", &amp; a )</code>	Eingabe 5: <code>c=5</code>
14	<code>a= 3 b= 4 c= 5</code>	<code>printf(...)</code>	Ausgabe: <code>a=3 b=4 c=5</code>
17	.....	<code>a&gt;0 &amp;&amp; b&gt;0 &amp;&amp; c&gt;0</code>	wahr
20	.....	<code>a+b&gt;c &amp;&amp; a+c&gt;b &amp;&amp; b+c&gt;a</code>	wahr
22	.....	<code>printf(...)</code>	Ausgabe: ein Dreieck
25	.....	<code>a==b &amp;&amp; b==c</code>	falsch
29	.....	<code>a==b    a==c    b==c</code>	falsch
33	.....	<code>a*a + b*b == c*c    ...</code>	wahr
35	.....	<code>printf(...)</code>	Ausgabe: rechtwinklig
42	.....	<code>return 0</code>	Programmende

## 6. Speicherorganisation

Um ein besseres Gefühl für die internen Dinge eines C-Programms zu bekommen, sollt ihr diesmal auch eine „Speicherkarte“ erstellen, deren Einträge *üblicherweise* absteigend sortiert sind (die großen Adressen oben, die kleinen unten). Für alle Variablen und Funktionen sind jeweils ihre Namen und Adressen einzutragen. Bei Variablen ist ferner anzugeben, wie viele Bytes sie im Arbeitsspeicher belegen. Hinweise findet ihr in der ersten Programmieraufgabe aus Aufgabenblatt 3 sowie in Kapitel 8 des Skriptes. Verwende folgende Tabelle, in der bereits zwei Beispiele (fast vollständig) eingetragen sind:

Adresse	Name	Typ	Größe
0x00401BE4	<code>printf</code>	Funktion	
0x00401BDC	<code>scanf</code>	Funktion	
0x00401418	<code>main</code>	Funktion	
0x0028FF1C	<code>a</code>	Variable: <code>int</code>	4 Bytes
0x0028FF18	<code>b</code>	Variable: <code>int</code>	4 Bytes
0x0028FF14	<code>c</code>	Variable: <code>int</code>	4 Bytes

Der zugehörige Teil des C-Programms sieht wie folgt aus, den wir beispielsweise hinter Programmzeile 7 oder Zeile 40 einfügen können:

```
// speichersegmente ausgeben
// zuerst die funktionen
printf( "Adressen der Funktionen:\n" );
printf( "\tmain: \t%p\n", main );
printf( "\tprintf: \t%p\n", printf );
printf( "\tscanf: \t%p\n", scanf );

// und jetzt noch die variablen
printf("Adressen und Groessen der Variablen:\n");
printf("\ta: %p %d Bytes\n", &a, sizeof( a ) );
printf("\tb: %p %d Bytes\n", &b, sizeof( b ) );
printf("\tc: %p %d Bytes\n", &c, sizeof( c ) );
```

# Übungspaket 5

## Abstrakte Programmierung

---

### Übungsziele:

1. Strukturierung einer gegebenen Aufgabe,
2. Bearbeitung von Arbeitsabläufen des alltäglichen Lebens mittels Struktogrammen
3. und der Methode der Schrittweisen Verfeinerung

### Skript:

Kapitel: 11 bis 14

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket betrachten wir Vorgänge aus dem alltäglichen Leben, die letztlich nichts anderes als Algorithmen sind und die wir daher mittels Struktogrammen und der Methode der Schrittweisen Verfeinerung beschreiben wollen. Dabei geht es uns mehr um die Anwendung der erlernten Methoden und *nicht* um eine ultra präzise Beschreibung der alltäglichen Situationen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Datentypen und Werte

Welche Datentypen werden üblicherweise in der Programmierung verwendet? Gebe zu allen Datentypen ein paar Beispiele für mögliche Werte.

Datentyp	Beispielhafte Werte
Zeichen ( <code>char</code> )	'a', 'b', 'X', '8', '.', ' '
Ganzzahlen ( <code>integer/int</code> )	-1, 27, 4711, 815, 0
Gleitkommazahlen ( <code>double</code> )	123.456, 10e4
Zeichenketten ( <code>string</code> )	"Hallo Peter", "Ciao, Inge"

## Aufgabe 2: Anweisungen

Welche fünf grundlegenden Anweisungsarten haben wir in Skript und Vorlesung besprochen?

Nr.	Anweisung	Beschreibung
1.	Zuweisung	Variablen einen Wert geben, den man vorher ausrechnet.
2.	Ein- und mehrfache Fallunterscheidung	Um je nach Variablenwert und Ergebnis einer Berechnung unterschiedlich im Programm weiter zu machen.
3.	<code>for</code> -Schleife	Um die selben Anweisungen öfters auszuführen, ohne sie erneut hinschreiben zu müssen. Die Bedingung wird <i>immer vor</i> dem Schleifenanfang überprüft. Im Software Engineering steht die Zahl der Wiederholungen üblicherweise <i>vor</i> dem Eintritt in die Schleife fest.
4.	<code>while</code> -Schleife	Wie oben. Nur geht man im Software Engineering davon aus, dass sich die Zahl der Schleifendurchläufe erst <i>innerhalb</i> der Schleife ergibt.
5.	<code>do-while</code> -Schleife	Wie oben. Aber die Bedingung wird <i>immer</i> am Ende eines Schleifendurchlaufs überprüft. Bei diesem Schleifentyp werden die Anweisungen also mindestens einmal ausgeführt.

## Aufgabe 3: Abstrakte Programmierung: warum?

Erkläre mit eigenen Worten, warum wir immer mit der abstrakten Programmierung und nicht gleich mit dem Eintippen eines kleinen C-Programms anfangen.

Die Versuchung ist groß, gleich mit der Kodierung anzufangen. Gerade Anfänger erliegen ihr besonders häufig und scheitern, da sie so gar nicht recht wissen, *was* sie eigentlich programmieren sollen. Daher versuchen wir erst einmal, das *Was* mittels des Konzeptes der abstrakten Programmierung (Schrittweisen Verfeinerung oder Struktogramme) zu klären. Dies geht insbesondere umgangssprachlich sehr gut, da man nicht an eine spezielle Syntax oder Schreibweise gebunden ist. Erst wenn das *Was* geklärt ist, beschäftigen wir uns mit dem *Wie*, also der Umsetzung in eine konkrete Programmiersprache.

## Aufgabe 4: Fallunterscheidungen

Beschreibe mit eigenen Worten, warum Fallunterscheidungen notwendig sind und illustriere am Beispiel wie diese in der abstrakten Programmierung aussehen.

Fallunterscheidungen sind immer dann nötig, wenn Ergebnisse oder Zustände geprüft werden müssen, um in Abhängigkeit von diesen weitere Schritte einzuleiten

Raumhelligkeit regeln

```
wenn  es ist zu dunkel
dann  wenn licht ist aus
      dann  Schalte das Licht an
      sonst besorge zusätzliche oder hellere Lichtquellen
sonst Nichts tun, denn es ist hell genug
```

## Aufgabe 5: Schleifen

Beschreibe mit eigenen Worten, warum Schleifen notwendig sind und illustriere am Beispiel wie diese in der abstrakten Programmierung aussehen.

Schleifen werden benötigt, um verschiedene Anweisungen und Operation zu wiederholen.

Berechnung von Summe und Mittelwert

```
setze summe = 0
für i = 1 bis Zahl der Messwerte
wiederhole setze summe = summe + Messwert  $x_i$ 
Ausgabe Text "Summe der Messwerte: " Summe
Ausgabe Text "Mittelwert: " Summe / Zahl der Messwerte
```

## Teil II: Quiz

---

### Aufgabe 1: Beispiel: Schleife

Was macht nebenstehendes Programm?

Es berechnet die Summe der ganzen Zahlen zwischen -3 und 3 und gibt sie aus.
--

setze summe = 0
für i = -3 bis 3 schrittweite 1
setze summe = summe + i
Drucke "resultat: " summe

### Aufgabe 2: Beispiel: Fallunterscheidung

Was macht nebenstehendes Programm?

Wenn i kleiner als j oder j negativ ist, wird i auf den Wert -j gesetzt. In allen anderen Fällen erhält i den Wert j - i.
---

Lese Variablen i und j	
i > j oder j < 0	
wahr	falsch
setze i = -j	setze i = j - i
Drucke "Resultat: " i	

## Teil III: Fehlersuche

---

### Aufgabe 1: Suche von Semantikfehlern

Das folgende Programm soll die Summe aller ungeraden Zahlen von -1 bis 13 (einschließlich) ausrechnen. Doch offensichtlich hat unser Aushilfsprogrammierer DR. NO ein paar Fehler gemacht. Finde die vier Fehler, erkläre und korrigiere sie.

setze summe = 0
für i = -3 bis 13 schrittweite 1
setze summe = summe - i
Drucke "resultat: " summe + 2

Fehler	Erläuterung	Korrektur
-3	Die Summenbildung soll bei -1 beginnen	-1
schrittweite 1	Da nur die ungeraden Zahlen genommen werden sollen, muss die Schrittweite 2 genommen werden.	schrittweite 2
-i	Da die <i>Summe</i> berechnet werden soll, muss addiert werden.	+
summe+2	Die Summe wurde bereits in der Schleife berechnet, sodass der Term +2 zu einem Fehler führt.	summe

**Eine mögliche Korrektur:**

setze summe = 0
für i = -1 bis 13 schrittweite 2
setze summe = summe + i
Drucke "resultat: "summe



## Teil IV: Anwendungen

---

Von den folgenden 4 Aufgaben sind mindestens drei zu bearbeiten und jede der beiden Methoden ist mindestens einmal zu verwenden. Bei allen Aufgaben gibt es kein Richtig oder Falsch. Anhand der Aufgaben sollt Ihr lernen, Arbeitsabläufe mittels strukturierter Methoden zu beschreiben und diese in einigen *wenigen* Stufen schrittweise zu verfeinern, bis sie wirklich klar sind. Diesen Prozess könnte man bei jedem Detail nahezu endlos weiterführen; findet also eine sinnvolle Ebene (ein sinnvolles Abstraktionsniveau), bei dem ihr abbrecht, weil die Beschreibung hinlänglich klar ist. Da es uns um die Prinzipien geht, sollte jede Beschreibung aus maximal 15-20 Zeilen bestehen.

### Aufgabe 1: Auto betanken

**Aufgabenstellung:** Ihr fahrt mit dem Auto und der Tank ist fast leer. Beschreibe, wie ihr an die Tankstelle fahrt, tankt und bezahlt.

#### Benzin tanken

- An die Tankstelle heran fahren
- Zapfsäule aussuchen
- Tankklappe entriegeln
- Benzin einfüllen
- Tankklappe schließen
- Bezahlen
- Wegfahren

#### An die Tankstelle heran fahren

- Alle Zapfsäulen mit dem richtigen Benzinangebot vergleichen.
- Zapfsäule mit der kürzesten Warteschlange auswählen
- Zur Zapfsäule fahren

#### Benzin einfüllen

- Zapfpistole aus Halterung nehmen
- Zapfpistole in den Tankeinlass einführen
- Zapfpistole betätigen bis Tank voll ist
- Zapfpistole in die Halterung zurückbringen

#### Bezahlen

- Je nach Bargeld und Plänen mit Bargeld oder Karte bezahlen
- Quittung einstecken

## Aufgabe 2: „Kaffeeklatsch“ in der Kneipe

**Aufgabenstellung:** Du sitzt mit ein paar Freunden zusammen in der Kneipe, ihr seid am Klönen über dies, das und die Lehrveranstaltung. Je nach Zusammensetzung der Gruppe sollst du verschiedene Getränke bestellen. Auch sollte die Auswahl davon abhängig sein, ob die Schwiegereltern anwesend sind oder nicht. Je nach Stimmungslage sind zwei bis vier Runden zu bestellen.

Kneipenabed

```
setze Rundenzaehler = 0
wiederhole
    setze Rundenzaehler = Rundenzaehler + 1
    wenn Schwiegereltern anwesend
    dann Bestellung: Wein und Bowle
    sonst wenn Frauen anwesend
        dann Bestellung: Sekt
        sonst Bestellung: Bier
bis Rundenzaehler = 4 oder (Stimmung nicht gut und Rundenzaehler >= 2)
```

Stimmung nicht gut

```
Zahl der lachenden Personen < 50%
```

## Aufgabe 3: Organisation eines Lernabends

**Aufgabenstellung:** Für die anstehenden Klausuren trifft ihr euch als Lerngruppe immer reihum bei einem der Mitglieder. Beschreibe, wie du den heutigen Abend organisierst:

Lernabend organisieren

```
Vorbereitungen treffen
wiederhole schaue Sport
solange noch niemand da ist // die anderen schauen auch Sport
wiederhole 1 stunde lernen
    15 minuten pause mit essen und trinken
bis alle muede sind
alle rausschmeissen
```

Vorbereitungen treffen

```
Knabberzeug, Cola und Kaffee einkaufen
Bestell- und Rabattzetten für Pizzadienst organisieren
Fernseh-Fernbedienung verstecken
Aufräumen
```

## Aufgabe 4: Reparieren eines Fahrradschlauchs

**Aufgabenstellung:** Was muss man alles in welcher Reihenfolge machen, wenn man einen kaputten Fahrradschlauch reparieren soll?

Fahrradschlauch reparieren

- Rad ausbauen
- Mantel mit Mantelhebern abziehen
- Schlauch entnehmen
- Loch suchen und finden
- Loch schließen
- Schlauch, Mantel und Rad wieder zusammenbauen
- Rad einbauen
- Rad aufpumpen

Rad ausbauen

- Mit den richtigen Schlüsseln die Muttern lösen
- Bremse lösen
- wenn Hinterrad
- dann Kette abnehmen
- Rad aus der Gabel entnehmen

Loch suchen und finden

- Schlauch aufpumpen
- wenn Loch nicht sofort sichtbar oder hörbar
- dann aufgepumpten Schlauch in einen Wassereimer mit Wasser halten
- wiederhole Schlauch drehen
- bis aufsteigende Blasen sichtbar sind.

# Übungspaket 6

## Arbeiten mit Syntaxdiagrammen

---

### Übungsziele:

Verstehen, Anwenden und Erstellen von Syntaxdiagrammen

### Skript:

Kapitel: 20

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

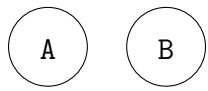

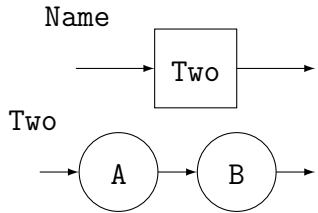
Für den Prozess des Lernens ist es anfangs meist sehr hilfreich, wenn man „ungefähre“, allgemein verständliche Erklärungen angeboten bekommt. Aber in der Technik muss man es früher oder später ganz genau wissen. Dies ist insbesondere beim Programmieren der Fall. Hier helfen vor allem die Syntaxdiagramme, mittels derer eine Programmiersprache (oder auch ein Stück Hardware) sehr präzise beschrieben werden kann. Aufgrund der Wichtigkeit der Syntaxdiagramme ist ihnen dieses Übungspaket gewidmet.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Elemente von Syntaxdiagrammen

In der Vorlesung haben wir den Aufbau und die Funktion von Syntaxdiagrammen am Beispiel von Variablennamen kennen gelernt. Diese Syntaxdiagramme bestehen aus lediglich drei Dingen. Wie heißen sie, wie werden sie visualisiert und was bedeuten sie?

1. Der erste Teil besteht aus Kreisen (oder Ellipsen). Diese Elemente nennt man auch Terminalsymbole, denn man muss das, was in ihnen steht, auch Tippen. Das rechte Beispiel zeigt zwei Terminalsymbole, die ein A oder B verlangen.
2. Der zweite Teil besteht aus Rechtecken, die man auch Nichtterminalsymbole nennt und die woanders weiter spezifiziert sein müssen. Das rechte Beispiel zeigt ein Nichtterminalsymbole namens **Two**.
3. Der dritte Teil sind Pfeile, die die einzelnen Terminal- und Nichtterminalsymbole untereinander verbinden. Das rechts stehende Beispiel zeigt, wie Namen definiert werden können, die aus einem A gefolgt von einem B bestehen.

## Aufgabe 2: Aufbau von Syntaxdiagrammen

Erkläre mit eigenen Worten, wie Syntaxdiagramme aufgebaut sind und wie sie insbesondere die Anwendung des Konzeptes der Schrittweisen Verfeinerung erlauben.

Syntaxdiagramme bestehen aus drei Komponenten. Erstens: Terminalsymbole werden als Kreise oder Ellipsen (manchmal auch als Rechtecke mit abgerundeten Ecken) dargestellt. Zweitens: Nichtterminalsymbole werden als Rechtecke mit *nicht* abgerundeten Ecken dargestellt. Drittens: Terminal- und Nichtterminalsymbole werden mit gerichteten Pfeilen, also Strichen mit Anfangs- und Endpunkt, miteinander verbunden. Diese Pfeile gehen von einem Rechteck zum nächsten oder beginnen bzw. enden an einem anderen Pfeil.

Das Lesen eines Syntaxdiagramms ist ganz einfach: Man startet beim Namen des Syntaxdiagramms (bei uns immer oben links) und geht den Pfeilen entlang, die man zuvor ausgewählt hat. In gleicher Weise muss man das Syntaxdiagramm jedes angetroffenen Nichtterminalsymbols durchlaufen. Eine vorhandene Zeichenfolge ist dann *und nur dann* korrekt, wenn es für diese einen vollständigen Weg durch das Syntaxdiagramm gibt.

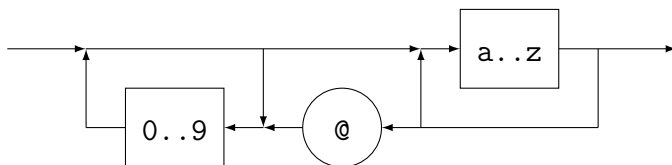
## Teil II: Quiz

---

### Aufgabe 1: Beispiel Aktenzeichen

Die Regeln, nach denen ein Aktenzeichen für ein virtuelles Amt generiert werden können, sind gemäß folgendem Syntaxdiagramm definiert:

Aktenzeichen



Welche der folgenden sechs gegebenen Aktenzeichen sind korrekt bzw. fehlerhaft?

Beispiel	korrekt	fehlerhaft
anton@2low	<input checked="" type="checkbox"/>	<input type="checkbox"/>
peter	<input checked="" type="checkbox"/>	<input type="checkbox"/>
frau@mustermann	<input type="checkbox"/>	<input checked="" type="checkbox"/>
4711x@0815y	<input checked="" type="checkbox"/>	<input type="checkbox"/>
_ingo	<input type="checkbox"/>	<input checked="" type="checkbox"/>
4711@5@0815	<input type="checkbox"/>	<input checked="" type="checkbox"/>

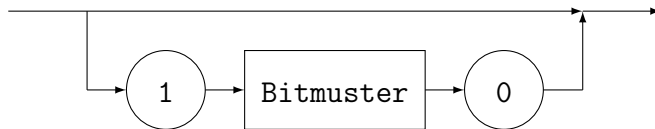
Konstruiere je drei eigene korrekte bzw. fehlerhafte Beispiele:

korrekte Beispiele	fehlerhafte Beispiele
mann@2musterfrau	a@@1x
4711sunshine	hierbinich@
a@1b@2x	ey@1000000

## Aufgabe 2: Beispiel Bitmuster

Das folgende Syntaxdiagramm definiert die Regel, nach denen gültige Bitmuster einer hypothetische Anwendung generiert werden können:

Bitmuster



Welche der folgenden sechs gegebenen Bitmuster sind korrekt bzw. fehlerhaft?

Beispiel	korrekt	fehlerhaft
10	<input checked="" type="checkbox"/>	<input type="checkbox"/>
01	<input type="checkbox"/>	<input checked="" type="checkbox"/>
100	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1100	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<leeres Bitmuster>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
111100000	<input type="checkbox"/>	<input checked="" type="checkbox"/>

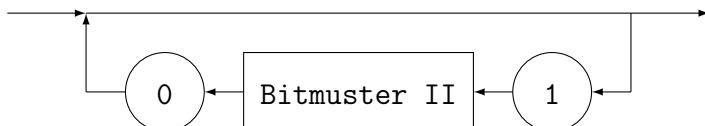
Konstruiere je drei eigene korrekte bzw. fehlerhafte Beispiele:

korrekte Beispiele	fehlerhafte Beispiele
111000	1010
11110000	110000
1111100000	11100

## Aufgabe 3: Beispiel Bitmuster II

Das folgende Syntaxdiagramm ist eine kleine Abwandlung des vorherigen Syntaxdiagramms **Bitmuster** aus Quizaufgabe 2.

Bitmuster II



Welche der folgenden sechs gegebenen Bitmuster sind korrekt bzw. fehlerhaft?

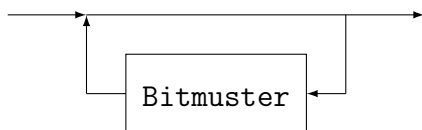
Beispiel	korrekt	fehlerhaft
10	<input checked="" type="checkbox"/>	<input type="checkbox"/>
01	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1010	<input checked="" type="checkbox"/>	<input type="checkbox"/>
0101	<input type="checkbox"/>	<input checked="" type="checkbox"/>
10100	<input type="checkbox"/>	<input checked="" type="checkbox"/>
10101	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Konstruiere vier Beispiele, die nach obigem Syntaxdiagramm **Bitmuster II** korrekt, aber nach dem Syntaxdiagramm **Bitmuster** aus Quizaufgabe 2 fehlerhaft sind:

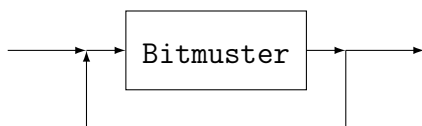
1.  2.  3.  4.

Welche der folgenden Syntaxdiagramme ist mit **Bitmuster II** identisch?

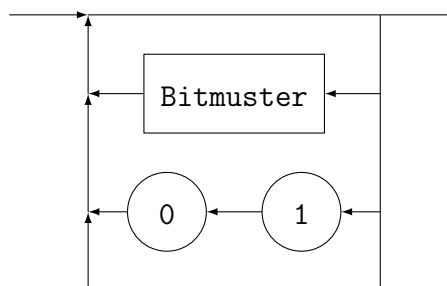
Bitmuster III



Bitmuster IV



Bitmuster V



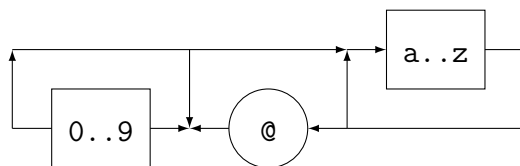


## Teil III: Fehlersuche

### Aufgabe 1: Kleine Fehlersuche

Im folgenden Syntaxdiagramm befinden sich vier kleine Fehler. Finde, erkläre und korrigiere diese.

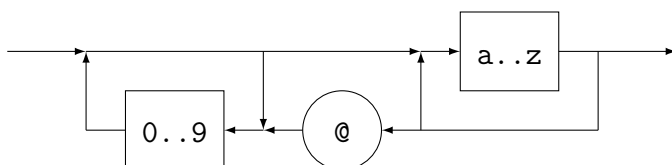
Label



Fehler	Erläuterung	Korrektur
kein „Eingang“	Das Syntaxdiagramm hat keinen „Eingang“. Dieser sollte vermutlich ganz links oben sein, da dies so <i>üblich</i> ist.	links oben
kein „Ausgang“	Das Syntaxdiagramm hat keinen „Ausgang“. Dieser sollte vermutlich ganz rechts oben sein, da dies so <i>üblich</i> ist.	rechts oben
„Sackgasse“	Am „Schnittpunkt“ zwischen 0..9 und @ treffen alle Pfeile aufeinander. Wenn man hier ankommt, gibt es kein Weiterkommen. Mindestens einer der Pfeile muss gedreht werden.	Pfeil drehen
unerreichbar	Das Nichtterminalsymbol 0..9 kann nicht erreicht werden. Daher muss einer der beiden Pfeile (zur rechten bzw. linken) gedreht werden.	Pfeil drehen

**Eine mögliche Korrektur:**

Label

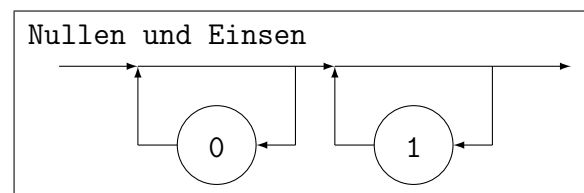


## Teil IV: Anwendungen

### Aufgabe 1: Ein erstes, einfaches Syntaxdiagramm

In dieser Aufgabe sind nur Folgen von Nullen (0) und Einsen (1) erlaubt, die mit beliebig vielen Nullen anfangen, mit beliebig vielen Einsen aufhören und dazwischen nichts anderes haben. Konstruiere ein Syntaxdiagramm für die folgenden Beispiele:

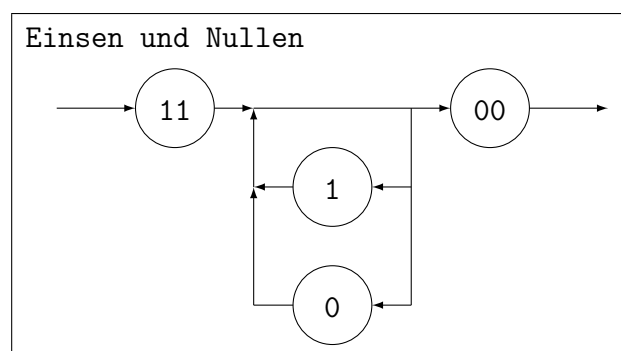
korrekte Beispiele	fehlerhafte Beispiele
00	10
11	001011
0001	00000010
011111	1000011111
000001111	1010101010



### Aufgabe 2: Ein zweites, einfaches Syntaxdiagramm

Nun sind beliebige Folgen von Nullen (0) und Einsen (1) erlaubt. Nur muss eine korrekte Folge mit zwei Einsen (1) anfangen und mit zwei Nullen (0) aufhören. Konstruiere ein Syntaxdiagramm für die folgenden Beispiele:

korrekte Beispiele	fehlerhafte Beispiele
1100	1000
11000	1100010
11100	01111100
11010101000	1011110000

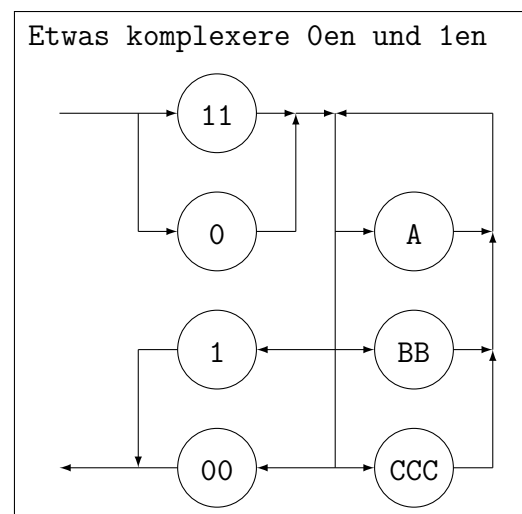


## Aufgabe 3: Ein etwas komplexeres Syntaxdiagramm

Konstruiere ein Syntaxdiagramm für die folgenden Beispiele. Die Sequenzen müssen wie folgt aufgebaut sein:

1. Sie muss mit zwei Einsen (1) oder einer Null (0) anfangen.
2. Sie muss mit einer Eins (1) oder zwei Nullen (0) aufhören.
3. Dazwischen dürfen sich beliebige Folgen von einem A, zwei Bs oder drei Cs befinden.

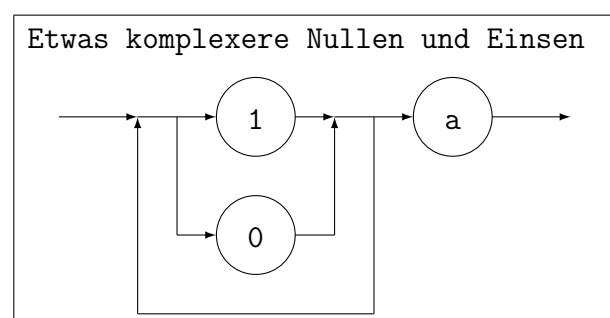
korrekte Beispiele	fehlerhafte Beispiele
11BBACCC00	1ABBCCC00
0BBACCC00	11CC00
11AAAA00	0ABAB1
1100	11ABB1CCC00



## Aufgabe 4: Ein letztes Syntaxdiagramm

Die Schwierigkeit dieser letzten Aufgabe besteht darin, dass nur Beispiele und keine Definition gegeben ist. Versuche zunächst das zugrunde liegende Prinzip aus den Beispielen abzuleiten. Konstruiere anschließend ein Syntaxdiagramm, das die folgenden Beispiele richtig klassifiziert:

korrekte Beispiele	fehlerhafte Beispiele
0a	a
1a	0
001a	0a1
101a	001a0
10111a	a00000
111a	0001110a1010



# Übungspaket 7

## Angemessenes Formatieren von C-Programmen

---

### Übungsziele:

1. Gute Layout-Struktur durch Einrücken
2. Richtiges Verwenden von Kommentaren

### Skript:

Kapitel: 19

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Das Kodieren eines Algorithmus in ein C-Programm und das Eintippen in den Rechner ist eigentlich gar nicht so problematisch. Schwierig wird's nur bei der Fehlersuche, wenn das Programm nicht das macht, was es machen soll. Noch schwieriger wird es, wenn dann der Betreuer „mal eben“ den Fehler finden soll. Hier helfen Kommentare und angemessenes Einrücken, was wir hier ein wenig üben wollen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Kommentare

In der Programmiersprache C gibt es zwei Möglichkeiten, Kommentare zu verwenden. Erkläre kurz die Syntax und zeige ein Beispiel.

### Möglichkeit 1:

Syntax	Die erste Möglichkeit erfordert sowohl öffnende <code>/*</code> als auch schließende <code>*/</code> „Klammern“ und kann sich über mehrere Zeilen erstrecken.
--------	---

Beispiel	<pre>int i /* hier ist ein kommentar       der hier zu ende ist */ = -1;</pre>
----------	--

### Möglichkeit 2:

Syntax	Die zweite Möglichkeit besteht in <code>//</code> . Derartige Kommentare beginnen mit <code>//</code> und gehen <i>immer</i> bis zum Ende der Zeile. Bei mehrzeiligen Kommentaren ist das <code>//</code> entsprechend in jeder Zeile zu wiederholen.
--------	---

Beispiel	<pre>int i =      // variablendefinition           -1; // initialisierung</pre>
----------	---

## Aufgabe 2: Geschachtelte Kommentare

Erkläre *kurz*, wie man Kommentare schachteln kann und was dabei passiert.

Natürlich kann man Kommentare schachteln wie man will, nur muss man verstehen, was dabei passiert. Die wesentliche Frage ist immer, welche Kommentarform die äußere ist.

#### 1. `//`-Kommentare:

Egal, was man hier hineinschreibt, der Kommentar endet am Zeilenende. Auch wenn die schließende Klammer fehlt, ist dieser am Zeilenende zu Ende.

*Beispiel:* `// aeusserer kommentar /* innerer kommentar`

#### 2. `/* ... */`-Kommentare:

Egal, was man hier hineinschreibt, der Kommentar ist beim Auftreten des ersten `*/` zu Ende, egal ob dies in der selben oder einer späteren Zeile der Fall ist.

*Beispiel:* `/* aeusserer kommentar */ kein kommentar mehr */`

## Aufgabe 3: Angemessenes Einrücken

Erkläre kurz in eigenen Worten, *warum* wir als Lehrpersonal so sehr auf eine angemessene Form des Einrückens Wert legen.

Ein C-Programm besteht in der Regel aus Variablendefinitionen und Anweisungsblöcken. Ein Charakteristikum von Anweisungsblöcken ist, dass sie aus verschiedenen Elementen wie Fallunterscheidungen, Schleifen und weiteren Anweisungen bestehen, die ineinander verschachtelt sein können. Daraus ergibt sich eine logische Struktur. Für das *Finden* möglicher Fehler ist es für *alle* sehr hilfreich, wenn man die intendierte Struktur auf den ersten Blick erfasst. Damit vermeidet man nicht nur Fehler, sondern sieht sehr schnell, wo möglicherweise Klammern fehlen oder zu viele gesetzt sind.

## Aufgabe 4: Recherche

Recherchiere etwa fünf bis zehn Minuten im Web zum Thema „Einrücken in C“ und notiere deine zwei Favoriten. Mögliche Schlagwörter sind: `einruecken c programm`

Die meisten von uns favorisieren einen Einrückstil, der sehr stark dem Allman/BSD/„East Coast“ Stil ähnelt.

Ihr könnt selbstverständlich euren Einrückstil frei wählen, solange er auch tatsächlich einrückt (also nicht linksbündig schreibt), konsistent ist und alle Beteiligten damit zurecht kommen.

## Teil II: Quiz

---

### Aufgabe 1: Finden von Kommentaren

Im folgenden (recht sinnfreien) Text befinden sich einige Kommentare. Schau dir die Texte an und unterstreiche alle Kommentare einschließlich der „Kommentarklammern“. Dabei ist jede Zeile für sich zu behandeln, d.h. eventuell noch nicht abgeschlossene Kommentare aus vorherigen Zeilen haben keinen Einfluss. Korrigiere die Kommentare, sofern dir dies sinnvoll, logisch oder naheliegend erscheint.

```
1 heute ist ein /* schoener */ tag
2 gestern schien die sonne // den ganzen tag
3 /* das */ lernen der /* programmiersprache C */ ist
  anstrengend
4 // ich wuenschte, das studium waere schon fertig
5 ich habe /* sehr /* nette */ */ kommilitonen
6 beim hochschulsport findet man viele /** gute */ kurse
7 im sommer ist es warm, im winter /* nicht **/ immer
8 mathe ist /* oft * / anstrengend, /* physik */ leider auch
9 das lernen /* in der // gruppe */ macht spass
10 schoen, // bald */ ist wochenende
11 kommentare sind /* sehr * / bloed // finde ich
```

Im Text haben wir folgende Kommentare gefunden:

Zeile	Kommentar
1	/* schoener */
2	// den ganzen Tag
3	/* das */
3	/* programmiersprache C */
4	// ich wuenschte, dass studium waere schon fertig
5	/* sehr /* nette */
6	/** gute */
7	/* nicht **/
8	/* oft * / anstrengend, /* physik */
9	/* in der // gruppe */
10	// bald */ ist wochenende
11	/* sehr * / bloed // finde ich

Die im Text verwendete Schreibweise *legt nahe*, dass einige Tippfehler passiert sind, die folgende Korrekturen nahelegen:

Zeile	Korrektur	Begründung
5	<code>/* sehr nette */</code>	Kommentare kann man nicht schachteln
6	<code>/* gute */</code>	Vermutlich ein * zu viel
7	<code>/* nicht */</code>	Vermutlich ein * zu viel
8	<code>/* oft */</code>	vermutlich ein Leerzeichen zu viel
10	<code>/* bald */</code>	vermutlich Verwechslung von / und *
11	<code>/* sehr */</code>	vermutlich ein Leerzeichen zu viel

Diese Korrekturen ergeben folgenden Text:

```
1 heute ist ein /* schoener */ tag
2 gestern schien die sonne // den ganzen tag
3 /* das */ lernen der /* programmiersprache C */ ist
  anstrengend
4 // ich wuenschte, das studium waere schon fertig
5 ich habe /* sehr nette */ kommilitonen
6 beim hochschulsport findet man viele /* gute */ kurse
7 im sommer ist es warm, im winter /* nicht */ immer
8 mathe ist /* oft */ anstrengend, /* physik */ leider auch
9 das lernen /* in der // gruppe */ macht spass
10 schoen, /* bald */ ist wochenende
11 kommentare sind /* sehr */ bloed // finde ich
```



## Teil III: Fehlersuche

---

### Aufgabe 1: Fehlerhafte Kommentare finden

Im folgenden Programm befinden sich für seine Größe recht viele Kommentare. Leider hat der Programmierer hier und dort den einen oder anderen Fehler gemacht.

**Fehlerhaftes Programm:**

```
1  #include <stdio.h>          /* kommentar
2
3  int main( int argc, kommentar char **argv )
4      // hier ist ganz /* viel
5      kommentar sogar ueber */
6      // drei zeilen
7      { /* schon wieder ein /* einfacher */ kommentar */
8          int // kommentar mitten i; drin
9          if ( i == 2 )
10             hier auch printf( "true\n" );
11             else printf( "false\n" ); /* und jetzt ist aber schluss
12     }
```

Viele dieser Kommentare sind völlig sinnfrei. Aber dennoch, finde und korrigiere sie. Bedenke, dass es immer mehrere Möglichkeiten gibt, die Fehler zu korrigieren. Entscheide dich jeweils für eine. Bei richtig gesetzten Kommentaren sollte der Compiler folgendes Programm sehen, sofern er selbige entfernt hat:

**Programm, wie es ohne Kommentare sein sollte:**

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5          int i;
6          if ( i == 2 )
7              printf( "true\n" );
8          else printf( "false\n" );
9      }
```

Die Richtigkeit der durchgeführten Korrekturen lässt sich sehr einfach feststellen. Einfach mal das Kommando `cpp <datei.c>` ausführen, wobei `<datei.c>` natürlich durch den richtigen Dateinamen zu ersetzen ist. Am Anfang steht sehr viel, das alles mit `stdio.h` zu tun hat. Am Ende jedoch sollte das Programm richtig erscheinen (unter Umständen sind aber die Zeilen ein wenig verrutscht). Statt `cpp` kann natürlich auch `gcc -E` verwendet werden.

### Fehlerbeschreibung:

Zeile	Fehler	Erläuterung	Korrektur
1	<code>*/</code> fehlt	Die schließende Kommentarklammer <code>*/</code> fehlt.	<code>*/</code> ergänzen
3	<code>/*...*/</code> fehlen	„kommentar“ steht nicht in Kommentarklammern.	<code>/*...*/</code> ergänzen
4		Alles völlig korrekt.	
5	<code>/*</code> oder <code>//</code> fehlen	Der Kommentar wird zwar beendet, aber die öffnende Klammer fehlt.	<code>/*</code> oder <code>//</code> ergänzen
6		Alles völlig korrekt	
7	falsche Schachtelung	Durch <code>/*</code> ist der hintere Teil kein Kommentar, da man Kommentare so nicht schachteln kann.	neu anfangen
8	<code>i</code> ist im Kommentar	Die Variable <code>i</code> ist leider im Kommentar. Hier muss der Kommentar vorher beendet und anschließend wieder angefangen werden.	<code>*/</code> und <code>//</code> ergänzen
9		Alles völlig korrekt.	
10	<code>/*...*/</code> fehlen	Die ersten beiden Wörter sind zu viel. Diese müssen durch weitere Kommentarklammern „entfernt“ werden.	<code>/*...*/</code> ergänzen
11	<code>*/</code> fehlt	Die schließende Kommentarklammer <code>*/</code> fehlt entweder <code>*/</code> am Ende ergänzen oder <code>//</code> am Kommentaranfang.	<code>*/</code> ergänzen

Wir haben das schreckliche Programm jedenfalls wie folgt korrigiert:

### Korrigiertes Programm:

```
1  #include <stdio.h>          /* kommentar */
2
3  int main( int argc, /* kommentar */ char **argv )
4      // hier ist ganz /* viel
5      // kommentar sogar ueber */
6      // drei zeilen
7      { /* schon wieder ein /* einfacher */ /* kommentar */
8          int /* kommentar mitten */ i; // drin
9          if ( i == 2 )
10             /* hier auch */ printf( "true\n" );
11             else printf( "false\n" ); // und jetzt ist aber schluss
12      }
```

## Teil IV: Anwendungen

---

In diesem Aufgabenteil sollt ihr schlecht formatierte Programme durch Verwenden von Leerzeichen und Zeilenumbrüchen so umwandeln, dass deren *Struktur* sofort erkennbar ist.

### Aufgabe 1: Ein erstes, einfaches Programm

```
1 #          include<stdio.h>
2
3 int main(int argc,char **argv)
4 {
5     int i, j;
6     if(i<j)
7         for(i=0;i<10;i=i+1)
8             printf("ha ha\n");
9     else
10    if(i==j)
11        for(i=0;i<3;i=i+1)
12            for(j=0;j<10;j=j+1)
13                printf("ho ho\n");
14    else
15        printf("total blood\n");
16 }
```

Neue Formatierung:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i, j;
6     if (i < j )
7         for( i = 0; i < 10; i = i + 1 )
8             printf( "ha ha\n" );
9     else if( i == j )
10        for( i = 0; i < 3; i = i + 1 )
11            for( j = 0; j < 10; j = j + 1 )
12                printf( "ho ho\n" );
13        else printf( "total blood\n" );
14 }
```

Dies ist nur eine von vielen Möglichkeiten. Hauptsache die Programm*struktur* ist auf den ersten Blick *erkennbar*!

## Aufgabe 2: Ein zweites, einfaches Programm

```
1 #           include<stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i, j;
6     if(i<j)
7         for(i=0;i<10;i=i+1)
8             if(i==3)
9                 for(j=0;j<3;j=j+1)
10                    printf("ha ha\n");
11     else
12         printf("ho ho\n");
13     else
14         printf("total blood\n");
15 }
```

**Neue Formatierung:** (Konsistent, aber mit vielen geschweiften Klammern)

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i, j;
6     if (i < j)
7     {
8         for(i = 0; i < 10; i = i + 1)
9         {
10             if (i == 3)
11             {
12                 for(j = 0; j < 3; j = j + 1)
13                 {
14                     printf("ha ha\n");
15                 }
16             }
17             else {
18                 printf("ho ho\n");
19             }
20         }
21     }
22     else {
23         printf("total blood\n");
24     }
25 }
```

Ohne alle nicht unbedingt notwendigen geschweiften Klammern hätten wir folgendes, recht kompaktes Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j;
6      if (i < j)
7          for(i = 0; i < 10; i = i + 1)
8              if (i == 3)
9                  for(j = 0; j < 3; j = j + 1)
10                     printf("ha ha\n");
11                 else printf("ho ho\n");
12             else printf("total bloed\n");
13 }
```

### Aufgabe 3: Ein drittes, einfaches Programm

```
1      #      include /* eingabe */      <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i
6  ; if ( i
7  == 2 ) printf( "true\n" ); else printf
8  ( "false\n" );
9
10 }
```

Neue Formatierung:

```
1  #include <stdio.h>          /* eingabe */
2
3  int main( int argc, char **argv )
4  {
5      int i;
6      if ( i == 2 )
7          printf( "true\n" );
8      else printf( "false\n" );
9  }
```

Bei diesem Programm haben wir zwecks Übersichtlichkeit und erkennbarer Programmstruktur zusätzlich die Inhalte zwischen den Zeilen hin- und hergeschoben.

# Übungspaket 8

## Datentyp `int`

---

### Übungsziele:

1. Umgang mit dem Datentyp `int`,
2. Deklarationen von `int`-Variablen,
3. `int`-Konstanten
4. und `int`-Rechenoperationen.

### Skript:

Kapitel: 21

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Aufgrund seiner grundlegenden Bedeutung muss man sich mit dem Datentyp `int` möglichst frühzeitig und möglichst intensiv auseinandersetzen. Genau das machen wir in diesem Übungspaket ;-)

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Definition von int-Variablen

Zeige anhand einiger Beispiele, wie man `int`-Variablen definieren und initialisieren kann.

```
1 int i = 1;           // def. i,      value  =  1
2 int j = 2 * i;       // def. j,      value  =  2*i =  2
3 int k = j - i;       // def. k,      value  =  1
4 int m = -1, n = 1 - m; // def. m, n, values = -1, 2
```

## Aufgabe 2: Zahlenbasis

In der Programmiersprache C können `int`-Zahlen bezüglich dreier Basen angegeben werden. Erkläre jede gültige Basis anhand eines kleinen Beispiels:

1.  2.  3.

## Aufgabe 3: Eingabe

Zeige am Beispiel, wie `int`-Variablen eingelesen werden können:

## Aufgabe 4: Ausgabe

Erkläre anhand zweier Beispiele wie `int`-Variablen ausgegeben werden können.

1.  2.

## Aufgabe 5: Rechenoperationen

Welche fünf Rechenoperationen sind bei `int`-Ausdrücken erlaubt?

1.  2.  3.  4.  5.

## Aufgabe 6: Basis und Zahlenwert

Aus der Schule wissen wir, dass jede Zahl bezüglich einer Basis angegeben wird. In unserem „normalen“ Leben ist die Basis meist 10 (zehn), weshalb wir auch vom Dezimalsystem reden. Dieses Dezimalsystem ist bei uns in Deutschland (und vielen anderen westlichen Ländern) fast überall zu finden. Aber Länder wie Grossbritannien und USA geben nicht alle

Größen bezüglich eines Zehnerwertes an. Ein bekanntes Beispiel ist: 1 Fuss ist gleich 12 Inch bzw.  $1/3$  Yard.

Im *Dezimalsystem* hat die Ziffernfolge 123 den Wert:  $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$  bzw.  $1 \times 100 + 2 \times 10 + 3 \times 1$ . Im *Oktalsystem* hat die Ziffernfolge 234 den Wert:  $2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0$  bzw.  $2 \times 64 + 3 \times 8 + 4 \times 1$ , was dem Dezimalwert 156 entspricht.

**Wandle folgende Oktalzahlen in Dezimalzahlen um:**

Oktalzahl	C-Syntax	Dezimal
1	01	1
10	010	8
100	0100	64

**Wandle folgende Dezimalzahlen in Oktalzahlen um:**

Dezimal	Oktalzahl	C-Syntax
1	1	01
10	12	012
100	144	0144

## Aufgabe 7: Zahlendarstellung in C

Wie werden Dezimalzahlen in C angegeben?

wie im Leben

Wie werden Oktalzahlen in C angegeben?

mit einer vorlaufenden Null

Wie werden Hexadezimalzahlen in C angegeben?

mit 0x am Anfang

**Hexadezimalziffern:**

Dezimalwert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadezimalziffer	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
											A	B	C	D	E	F



## Teil II: Quiz

---

### Aufgabe 1: Umrechnen zwischen verschiedenen Basen

In dieser Übungsaufgabe geht es darum, Zahlen zwischen den einzelnen Zahlensystemen hin und her zu rechnen. Vervollständige folgende Tabelle, in der alle Zahlen einer Zeile den selben Wert haben sollen:

Basis							
16	12	10	9	8	7	3	2
F	13	15	16	17	21	120	1111
4E	66	78	86	116	141	2220	1001110
B	B	11	12	13	14	102	1011
28	34	40	44	50	55	1111	101000
10	14	16	17	20	22	121	10000
1C	24	28	31	34	40	1001	11100

### Aufgabe 2: Division und Modulo

Wie nennt man die folgenden Operatoren und was sind deren Ergebnisse?

Operator	Name	Ergebnis
/	Division	Ganzzahldivision ohne Rest
%	Modulo	Divisionsrest

Vervollständige die nachfolgende Tabelle:

i	3	10	4	123	123	10	-10
j	2	3	5	10	100	-3	-3
i / j	1	3	0	12	1	-3	3
i % j	1	1	4	3	23	1	-1

## Teil III: Fehlersuche

---

### Aufgabe 1: Einfache Fehler bei int-Variablen

Finde und korrigiere die sieben Fehler in folgendem Nonsens-Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i  j;
6      int m == 2;
7      i = m ** 2;
8      printf( "bitte i eingeben: " );
9      scanf( "%c", & i );
10     printf( "bitte j eingeben: " );
11     scanf( "%d",  j );
12     m = i * j
13     printf( "ergebnis: m=%d\n", & m );
14 }
```

Zeile	Fehler	Erläuterung	Korrektur
5	, fehlt	Bei der Definition müssen die einzelnen Variablen mit Kommas voneinander getrennt werden.	<code>int i, j;</code>
6	<code>==</code> statt <code>=</code>	Die Wertzuweisung geschieht mittels <code>=</code> ; das <code>==</code> wird für Vergleiche benötigt.	<code>int m = 2;</code>
7	<code>**</code> statt <code>*</code>	Die Multiplikation wird mittels <code>*</code> ausgeführt; ein <code>**</code> (für beispielsweise Exponent) gibt es in der Programmiersprache C nicht.	<code>m * 2</code>
9	<code>%c</code> statt <code>%d</code>	Beim Einlesen von <code>int</code> -Werten wird <code>%d</code> als Formatierung benötigt; bei <code>%c</code> wird nur ein einzelnes Zeichen gelesen.	<code>%d</code>
11	<code>&amp;</code> fehlt	Bei der Eingabeanweisung müssen nicht die Werte sondern die Adressen der Variablen übergeben werden, die mittels des Adressoperators <code>&amp;</code> gebildet werden.	<code>&amp; j</code>
12	<code>;</code> fehlt	In der Programmiersprache C wird jede Anweisung mit einem Semikolon beendet.	<code>m = i * j;</code>
13	<code>&amp;</code> zu viel	Bei Ausgabe eines <code>int</code> -Wertes muss die Variable und nicht ihre Adresse übergeben werden.	<code>m</code>

### Programm mit Korrekturen:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i, j, m = 2;
6     i = m * 2;
7     printf( "bitte i eingeben: " ); scanf( "%d", & i );
8     printf( "bitte j eingeben: " ); scanf( "%d", & j );
9     m = i * j;
10    printf( "ergebnis: m=%d\n", m );
11 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Testprogramm für die Grundrechenarten

In dieser Aufgabe sollt ihr ein wenig mit dem Datentyp `int` vertraut werden. Dazu entwickeln wir einen kleinen „Taschenrechner“. Die Programmentwicklung führen wir wie immer gemäß des Software Life Cycle durch.

### 1. Aufgabenstellung

Entwickle ein kleines Testprogramm, das zwei Werte vom Typ `int` nimmt und darauf die fünf Grundrechenarten anwendet. Die fünf Ergebnisse sollten jeweils ausgegeben und mit den erwarteten Ergebnissen verglichen werden. Uns ist es egal, ob ihr den beteiligten Variablen direkt konstante Werte zuweist, oder diese mittels einer entsprechenden Eingabeanweisung einlest.

### 2. Pflichtenheft

Aufgabe : Testprogramm mit den fünf Grundrechenarten  
Eingabe : zwei Zahlen  
Ausgabe : Ergebnis der fünf Grundrechenarten  
Sonderfälle: keine, um das Programm einfach zu halten

### 3. Testdaten

Überlegt euch *mindestens* sechs Datensätze, mit denen Ihr euer Programm testen wollt. Die Testdaten sollten auch ungültige Zahlenwerte enthalten, da es unter Umständen für später interessant ist, schon jetzt das Verhalten des Programms für „unsinnige“ Eingaben zu testen. Ungültige Eingabe sind eines der häufigsten Ursachen für manch merkwürdiges Programmverhalten.

Zahl 1	Zahl 2	+	-	*	/	%
2	1	3	1	2	2	0
5	-2	3	7	-10	-2	1
3	-7	-4	10	-21	0	3
-4	-5	-9	1	20	0	-4
-12	5	-7	-17	-60	-2	-2
8	0	8	8	0	undef.	undef.
300	-300	0	600	-90 000	-1	0
5 000	4 000	9 000	1 000	20 000 000	1	10 000

### 4. Implementierung

Testprogramm für die Grundrechenarten

Variablen: Integer: z1, z2

Ausgabe: Begrüßung

Eingabeaufforderung und Einlesen der Zahl z1

Eingabeaufforderung und Einlesen der Zahl z2

Berechne  $z1 + z2$  und Ausgabe des Ergebnisses

Berechne  $z1 - z2$  und Ausgabe des Ergebnisses

Berechne  $z1 * z2$  und Ausgabe des Ergebnisses

Berechne  $z1 / z2$  und Ausgabe des Ergebnisses

Berechne  $z1 \% z2$  und Ausgabe des Ergebnisses

## 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int z1, z2;
6      printf( "Programm: Grundrechenarten in C\n" );
7      printf( "Bitte erste Zahl z1 eingeben: " );
8      scanf( "%d", & z1 );
9      printf( "Bitte zweite Zahl z2 eingeben: " );
10     scanf( "%d", & z2 );
11     printf("plus      : %6d + %6d = %8d\n",z1,z2,z1+z2);
12     printf("minus     : %6d - %6d = %8d\n",z1,z2,z1-z2);
13     printf("mal       : %6d * %6d = %8d\n",z1,z2,z1*z2);
14     printf("geteilt:  %6d / %6d = %8d\n",z1,z2,z1/z2);
15     printf("modulo    : %6d %% %6d = %8d\n",z1,z2,z1%z2);
16 }
```

Auf der nächsten Seite präsentieren wir noch eine weitere Version, die zusätzliche die „Korrektheit“ der eingegebenen Zahlen überprüft. Für die meisten von euch ist es sicherlich vorteilhaft, wenn ihr euch auch diese Variante anschaut und versucht zu verstehen, was, wie, wo passiert.

```

1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int z1 = 0, z2 = 0, anzahl;
6      printf( "Programm: Grundrechenarten in C\n" );
7      printf( "Eingabe: ganze Zahl z1: " );
8      anzahl = scanf( "%d", & z1 );
9      if ( anzahl > 0 )
10     {
11         printf( "Eingabe: ganze Zahl z2: " );
12         anzahl = scanf( "%d", & z2 );
13         if ( anzahl > 0 )
14         {
15             printf( "### Ergebnisse ###\n" );
16             printf( "%d + %d = %d\n", z1, z2, z1 + z2 );
17             printf( "%d - %d = %d\n", z1, z2, z1 - z2 );
18             printf( "%d * %d = %d\n", z1, z2, z1 * z2 );
19             if ( z2 )          // z2 != 0 waere genauso gut
20             {
21                 printf("%d / %d = %d\n", z1, z2, z1 / z2);
22                 printf("%d %% %d = %d\n",z1, z2, z1 % z2);
23             }
24             else printf("Fehler: Division durch 0\n");
25         }
26         else printf("Fehler: keine Ganzzahl\n");
27     }
28     else printf("Fehler: keine Ganzzahl\n");
29     return 0;
30 }

```

## Aufgabe 2: Ausgabeformate für int-Werte

In Skript und Vorlesung haben wir kurz erwähnt, dass man `int`-Werte mittels `%d` ausgeben kann. Zweck dieser Aufgabe ist es, sich mit der `int`-Formatierung vertiefter zu beschäftigen. Eine erste sehr gute Möglichkeit ist es, sich mittels `man 3 printf` die entsprechende Manual Page durchzulesen. Aufgrund der Länge der Manual Page ist dies sehr mühsame, im Zweifelsfalle aber *unumgänglich*. Hier versuchen wir einen etwas einfacheren Weg, in dem wir mit einigen Elementen ein wenig herumspielen. Also, nur Mut! Schreibe hierfür ein kleines Testprogramm, das verschiedene Werte mittels verschiedener Formatierungen ausgibt. Dieses Testprogramm sollte folgende Eigenschaften haben:

1. Gebe zur besseren Spaltenorientierung hin und wieder eine „Kopfzeile“ aus, die wie folgt aussehen kann: 012345678901234567890123456789.
2. Schließe zur besseren Veranschaulichung die Ausgabe in zwei Marker ein. Ein Beispiel hierfür lautet: `printf( "|%7d|\n", 4711 );`
3. Neben der dezimalen Formatierung `d` sind auch `o`, `x` und `X` möglich. Zu welcher Basis werden die Zahlenwerte *dargestellt* und was ist der Unterschied zwischen `x` und `X`?
4. Das Zusatzzeichen `#` in der Formatierung, beispielsweise `%#x`, veranlasst, dass der Präfix beim Oktal- bzw. Hexadezimalformat automatisch ergänzt wird.
5. Probiere, ob die Längenangabe ein eventuelles Vorzeichen einschließt?
6. Probier, was passiert, wenn die Längenangabe kleiner als die Breite der Zahl ist.
7. Die Längenangabe wie beispielsweise `%7d` kann man auch mit einer Null `%07d` beginnen lassen. Wozu führt dies?
8. Der Längenangabe kann man auch ein Minuszeichen voranstellen. Beispiel: `"%-6d"`, was man mit der führenden Null kombinieren kann. Welchen Effekt hat dies?

**Hinweis zur Fehlererkennung:** Für diejenigen, die ihr Programm mit einer ersten kleinen Fehlererkennung ausstatten wollen, seien folgende Hinweise gegeben: Die Funktion `scanf()` liefert zurück, wie viele ihrer Argumente sie erfolgreich einlesen konnte. Das bedeutet, der Aufruf `ret = scanf( "%d", & i )` weist der Variablen `ret` den Wert 1 zu, wenn sie tatsächlich einen Ganzzahlwert (`int`) einlesen konnte. Sofern in der Eingabezeile etwas anderes, beispielsweise ein nette Anrede wie `Na Du Computer`, vorkommt, erhält die Variable `ret` den Wert 0. Sollte beim Einlesen von einer Datei das Dateende erreicht worden sein oder der Nutzer die Tastenkombination `CTRL-D` gedrückt haben, erhält die Variable `ret` den Wert `EOF`, der in der Regel `-1` ist.

Unser Testprogramm zeigen wir auf der nächsten Seite, damit es durch den Seitenumbruch nicht zerstückelt wird.

## Programm zum Testen diverser Formatierungen:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv )
4  {
5      printf( "dezimal: der Zahlenwert ist 4711\n" );
6      printf( "01234567890123456789\n" );
7      printf( "%d\n", 4711 );                // 4 ziffern
8      printf( "%2d\n", 4711 );                // 4 ziffern
9      printf( "%6d\n", 4711 );                // 2 leerzeichen 4 ziffern
10     printf( "%-6d\n", 4711 );                // dito, aber inksbueendig
11     printf( "%+d\n", 4711 );                // vorzeichen, 4 ziffern
12
13     printf( "dezimal: der Zahlenwert ist -4711\n" );
14     printf( "01234567890123456789\n" );
15     printf( "%6d\n", -4711 );                // blank, '-', 4 ziffern
16     printf( "%2d\n", -4711 );                // vorzeichen, 4 ziffern
17
18     printf( "\nhex: der Zahlenwert ist 1234\n" );
19     printf( "01234567890123456789\n" );
20     printf( "%x\n", 1234 );                  // hex, kleinbuchstaben
21     printf( "%X\n", 1234 );                  // hex, grossbuchstaben
22     printf( "%#x\n", 1234 );                 // hex, mit 0x
23     printf( "%#X\n", 1234 );                 // hex, mit 0X
24     printf( "%#-8x\n", 1234 );                // hex, 0x1234, 4 blanks
25     printf( "%#08x\n", 1234 );                // hex, mit 0x, nullen
26
27     printf( "\noctal: der Zahlenwert ist 4711\n" );
28     printf( "01234567890123456789\n" );
29     printf( "%o\n", 4711 );                  // 5 ziffern, ohne 0
30     printf( "%#o\n", 4711 );                 // 5 ziffern, mit 0
31
32     printf( "\ndynamisch dez.: der Zahlenwert ist 4711\n");
33     printf( "01234567890123456789\n" );
34     printf( "%*d\n", 8, 4711 );              // acht ziffern
35     return 0;
36 }
```



# Übungspaket 9

## Logische Ausdrücke

---

### Übungsziele:

1. Umgang mit logischen Vergleichs- und Verknüpfungsoperatoren
2. Bilden einfacher und komplexer logischer Ausdrücke

### Skript:

Kapitel: 22

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Logische Ausdrücke bestehen aus Vergleichsoperatoren wie „größer“ oder „kleiner“ sowie aus logischen Verknüpfungen wie „und“, „oder“ bzw. „nicht“. Logische Ausdrücke benötigen wir vor allem bei Fallunterscheidungen sowie Schleifen und üben sie deshalb hier gesondert ein.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Wahr und Falsch

Wie werden in der Programmiersprache C die beiden logischen Werte „wahr“ und „falsch“ ausgedrückt?

wahr :

falsch:

## Aufgabe 2: Präzedenzregeln

Erkläre *kurz* mit eigenen Worten, was man beim Programmieren als Präzedenzregeln bezeichnet. Was hat dies mit der Regel „Punktrechnung geht vor Strichrechnung“ zu tun, die wir aus dem Mathematikunterricht kennen?

Präzedenzregeln (auch Vorrangregeln genannt) geben an, welche Operationen in welcher Reihenfolge ausgeführt werden. In der Mathematik ist es ähnlich. Hier gibt es beispielsweise die Regel „Punktrechnung geht vor Strichrechnung“. Diese Regeln stellen sicher, dass derartige Ausdrücke von allen in gleicherweise ausgerechnet werden.

## Aufgabe 3: Klammersetzung

Welche Klammern sind für das Einklammern von arithmetischen und logischen Ausdrücken in der Programmiersprache C erlaubt?

() :     [] :     {}:

## Teil II: Quiz

---

### Aufgabe 1: Klammersetzung

Bei gemischten Ausdrücken ist immer die Frage, ob man Klammern setzen muss und wenn ja, wo? Hier hilft die Präzedenztabelle (siehe Anhang im Skript oder den gesammelten Übungspaketen) weiter, nach der der Compiler die Ausdrücke auswertet. Wo sind in den folgenden Ausdrücken die Klammern, wie sie sich der Compiler denkt. In den Ausdrücken sind `i` und `j` immer Variablen vom Typ `int`.

Ohne Klammern	mit gedachten Klammern
<code>i &gt; 1 &amp;&amp; j &gt; 2</code>	<code>(i &gt; 1) &amp;&amp; (j &gt; 2)</code>
<code>i &lt; j    j == 1</code>	<code>(i &lt; j)    (j == 1)</code>
<code>i &lt; j    j == 1 &amp;&amp; j != 4</code>	<code>(i &lt; j)    ((j == 1) &amp;&amp; (j != 4))</code>
<code>i + 1 &lt; j    j + 2 &lt; i</code>	<code>((i + 1) &lt; j)    ((j + 2) &lt; i)</code>

### Aufgabe 2: Vereinfachung logischer Ausdrücke

Manchmal kann man logische Ausdrücke vereinfachen. Im Folgenden steht `a` immer für einen (logischen) Ausdruck, der die Werte „wahr“ und „falsch“ annehmen kann. Überlege, wie sich die folgenden Ausdrücke vereinfachen lassen:

Ausdruck	Vereinfachung
<code>a und wahr</code>	<code>a</code>
<code>a oder wahr</code>	<code>wahr</code>
<code>a und falsch</code>	<code>falsch</code>
<code>a oder falsch</code>	<code>a</code>

## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler in logischen Ausdrücken

Im folgenden Programm haben sich in den logischen Ausdrücken wieder ein paar kleine Fehler eingeschlichen. Die nachfolgende Ausgabeanweisung (`printf()`) gibt immer an, für welche Fälle der Ausdruck ein logisches „wahr“ liefern soll. Finde und korrigiere die Fehler.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, k;
6      /* einlesen von Werten fuer i, j und k */
7      if i == 1 && j == 1 )
8          printf( "i und j haben beide den wert 1\n" );
9      if ( i == 1 && j == 1 || k > 0 )
10         printf( "i ist 1 und gleichzeitig j gleich 1 oder k
              positiv\n" );
11      if ( i => 1 )
12          printf( "i ist groesser oder gleich 1\n" );
13      if ( i = 1 )
14          printf( "i hat den wert 1\n" );
15      if ( i == 1 &&& j == 1 )
16          printf( "i und j haben beide den wert 1\n" );
17      if ( ! (i < 1) && j > -1 )
18          printf( "i und j sind beide groesser als 0\n" );
19  }
```

Zeile	Fehler	Erläuterung	Korrektur
7	fehlende (	Die if-Anweisung benötigt ein Paar ()-Klammern.	if ( ... )
9	fehlende )	Der und-Operator && bindet stärker als der oder-Operator   . Gemäß Ausgabertext sollte gegenteilig ausgewertet werden.	& ( ...    ... )
11	Schreibweise =>	Die Zeichen sind vertauscht	>=
13	= statt ==	Im Programm steht der Zuweisungsoperator = anstelle des Vergleichsoperators ==	==
15	&&&	Falsche Schreibweise des und-Operators	&&
17	j > -1	Auch der Fall j == 0 ergibt fälschlicherweise wahr. Hinweis: der erste Ausdruck ist korrekt.	j > 0

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, k;
6      /* einlesen von Werten fuer i, j und k */
7      if (i == 1 && j == 1 )
8          printf( "i und j haben beide den wert 1\n" );
9      if ( i == 1 && (j == 1 || k > 0) )
10         printf( "i ist 1 und gleichzeitig j gleich 1 oder k
           positiv\n" );
11     if ( i >= 1 )
12         printf( "i ist groesser oder gleich 1\n" );
13     if ( i == 1 )
14         printf( "i hat den wert 1\n" );
15     if ( i == 1 && j == 1 )
16         printf( "i und j haben beide den wert 1\n" );
17     if ( ! (i < 1) && j > 0 )
18         printf( "i und j sind beide groesser als 0\n" );
19 }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Werte logischer Ausdrücke

Welche Bedeutung und Werte haben bzw. liefern die folgenden logischen Ausdrücke? In allen Fällen sind `i = 1`, `j = 2` und `n = 0` Variablen vom Typ `int` und haben in allen Fällen die angegebenen drei Werte. Überlege dir erst das Ergebnis und überprüfe es erst anschließend anhand eines kleinen eigenen Programms. Beispiel für die erste Aufgabe:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i = 1, j = 2, n = 0;
6     printf( "Ergebnis: %d\n", i > 2 );
7 }
```

Ausdruck	Ergebnis	Bedeutung
<code>i &gt; 2</code> .....	0/„falsch“	i ist größer als 2
<code>i &lt; 2</code> .....	1/„wahr“	i ist kleiner als 2
<code>i &lt; 2 &amp;&amp; n == 0</code> .....	1/„wahr“	i ist kleiner als 2 und n ist null
<code>i</code> .....	1/„wahr“	i ist ungleich null („wahr“)
<code>! i</code> .....	0/„falsch“	i hat den Wert null („falsch“)
<code>! n</code> .....	1/„wahr“	n hat <i>nicht</i> den Wert „wahr“
<code>i == 1 &amp;&amp; j</code> .....	1/„wahr“	i hat den Wert 1 und j ist „wahr“
<code>i &gt; 2    n == 0</code> .....	1/„wahr“	i ist größer als 2 oder n hat den Wert 0
<code>i &gt; 2    j == 2</code> .....	1/„wahr“	i ist größer als 2 oder j hat den Wert 2
<code>i &amp;&amp; ! n</code> .....	1/„wahr“	i ist „wahr“ und n ist „falsch“
<code>i == 2 &amp;&amp; (j == 1    n == 1)</code>	0/„falsch“	i hat den Wert 2 und entweder ist j gleich 1 oder n gleich 1
<code>(i == 2 &amp;&amp; j == 1)    n == 1</code>	0/„falsch“	i ist gleich 2 und j hat den Wert 1 oder n hat den Wert 1
<code>i &amp;&amp; j &amp;&amp; ! n</code> .....	1/„wahr“	i und j sind „wahr“ und n ist „falsch“

# Übungspaket 10

## Fallunterscheidungen

---

### Übungsziele:

1. Umgang mit der einfachen Fallunterscheidung,
2. sowie mehrfachen Fallunterscheidung und
3. problemangepasster Auswahl

### Skript:

Kapitel: 24 und 25

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

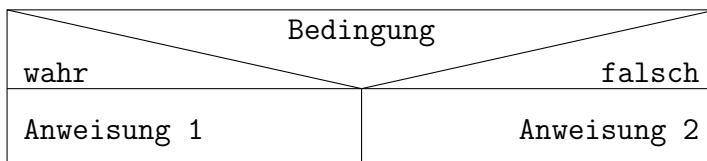
Die meisten C-Programme können nicht einfach von oben nach unten abgearbeitet werden. Zwischendurch muss der Programmablauf immer mal wieder in die eine oder andere Richtung geändert werden, wie wir es schon bei den ersten Programmieraufgaben gesehen haben. In diesem Übungspaket gehen wir nun genauer auf die einfache sowie mehrfache Fallunterscheidung ein.

# Teil I: Stoffwiederholung

## Aufgabe 1: Einfache Fallunterscheidung

Erkläre die einfache Fallunterscheidung anhand folgender Punkte mit eigenen Worten:

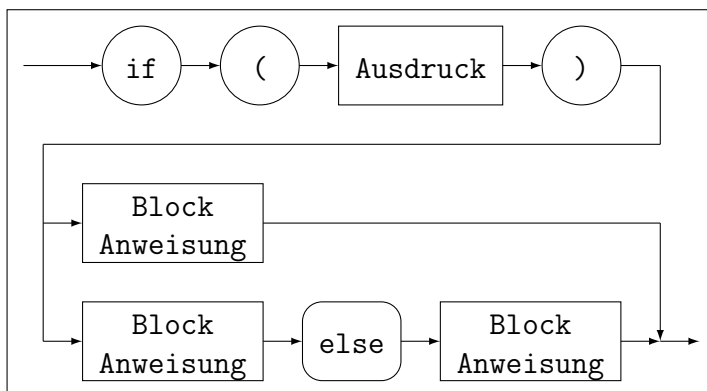
Struktogramm:



Schrittweise Verfeinerung:

```
wenn Bedingung
dann Anweisung 1
sonst Anweisung 2
```

C Syntax:



Zwei konkrete Beispiele:

Beispiel I:

```
if ( i >= 0 )
    printf( "positiv\n" );
else printf( "negativ\n");
```

Beispiel II:

```
if ( j == 3 )
    printf( "j = 3\n" );
else printf( "j != 3\n");
```

## Aufgabe 2: Einfache Fallunterscheidung geschachtelt

Erkläre an *einem Beispiel* eine geschachtelte Fallunterscheidung (eine innere Fallunterscheidung in einer äußeren)

Schrittweise Verfeinerung:

```
wenn i ≥ 0
dann wenn ≥ j 0
    dann ausgabe fall 1
    sonst ausgabe fall 2
sonst wenn j ≥ 0
    dann ausgabe fall 3
    sonst ausgabe fall 4
```

C-Beispiel:

```
if ( i >= 0 )
    if ( j >= 0 )
        printf( "Fall 1\n" );
    else printf( "Fall 2\n" );
else if ( j >= 0 )
    printf( "Fall 3\n" );
else printf( "Fall 4\n" );
```



## Aufgabe 3: Mehrfache Fallunterscheidung

Erkläre mit eigenen Worten die mehrfache Fallunterscheidung anhand folgender Punkte:

### Schrittweise Verfeinerung:

auswahl: Ausdruck oder Variable	
wenn Wert 1:	Anweisung 1
wenn Wert 2:	Anweisung 2
wenn Wert 3:	Anweisung 3
sonst	: Anweisung 4

### Struktogramm:

Ausdruck oder Variable			
Wert 1	Wert 2	Wert 3	sonst
Anweisung 1	Anweisung 2	Anweisung 3	Anweisung 4

### Ein konkretes Beispiel:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i;
6      printf("Bitte Zahl 0..20 eingeben: "); scanf("%d", &i);
7      switch( i )
8      {
9          case 2: printf( "Primzahl 2\n" ); break;
10         case 3: printf( "Primzahl 3\n" ); break;
11         case 5: printf( "Primzahl 5\n" ); break;
12         case 7: printf( "Primzahl 7\n" ); break;
13         case 11: printf( "Primzahl 11\n" ); break;
14         case 13: printf( "Primzahl 13\n" ); break;
15         case 17: printf( "Primzahl 17\n" ); break;
16         case 19: printf( "Primzahl 19\n" ); break;
17         default: if ( i < 0 || i > 20 )
18                 printf( "falsche Eingabe\n" );
19                 else printf( "keine Primzahl\n" );
20                 break;
21     }
22 }
```

## Teil II: Quiz

---

### Aufgabe 1: Ablauf: einfache Fallunterscheidung

Gegeben sei folgendes kleines Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i;
6      scanf( "%d", & i );
7      if ( i > 1 )
8          printf( "i= %d\n", i );
9      if ( i > 1 && i < 6 )
10         printf( "i= %d\n", i );
11     if ( i > 1 || i == -4 )
12         printf( "i= %d\n", i );
13     if ( i < 0 )
14         if ( i > -10 )
15             printf( "i= %d\n", i );
16         else printf( "i= %d\n", i );
17 }
```

Notiere, welche `printf()`-Zeilen für welche Werte der Variablen `i` ausgeführt werden:

i	Ausgeführte Zeilen				
	8	10	12	15	16
-20	nein	nein	nein	nein	ja
-5	nein	nein	nein	ja	nein
-4	nein	nein	ja	ja	nein
0	nein	nein	nein	nein	nein
3	ja	ja	ja	nein	nein
8	ja	nein	ja	nein	nein

## Mehrfache Fallunterscheidung:

Gegeben sei folgendes kleines Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i;
6      scanf( "%d", & i );
7      switch( i )
8      {
9          case 0: printf( "i= %d\n", i ); break;
10         case 2: printf( "i= %d\n", i );
11         case 4: printf( "i= %d\n", i );
12         case 6: printf( "i= %d\n", i ); break;
13         case -1: printf( "i= %d\n", i );
14         case -3: printf( "i= %d\n", i ); break;
15         default: printf( "i= %d\n", i ); break;
16     }
17 }
```

Notiere, welche `printf()`-Zeilen für welche Werte der Variablen `i` ausgeführt werden:

i	Ausgeführte Zeilen						
	9	10	11	12	13	14	15
-4	nein	nein	nein	nein	nein	nein	ja
-3	nein	nein	nein	nein	nein	ja	nein
-2	nein	nein	nein	nein	nein	nein	ja
-1	nein	nein	nein	nein	ja	ja	nein
0	ja	nein	nein	nein	nein	nein	nein
1	nein	nein	nein	nein	nein	nein	ja
2	nein	ja	ja	ja	nein	nein	nein
3	nein	nein	nein	nein	nein	nein	ja
4	nein	nein	ja	ja	nein	nein	nein
5	nein	nein	nein	nein	nein	nein	ja
6	nein	nein	nein	ja	nein	nein	nein
7	nein	nein	nein	nein	nein	nein	ja

## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler in Fallunterscheidungen

In den folgenden Programmen sind DR. DREAM ein paar kleine Fehler unterlaufen. Finde diese und korrigiere sie direkt im Quelltext.

**Einfache Fallunterscheidung:**

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, k;
6      scanf( "%d", i ); scanf( "%z", & j );
7      scanf( "%d", & k )
8      if ( i > j )
9          if ( i == k )
10             printf( "i= %d\n", i );
11             else printf( "j= %d k= %d\n", j, k ;
12     else {
13         if ( k == -1 || -3 )
14             printf( "so doof hier\n" );
15             printf( "kaum zu glauben\n" );
16             else printf( "na servus\n" );
17     }
```

Zeile	Fehler	Erläuterung	Korrektur
6	fehlendes &	Bei der <code>scanf()</code> -Anweisung müssen die <i>Adressen</i> & i der Variablen übergeben werden.	
6	z statt d	Beim Einlesen von Ganzzahlwerten ist die Formatangabe <code>%d</code> zu verwenden.	<code>"%d"</code>
7	; fehlt	In C muss jede Anweisung mit einem Semikolon abgeschlossen werden.	<code>scanf();</code>
11	) Klammer fehlt	Die Parameter einer Anweisung (Funktion) müssen in runden Klammern übergeben werden.	<code>k );</code>
13	Variable k fehlt	Ein logischer Ausdruck darf nicht wie in der Umgangssprache üblich einfach abgekürzt werden.	<code>k == -1    k == -3</code>
14/15	{ } Klammern fehlen	Bei der if/else-Kontrollstruktur <i>müssen</i> mehrere Anweisung geklammert werden.	<code>{ ... }</code>

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, k;
6      scanf( "%d", & i ); scanf( "%d", & j );
7      scanf( "%d", & k );
8      if ( i > j )
9          if ( i == k )
10             printf( "i= %d\n", i );
11             else printf( "j= %d k= %d\n", j, k );
12     else {
13         if ( k == -1 || k == -3 )
14             {
15                 printf( "so doof hier\n" );
16                 printf( "kaum zu glauben\n" );
17             }
18         else printf( "na servus\n" );
19     }
20 }
```

## Mehrfache Fallunterscheidung:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6     scanf( "%d", & i );
7     switch( i )
8     {
9         case 0: printf( "huhu " );
10        case 1: printf( "doofi\n" ); break;
11        case 2  printf( "wer ist hier ein doedel?\n" );
12        csae 3: printf( "jetzt reicht's aber\n" );
13        DEFAULT: printf( "jetzt ist endgueltig schluss\n" );
14    }
15 }
```

Zeile	Fehler	Erläuterung	Korrektur
11	: fehlt	Nach jedem Wert einer <b>case</b> -Auswahl muss ein Doppelpunkt stehen.	<b>case 2:</b>
12	csae statt case	Das ist ein einfacher Tippfehler.	<b>case 3:</b>
13	DEFAULT falsch geschrieben	Das Label <b>default:</b> muss in Kleinbuchstaben geschrieben werden.	<b>default:</b>

## Programm mit Korrekturen:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6     scanf( "%d", & i );
7     switch( i )
8     {
9         case 0: printf( "huhu " );
10        case 1: printf( "doofi\n" ); break;
11        case 2: printf( "wer ist hier ein doedel?\n" );
12        case 3: printf( "jetzt reicht's aber\n" );
13        default: printf( "jetzt ist endgueltig schluss\n" );
14    }
15 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Korrelation zweier Variablen

In dieser Aufgabe verwenden wir die *Fallunterscheidung*, um die Korrelation zweier Eingangsgrößen zu ermitteln. Die Programmentwicklung erfolgt natürlich wieder gemäß der Struktur des Software Life Cycle.

### 1. Aufgabenstellung

Gegeben seien zwei Variablen `i` und `j` vom Typ `int`. Diese beiden Variablen haben beliebige Werte. Ferner haben wir eine Ergebnisvariable `ergebnis`, die ebenfalls vom Typ `int` ist. Schreibe ein Programm, das die beiden Variablen wie folgt korreliert:

1. Ergebnis `1`, wenn beide Variablen das gleiche Vorzeichen haben
2. Ergebnis `-1`, wenn beide Variablen ein unterschiedliches Vorzeichen haben
3. Ergebnis `0`, wenn mindestens eine der beiden Variablen den Wert null hat.

### 2. Pflichtenheft

Aufgabe	: Korrelation zweier Variablen <code>i</code> und <code>j</code> bestimmen; Formeln laut Aufgabenstellung
Eingabe	: Einlesen der beiden Variablen <code>i</code> und <code>j</code>
Ausgabe	: Ergebnis der Korrelationsberechnung <code>-1</code> , <code>0</code> oder <code>1</code> .
Sonderfälle	: keine.

### 3. Testdaten

<code>i</code>	<code>j</code>	<code>ergebnis</code>
1	1	1
10	3	1
1	-1	-1
-3	1	-1

<code>i</code>	<code>j</code>	<code>ergebnis</code>
73	0	0
0	-14	0
0	0	0
-55	-76	1

### 4. Implementierung (nur für die Bestimmung der Korrelation)

```
Bestimmung der Korrelation
wenn i = 0 oder j = 0
dann setze ergebnis = 0
sonst wenn (i > 0 und j > 0) oder (i < 0 und j < 0)
dann setze ergebnis = 1
sonst setze ergebnis = -1
```

## 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, ergebnis;
6
7      // einlesen der beiden variablen
8      printf( "Bitte die erste Zahl eingeben: " );
9      scanf( "%d", & i );
10     printf( "Bitte die zweite Zahl eingeben: " );
11     scanf( "%d", & j );
12
13     // bestimmen der korrelation
14     if ( i == 0 || j == 0 )
15         ergebnis = 0;
16     else if ((i > 0 && j > 0) || (i < 0 && j < 0))
17         ergebnis = 1;
18     else ergebnis = -1;
19
20     // ausgabe des ergebnisses
21     printf( "i=%d j=%d ", i, j );
22     printf( "Korrelation=%d\n", ergebnis );
23 }
```

## Aufgabe 2: Code-Umwandlung

### 1. Aufgabenstellung

Schreibe ein Programm, das für die Anpassung einer Steuerelektronik (beispielsweise eines Elektroautos) ein paar Konvertierungen vornimmt. Hierzu müssen einige ausgewählte Eingabewerte in neue Werte umgewandelt werden; alle anderen Werte sollen unverändert bleiben. Die Umwandlungstabelle sieht wie folgt aus:

alter Wert	-1	0	4	5	12	13	14	17	34	35
neuer Wert	12	-3	8	4	11	-2	-3	-4	24	1

### 2. Pflichtenheft

Aufgabe : Codeumwandlung gemäß Aufgabenstellung  
Eingabe : Einlesen eines alten Codes  
Ausgabe : Ausgabe des neuen Codes



### 3. Implementierung

#### Codeumwandlung

```
Variablen: Integer: eingabewert, ausgabewert
Eingabeaufforderung und einlesen des alten Wertes
auswahl: eingabewert
    wenn -1: setze ausgabewert = 12
    wenn 0: setze ausgabewert = -3
    .... ..: .....
    wenn 35: setze ausgabewert = 1
    sonst : setze ausgabewert = eingabewert
ausgabe text "Neuer Wert" ausgabewert
```

### 4. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int in, out;
6
7      // einlesen der beiden variablen
8      printf( "Bitte Wert eingeben: " );
9      scanf( "%d", & in );
10
11     // codeumwandlung
12     switch( in )
13     {
14         case -1: out = 12; break;
15         case 0: out = -3; break;
16         case 4: out = 8; break;
17         case 5: out = 4; break;
18         case 12: out = 11; break;
19         case 13: out = -2; break;
20         case 14: out = -3; break;
21         case 17: out = -4; break;
22         case 34: out = 24; break;
23         case 35: out = 1; break;
24         default: out = in;
25     }
26     // ausgabe des ergebnisses
27     printf( "Neuer Wert: %d\n", out );
28 }
```

# Übungspaket 11

## Schleifen

---

### Übungsziele:

1. Umgang mit den Schleifen in C
2. Wahl des richtigen Schleifentyps
3. Umwandlung der Schleifen ineinander

### Skript:

Kapitel: 26 bis 28

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Gegenstand dieses Übungspakets ist das Einüben der Schleifenkonstrukte, die in der Vorlesung erläutert wurden. Dabei verfolgen die Aufgaben zwei Ziele: erstens, das Automatisieren und routinierte Anwenden der Schleifen-Befehle, und zweitens, das Erarbeiten grundlegender Entwurfs- bzw. Programmiermuster.

# Teil I: Stoffwiederholung

## Aufgabe 1: Syntax der Schleifenbefehle

Wie lauten die drei unterschiedlichen Schleifenkonstrukte/-befehle, wie lautet deren Syntax und wie sehen ihre zugehörigen Struktogramme aus?

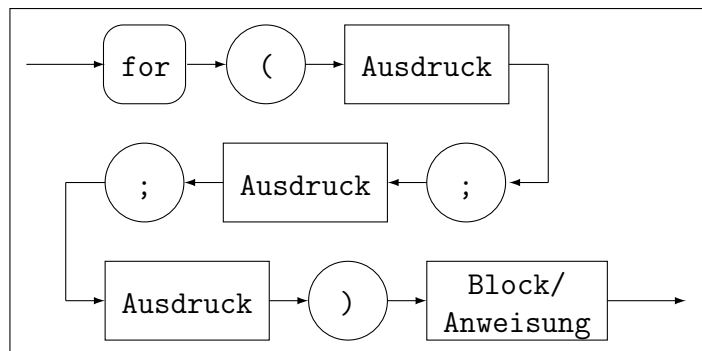
**Befehl:**

for-Schleife

**Struktogramm:**

```
für x = a bis e  
    schrittweite s  
Anweisung
```

**Syntaxdiagramm:**



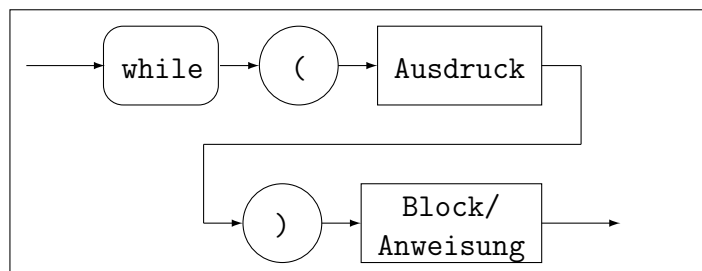
**Befehl:**

while-Schleife

**Struktogramm:**

```
solange Bedingung (erfüllt)  
Anweisung
```

**Syntaxdiagramm:**

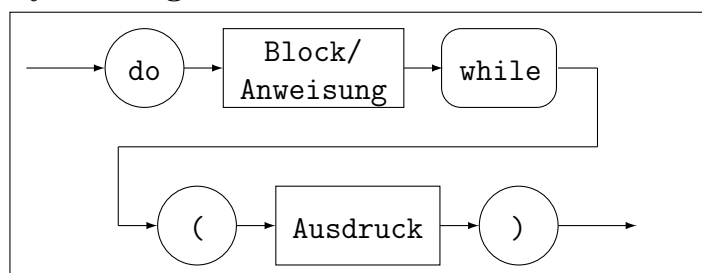


do-while-Schleife

**Struktogramm:**

```
Anweisung  
solange Bedingung (erfüllt)
```

**Syntaxdiagramm:**



**Wichtiger Hinweis:** Aus Gründen der Übersichtlichkeit ist in obigen Syntaxdiagrammen nicht explizit kenntlich gemacht, dass ein Ausdruck auch leer sein kann oder aus einer Liste von Ausdrücken bestehen kann, die durch Kommas voneinander getrennt sind.

## Aufgabe 2: Detailfragen zu Schleifen

1. Wann müssen die Anweisungen des Schleifenblocks, also diejenigen Anweisungen, die wiederholt werden sollen, in geschweifte Klammern `{}` gesetzt werden? Wann können diese Klammern entfallen?

Prinzipiell muss der Schleifenrumpf aus genau einer Anweisung bestehen. Hierzu zählen auch die Leeraanweisungen, die nur aus einem Semikolon bestehen. Bei mehr als einer Anweisung müssen diese in geschweifte Klammern `{}` gesetzt werden.

2. Wie viele Anweisungen müssen mindestens im Anweisungsblock stehen?

Mindestens: `eine Anweisung`

**Hinweis::** Auch die Leeraanweisung ist eine Anweisung. Wie wird diese Leeraanweisung in C geschrieben? Leeraanweisung: `besteht nur aus einem Semikolon ;`

3. Wo steht in den Schleifenbefehlen jeweils die Bedingung? Unterstreiche jeweils in der ersten Teilaufgabe.

Hinter `while` bzw. der zweite Ausdruck in der `for`-Schleife

4. Bei welchen Werten der Schleifenbedingung wird die Schleife beendet?

Ende wenn: `die Bedingung nicht mehr erfüllt („false“) ist bzw. den Wert 0 liefert.`

5. Bei welchen Werten der Schleifenbedingung wird die Schleife *nicht* abgebrochen?

Weiter wenn: `die Bedingung erfüllt („true“) ist bzw. den Wert ungleich 0 liefert.`

6. Erläutere kurz, wann welche Teile der `for`-Schleife ausgeführt werden?

1. Teil: `vor` Eintritt in die Schleife

2. Teil: `am Anfang jedes` Schleifendurchlaufs

3. Teil: `am Ende jedes` Schleifendurchlaufs

7. Wie viele Wertzuweisungen dürfen im dritten Teil der `for`-Schleife stehen?

Beliebig viele, durch Kommas getrennte Ausdrücke

8. Wandle die Schleife `for( exp1 ; exp2; exp3 ) Anweisung;` in eine entsprechende `while` bzw. `do-while`-Schleife um:

**while-Schleife:**

```
exp1;
while( exp2 )
{
    Anweisung; exp3;
}
```

**do-while-Schleife:**

```
exp1;
do {
    Anweisung;
    if ( exp2 ) exp3;
} while( exp2 );
```

## Teil II: Quiz

---

### Aufgabe 1: Formatierte Ausgabe

Gegeben sei folgendes Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, end;
6      scanf( "%d", & end );
7      for( i = 0; i < end; i = i + 1 )
8      {
9          for( j = 0; j < end * 2; j = j + 1 )
10             printf( "*" );
11             printf( "\n" );
12     }
13     return 0;
14 }
```

In dieser Quizaufgabe sollt ihr herausfinden, was obiges Programmstück eigentlich macht; ist also die übliche Aufgabe eurer Betreuer ;-). Führe für obiges Programm für die Eingabe 1 eine Handsimulation durch:

Zeile	Variablen	Aktion	Resultat/Effekt
5	.....	Definition int i,j,end	Anlegen, aber <i>keine</i> Initialisierung
6	i=? j=? end=?	scanf( "%d", & end )	Eingabe 1: end = 1
7	i=? j=? end=1	for-init: i = 0	i = 0
7	i=0 j=? end=1	for-test: i < end	wahr
9	i=0 j=? end=1	for-init: j = 0	j = 0
9	i=0 j=0 end=1	for-test: j < end*2	wahr
10	.....	printf( "*" )	es wird ein * ausgegeben
9	.....	for-loop: j = j + 1	j = 1
9	i=0 j=1 end=1	for-test: j < end*2	wahr
10	.....	printf( "*" )	es wird ein * ausgegeben
9	.....	for-loop: j = j + 1	j = 2
9	i=0 j=2 end=1	for-test: j < end*2	falsch
11	.....	printf( "\n" )	es wird ein Zeilenumbruch ausgegeben
7	.....	for-loop: i = i + 1	i = 1
7	i=1 j=2 end=1	for-test: i < end	falsch
13	.....	return 0	Programmende

Im Folgenden werden wir schrittweise, von innen nach außen, versuchen, uns die Funktionsweise des Programms zu erschließen. Beantworte dazu der Reihe nach folgende Fragen. Beschreibe zunächst, was die innere Schleife macht, die sich in den Zeilen 9-11 befindet.

In der inneren Schleife (Zeile 9) wird die „Laufvariable“ `j` von 0 schrittweise um 1 erhöht, bis sie den Wert von `end * 2` erreicht (solange sie den Wert `end * 2` unterschreitet). Entsprechend werden insgesamt `end * 2` einzelne Schleifendurchläufe ausgeführt. In jedem einzelnen Schleifendurchlauf (Zeile 10) wird genau ein `*` (Sternchen) ausgegeben. Insgesamt werden also `end * 2` Sternchen ausgegeben.

Beschreibe kurz, was die äußere Schleife macht, die sich von Zeile 7-12 erstreckt.

Die „Laufvariable“ `i` der äußeren `for`-Schleife wird von 0 beginnend schrittweise um 1 erhöht, so lange ihr Wert kleiner als `end` ist. Dies bedeutet, dass die äußere Schleife genau `end` Mal durchlaufen wird. Der Schleifenrumpf besteht aus zwei Teilen. Im ersten Teil werden mittels der inneren Schleife `end*2` Sternchen ausgegeben. Im zweiten Teil des Schleifenrumpfes wird diese Ausgabe durch einen Zeilenumbruch `\n` abgeschlossen.

Fasse die beiden obigen Erklärungen in einem kurzen, prägnanten Satz zusammen.

Durch die beiden ineinander verschachtelten Schleifen gibt das Programm `end` Zeilen aus, in denen sich jeweils `2*end` Sternchen befinden.

Warum wurden die `for()`-Schleifen gewählt?

Die Zahl der jeweiligen Schleifendurchläufe steht bereits *vor* Eintritt in die Schleife fest. Um dies anderen Lesern und Programmierern zu verdeutlichen, wurde die `for`-Schleife gewählt. In C könnte man auch eine der beiden anderen Schleifen nehmen.

## Aufgabe 2: Eingabeverarbeitung

Gegeben sei folgendes Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int in, cnt = 0;
6      scanf( "%d", & in );
7      while( in != 10 && in > 0 )
8      {
9          cnt = cnt + 1;
10         scanf( "%d", & in );
11     }
12     printf( "cnt= %d\n", cnt );
13     return 0;
14 }
```

Eingabe: 20 1 13 9 -1 2 10

Zeile	Variablen	Aktion	Resultat/Effekt
5	.....	Definition <code>int in, cnt=0</code>	Anlegen und Teilinitialisierung
6	<code>in= ? cnt=0</code>	<code>scanf( "%d", &amp; in )</code>	Eingabe 20: <code>in = 20</code>
7	<code>in=20 cnt=0</code>	<code>in != 10 &amp;&amp; in &gt; 0</code>	wahr
9	.....	<code>cnt = cnt + 1</code>	<code>cnt = 1</code>
10	<code>in=20 cnt=1</code>	<code>scanf( "%d", &amp; in )</code>	Eingabe 1: <code>in = 1</code>
7	<code>in= 1 cnt=1</code>	<code>in != 10 &amp;&amp; in &gt; 0</code>	wahr
9	.....	<code>cnt = cnt + 1</code>	<code>cnt = 2</code>
10	<code>in= 1 cnt=2</code>	<code>scanf( "%d", &amp; in )</code>	Eingabe 13: <code>in = 13</code>
7	<code>in=13 cnt=2</code>	<code>in != 10 &amp;&amp; in &gt; 0</code>	wahr
9	.....	<code>cnt = cnt + 1</code>	<code>cnt = 3</code>
10	<code>in=13 cnt=3</code>	<code>scanf( "%d", &amp; in )</code>	Eingabe 9: <code>in = 9</code>
7	<code>in= 9 cnt=3</code>	<code>in != 10 &amp;&amp; in &gt; 0</code>	wahr
9	.....	<code>cnt = cnt + 1</code>	<code>cnt = 4</code>
10	<code>in= 9 cnt=4</code>	<code>scanf( "%d", &amp; in )</code>	Eingabe -1: <code>in = -1</code>
7	<code>in=-1 cnt=4</code>	<code>in != 10 &amp;&amp; in &gt; 0</code>	falsch
12	.....	<code>printf( ... )</code>	Ausgabe: <code>cnt= 4</code> nebst Zeilenumbruch
13	.....	<code>return 0</code>	Programmende

Beschreibe zunächst, was die Schleife macht, die sich von Zeile 7 bis Zeile 11 erstreckt.

Diese Schleife zählt bei jedem einzelnen Durchlauf die Variable `cnt` um 1 hoch und liest anschließend die nächste Zahl ein. Die Schleife wird solange wiederholt, wie die eingelesene Zahl `in` größer als 0 und ungleich 10 ist.

Beschreibe nun, was das ganze Programm macht.

Zunächst wird die „Zählvariable“ `cnt` mit dem Wert 0 initialisiert. Anschließend wird die erste Zahl `in` eingelesen. Im Anschluss daran wird oben beschriebene Schleife ausgeführt, bis die eingelesene Zahl entweder 10 oder kleiner/gleich 0 ist. Am Ende wird die Zahl der eingelesenen „Nutzahlen“ ausgegeben.

Fasse die beiden obigen Erklärungen in einem kurzen, prägnanten Satz zusammen.

Das Programm zählt, wie viele positive Zahlen ungleich 10 sich im Eingabestrom befinden und gibt deren Anzahl aus.

Warum wurde die `while()`-Schleife gewählt?

Die Zahl der sich im Eingabestrom befindlichen „nutzbaren“ Zahlen (also größer als 0 und ungleich 10) ist nicht bei Programmstart bekannt, sondern ergibt sich erst während der Laufzeit des Programms. Um dies anderen Lesern und Programmierern zu verdeutlichen, wurde die `while`-Schleife gewählt. In der Programmiersprache C könnte man auch eine der beiden anderen Schleifen nehmen.

## Teil III: Fehlersuche

---

### Aufgabe 1: Erstellen einer Ausgabezeile

Das folgende Programm soll eine Zeile ausgeben, in der genau zehn Sternchen zu sehen sind. Finde die Fehler und korrigiere direkt im Programm:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6     for( i = 2, i <= 10; i++ );
7     printf( " *" );
8     printf( "\n" );
9 }
```

Zeile	Fehler	Erläuterung	Korrektur
7	Leerzeichen zu viel	Hier wird mehr als ein Sternchen ausgegeben, das Leerzeichen ist zu viel.	"*"
6	, statt ;	Die drei Teile innerhalb der for-Anweisung müssen mittels Semikolon getrennt werden.	for( ... ; ... ; ... )
6	; am Ende	Meist ist das Semikolon nach der runden Klammer der for-Anweisung falsch, da dadurch die Leeraanweisung entsprechend ausgeführt wird.	kein Semikolon am Ende
6	falsche Schleifenbedingungen	Der Schleifenrumpf wird nicht zehn mal wiederholt. Dies kann auf mehrere Arten korrigiert werden.	for( i = 0; i < 10; i++ )

**Programm mit Korrekturen:**

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6     for( i = 0; i < 10; i++ )
7         printf( "*" );
8     printf( "\n" );
9 }
```



Das folgende Programm soll solange Zahlen einlesen, wie diese ungleich 0 sind. Dabei soll gezählt und ausgegeben werden, wie viele negative Zahlen im Eingabestrom vorkamen:

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int n, cnt;
6      scanf( "%d";    n );
7      while( n = 0 )
8      {
9          if ( n <> 0 )
10             cnt + 1;
11             scanf(  %d , & n );
12         }
13         printf(  %d negative Zahlen gefunden\n", cnt );
14     }

```

Zeile	Fehler	Erläuterung	Korrektur
6	; statt ,	Die einzelnen Parameter müssen durch Kommas getrennt werden.	"%d",
6	& fehlt	Bei der Leseanweisung <code>scanf()</code> müssen die Adressen der Variablen übergeben werden.	& n
7	= statt !=	Der notwendige Operator heißt !=	!=
9	<> statt <	Gemäß Aufgabenstellung muss hier auf negative Zahlen geprüft werden.	n < 0
10	= fehlt	Den Wert einer Variablen erhöht man mittels <code>var += inc.</code>	cnt++;
10	Initialisierung	Das Programm wird meistens ein falsches Ergebnis ausgeben, da die Variable <code>cnt</code> nirgends initialisiert wurde. Dies kann man beispielsweise bei der Variablendefinition (Zeile 5), mittels einer weiteren Anweisung oder durch Verwenden einer <code>for</code> -Schleife erreichen.	<code>for( cnt = 0; n != 0; )</code>
11	"..." fehlen	Die Formatierung innerhalb der Eingabeangabe muss in Gänsefüßchen stehen.	"%d"
13	" fehlt	Der Text bei der Ausgabeangabe muss in Gänsefüßchen stehen.	"..."

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int n, cnt;
6      scanf( "%d", & n );
7      for( cnt = 0; n != 0; )
8      {
9          if ( n < 0 )
10             cnt++;
11             scanf( "%d", & n );
12     }
13     printf( "%d negative Zahlen gefunden\n", cnt );
14 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Eine erste, einfachste Schleife

### 1. Aufgabenstellung

Schreibe ein Programm, das 46 mal den Satz `Nur Uebung macht den Meister` ausgibt. Überlege und begründe (bei der Implementierung), welche der obigen drei Schleifenkonstrukte problemadäquat ist.

### 2. Pflichtenheft

Aufgabe	: 46 Ausgabe des Satzes „Nur Uebung macht den Meister.“
Eingabe	: keine Eingaben.
Ausgabe	: 46 mal der Satz „Nur Uebung macht den Meister.“
Sonderfälle	: keine.

### 3. Testdaten

Entfällt, da keine Daten eingelesen werden müssen.

### 4. Implementierung

Da die Zahl der Schleifendurchläufe bereits vor Eintritt in die Schleife feststeht, nehmen wir eine <code>for</code> -Schleife.
Wiederholte Ausgabe des geforderten Satzes
<pre>for i = 1 bis 46 schrittweite 1   wiederhole Ausgabe Text: "Nur Uebung macht den Meister"</pre>

### 5. Kodierung

<pre>1 #include &lt;stdio.h&gt; 2 3 int main( int argc, char **argv ) 4 { 5     int i; 6     for( i = 1; i &lt;= 46; i++ ) 7         printf( "Nur Uebung macht den Meister\n" ); 8 }</pre>
--

## Aufgabe 2: Eine zweite, einfachste Schleife

### 1. Aufgabenstellung

Schreibe ein Programm, das die Zahlen von 1 bis 100 in einer Zeile ausgibt, wobei zwischen den Zahlen immer genau ein Leerzeichen stehen soll. Überlege und begründe (bei der Implementierung), welche der obigen drei Schleifenkonstrukte problemadäquat ist.

### 2. Pflichtenheft

Aufgabe	: Drucken der Zahlen von 1 bis 100 in einer Zeile.
Eingabe	: keine Eingaben.
Ausgabe	: die Zahlen von 1 bis 100.
Sonderfälle	: keine.

### 3. Testdaten

Entfällt, da keine Daten eingelesen werden müssen.

### 4. Implementierung

Da die Zahl der Schleifendurchläufe bereits vor Eintritt in die Schleife feststeht, nehmen wir eine <code>for</code> -Schleife.
---

Ausgabe der Zahlen von 1 bis 100

```
Ausgabe der Zahl 1
for i = 2 bis 100 schrittweite 1
wiederhole Ausgabe Text: "_" Zahl i
```

**Hinweis:** Wir geben erst die erste Zahl aus und dann die restlichen 99 in einer Schleife. Dadurch vermeiden wir das letzte Leerzeichen zwischen der letzten Zahl und dem Zeilenumbruch.

### 5. Kodierung

<pre>1  #include &lt;stdio.h&gt; 2 3  int main( int argc, char **argv ) 4      { 5          int i; 6          printf( "1" ); 7          for( i = 2; i &lt;= 100; i++ ) 8              printf( " %d", i ); 9          printf( "\n" ); 10     }</pre>
---

## Aufgabe 3: „Rechnen“ mit Schleifen

Die folgenden Aufgaben ähneln sehr den beiden ersten kleinen Aufgaben. Daher kannst du i.d.R. gleich mit der Implementierung oder bei genügend Sicherheit gleich mit der Kodierung anfangen. Überlege auch, inwiefern du die einzelnen „Grenzen“ fest kodierst oder mittels einer geeigneten Eingabeanweisung interaktiv vom Benutzer abforderst. Führe dies an mindestens zwei Aufgaben exemplarisch durch. Auf dieser und der nächsten Seite ist hierfür genügend Platz.

1. Schreibe ein Programm, das mittels einer Schleife die Summe der Zahlen von 1 bis 100 ermittelt; aus Übungsgründen soll *nicht* die Gaußsche Formel verwendet werden.
2. Gib alle natürlichen Zahlen zwischen 1 und 100 aus, die durch 4 teilbar sind.
3. Gib alle natürlichen Zahlen zwischen 1 und 100 aus, die durch 3 und 16 teilbar sind.
4. Schreibe ein Programm, das alle Schaltjahre zwischen 1900 und 2100 (jeweils einschließlich) ausgibt. Die Ausgabe sollte je Zeile eine Jahreszahl enthalten. Zur Erinnerung: Schaltjahre lassen sich durch vier dividieren. Ein Jahr ist aber kein Schaltjahr, wenn es sich durch 100 teilen lässt. Ausgenommen davon sind alle Jahre, die sich aber durch 400 teilen lassen.
5. Gib alle natürlichen Zahlen zwischen 1 und 100 aus, die durch 6 oder 8 teilbar sind. Erweitere das Programm so, dass es nach jeweils 6 gedruckten Zahlen auf den Anfang der nächsten Zeile springt.

Wir präsentieren hier nur die einzelnen Programme. Natürlich gibt es auch viele andere mögliche Lösungen. Wichtig ist immer, dass das Programm das macht, was in der Aufgabenstellung steht. Bei Fragen zu den Unterschieden oder möglichen Alternativen, einfach mal mit den Betreuern sprechen.

### Programm zu Aufgabe 1:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int x, summe = 0;
6      for ( x = 1; x <= 100; x = x + 1 )
7          summe = summe + x;
8      printf( "Die Summe von 1 bis 100 betraegt: %d\n", summe
9             );
9      return 0;
10 }
```

### Programm zu Aufgabe 2:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int x, summe = 0;
6     for ( x = 1; x <= 100; x = x + 1 )
7         if ( x % 4 == 0 )
8             printf( "%d\n", x );
9     return 0;
10 }
```

### Programm zu Aufgabe 3:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int x, summe = 0;
6     for ( x = 1; x <= 100; x = x + 1 )
7     {
8         if ( x % 3 == 0 && x % 16 == 0 )
9             { // this time with braces,
10                // just for illustration purposes
11                printf( "%d\n", x );
12            }
13     }
14     return 0;
15 }
```

### Programm zu Aufgabe 4:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int jahr;
6     for ( jahr = 1900; jahr <= 2100; jahr = jahr + 1 )
7         if ((jahr % 4 == 0 && jahr % 100 != 0)
8             || jahr % 400 == 0 )
9             printf( "%d\n", jahr );
10     return 0;
11 }
```

### Programm zu Aufgabe 5:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int x, spalte = 0, summe = 0;
6      for ( x = 1; x <= 100; x = x + 1 )
7          if ( x % 6 == 0 || x % 8 == 0 )
8          {
9              printf( "%2d ", x );
10             spalte = (spalte + 1) % 6;
11             if ( spalte == 0 )
12                 printf( "\n" );
13         }
14
15         // fuer den fall, dass schon zahlen gedruckt
16         // wurden, wir aber noch kein \n ausgegeben haben,
17         // brauchen wir noch ein \n
18         if ( spalte != 0 )
19             printf( "\n" );
20         return 0;
21     }
```

## Aufgabe 4: „Formatierte“ Ausgabe

In vielen Anwendungen ist es unumgänglich, dass die erzielten Ergebnisse geeignet aufbereitet werden. Ein Teil der zur Verfügung stehenden Möglichkeiten haben wir bereits in Übungspaket 8 kennengelernt. Manchmal reichen diese Möglichkeiten aber nicht aus, sodass man einige Details selbst programmieren muss. In diesem Übungspaket beschäftigen wir uns mit einigen ausgewählten Standardelementen. Bearbeite mindestens zwei der folgenden Aufgaben. Die eigentlichen Algorithmen sind wieder so einfach, dass ihr direkt mit der Implementierung bzw. Kodierung anfangen könnt.

1. Schreibe ein Programm, das die Zahlen von 0 bis 99 in Form einer 10×10 Matrix ausgibt. Die Ausgabe sollte also wie folgt aussehen:

```
0  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 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

**Hinweis:** Die Formatierungsanweisung `%2d` sorgt dafür, dass eine natürliche Zahl mit zwei Stellen ausgegeben wird. Sollte die Zahl einstellig sein, wird ihr ein Leerzeichen vorangestellt.

**Lösungsansatz:** Die Aufgabe ist eigentlich recht einfach. Wir müssen nur die Zahlen von 0 bis 99 ausgeben. Hierfür eignet sich eine `for`-Schleife, da die Zahl der Schleifendurchläufe von Anfang an fest steht. Hinzu kommt jetzt noch, dass wir nach jeweils zehn Zahlen einen Zeilenumbruch ausgeben. Dies bekommen wir einfach mittels des logischen Ausdrucks `x mod 10 == 9` heraus. Daraus ergibt sich folgende abstrakte Formulierung des Programms:

Drucken einer Matrix

```
Variablen: Integer: x
für x = 0 bis 99 schrittweite 1
wiederhole Drucke x ohne Zeilenumbruch
    wenn x mod 10 = 9
    dann Ausgabe Zeilenumbruch
```

Auf der folgenden Seite setzen wir diese einfache abstrakte Formulierung in einen entsprechenden C-Quelltext um.



```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int x;
6
7      // all numbers in one for loop
8      for ( x = 0; x < 100; x = x + 1 )
9      {
10
11         // we print the number
12         printf( "%2d ", x );
13
14         // is a newline required?
15         if ( x % 10 == 9 )
16             printf( "\n" );
17     }
18     return 0;          // done
19 }

```

Alternativ bietet es sich auch an, die inherente 10×10 Struktur der Matrix direkt in das C-Programm umzusetzen. Dies geht direkt mit zwei verschachtelten `for`-Schleifen.

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j;
6
7      // it's 10 lines ...
8      for ( i = 0; i < 10; i = i + 1 )
9      {
10
11         // ... by 10 columns
12         for ( j = 0; j < 10; j = j + 1 )
13             printf( "%2d ", i * 10 + j );
14
15         // finish the current line
16         printf( "\n" );
17     }
18     return 0;          // done
19 }

```

2. Schreibe ein Program, das mittels einer Schleife acht Zeilen druckt, in der jeweils drei Sternchen stehen. Dabei sollen diese drei Sternchen aber in jeder Zeile um ein Leerzeichen mehr als in der vorangehenden Zeile eingerückt werden. Die Ausgabe soll also wie folgt aussehen:

```
***
 ***
  ***
   ***
    ***
     ***
      ***
       ***
```

**Lösungsansatz:** Gemäß der Aufgabenstellung müssen wir acht Zeilen drucken, auf denen sich jeweils drei Sternchen befinden. Wir müssen beachten, dass wir in der ersten Zeile kein, in der zweiten Zeile ein usw. Leerzeichen vor den Sternchen drucken. Hierfür benötigen wir wieder Schleifen. Da die Zahl der Schleifendurchläufe bereits jetzt schon feststehen, eignen sich wiederum `for`-Schleifen.

Drucken von eingerückten Sternchen

```
für x = 0 bis 7 schrittweite 1
wiederhole für l = 0 bis x-1 schrittweite 1
    wiederhole Ausgabe Text " " ohne Zeilenumbruch
    Ausgabe Text "***" mit Zeilenumbruch
```

Daraus lässt sich folgendes C-Programm erstellen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int x, l;
6
7      // one loop for the eight lines
8      for ( x = 0; x < 8; x = x + 1 )
9      {
10         // first, we print the spaces
11         for ( l = 0; l < x; l = l + 1 )
12             printf( " " );
13
14         // and now we print the three stars
15         printf( "***\n" );
16     }
17     return 0;          // done
18 }
```

3. Schreibe ein Programm, das mittels mehrerer Schleifen folgendes „Dreieck“ ausgibt:

```
*****
*****
*****
*****
*****
*****
****
***
**
*
```

**Lösungsansatz:** In dieser Aufgabe müssen wir zehn Zeilen Drucken, die sich in der Zahl der vorangehenden Leerzeichen und Sternchen voneinander unterscheiden. Wir können aber beobachten, dass die Summe aus Leerzeichen und Sternchen in allen Zeilen konstant zehn ergibt. Das führt uns zu folgendem Lösungsansatz:

Drucken eines Sternendreiecks

für x = 0 bis 9 schrittweite 1  
wiederhole Ausgabe von x Leerzeichen  
Ausgabe von 10-x Sternchen

Das ergibt folgendes C-Programm:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int x, l;
6      for ( x = 0; x < 10; x = x + 1 )
7      {
8          // first, printing x spaces
9          for ( l = 0; l < x; l = l + 1 )
10             printf( " " );
11
12         // then, printing 10-x stars
13         for ( ; l < 10; l = l + 1 )
14             printf( "*" );
15
16         // finish the current line
17         printf( "\n" );
18     }
19     return 0;          // done
20 }
```

## Aufgabe 5: Eingabeverarbeitung

Bei Programmen, die interaktive Eingaben eines Nutzers verarbeiten, entsteht immer folgendes „Problem“: Als Programmierer weiß man in der Regel nicht, wie viele Eingaben kommen. Daher muss man sich ein geeignetes „Abbruchkriterium“ überlegen, das das Ende der Eingabeverarbeitung angibt. Manchmal folgt so ein Abbruchkriterium unmittelbar aus der Ausgabenstellung, beispielsweise durch Eingabe eines Wertes, der außerhalb des Gültigkeitsbereiches liegt, manchmal aber muss man dieses Abbruchkriterium als Programmierer einfach sinnvoll definieren. Ferner sollte man vor einer Eingabeaufforderung eine entsprechende Ausgabe machen, damit der Nutzer weiß, was das Programm von ihm will.

### 1. Aufgabenstellung

Entwickle ein Programm, das natürliche Zahlen (also Zahlen größer null) einliest und am Ende die Summe dieser Zahlen ohne das Abbruchkriterium ausgibt.

### 2. Pflichtenheft

Das Pflichtenheft: Aufgabe, Eingabe, Ausgabe, Sonderfälle

Aufgabe : einlesen positiver, ganzer Zahlen und daraus die Summe bilden.  
Eingabe : beliebig viele positive, ganze Zahlen.  
Ausgabe : die Summe der eingelesenen Zahlen.  
Sonderfälle: Zahlen kleiner oder gleich null.

Definiere und begründe die Wahl eines geeigneten Abbruchkriteriums:

Da nur Zahlen *größer* null zugelassen sind, kann jede Zahl kleiner oder gleich null als Abbruchkriterium verwendet werden.

Wähle und begründe ein Schleifenkonstrukt, das dir problemadäquat erscheint:

Da die Zahl der durchzuführenden Schleifendurchläufe nicht von vornherein feststeht, sondern sich aufgrund der Nutzereingaben bestimmen, wählen wir eine **while**-Schleife.

### 3. Testdaten

Testreihe	Zahlenfolge	Summe	Kommentar
1	1 2 3 4 100 -1	110	normale Eingabe, fünf Zahlen
2	34 0 12 99 45 6 -1	34	nur eine Zahl, da Abbruch durch die 0
3	-1	0	-1 ist keine positive Zahl

#### 4. Implementierung

```
Einlesen positiver ganzer Zahlen

Variablen: Integer: Zahl, Summe
Ausgabe einer Eingabeaufforderung
Eingabe Zahl
Setze Summe = 0
solange Zahl > 0
wiederhole Summe = Summe + Zahl
        Eingabe Zahl
Ausgabe Text "Die Summe betraegt:" Summe
```

#### 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int n, summe = 0;
6      printf( "Bitte mehrere Zahlen > 0 eingeben\n" );
7      printf( "Programmende wenn Zahl <= 0\n" );
8      scanf( "%d", & n );          // read first number
9      while( n > 0 )                // as long as we have n>0
10     {
11         summe = summe + n;        // add to the sum
12         scanf( "%d", & n );      // read next number
13     }
14     printf( "Die Summe betraegt: %d\n", summe );
15     return 0;
16 }
```

## Aufgabe 6: Bestimmung eines indirekten Parameters

### 1. Aufgabenstellung

Entwickle ein Programm, das folgende Funktion hat: Der Nutzer muss eine ganze Zahl  $z$  eingeben. Das Programm soll dann denjenigen Index  $i$  bestimmen, für den gilt:  $\sum_i i^3 \leq z$ . Neben dem Index soll das Programm auch die Kubikzahlen und deren Summe ausgeben.

**Beispiel:** Eingabe: 105

Ausgabe: Index= 4 Kubikzahlen= 1 8 27 64 Summe= 100

### 2. Pflichtenheft

Aufgabe	: Bestimmung eines Indexes $i$ , sodass die Summe $s=1^3+\dots+(i-1)^3+i^3$ kleiner oder gleich der eingegebenen Zahl $z$ ist.
Eingabe	: Zahl $z$ .
Ausgabe	: ermittelter Index $i$ sowie die Summe der Kubikzahlen.
Sonderfälle:	Zahlen kleiner null.

### 3. Testdaten

Zahl $z$	Index $i$	Kubiksumme $s$	Zahl $z$	Index $i$	Kubiksumme $s$
0	0	0	100	4	100
10	2	9	101	4	100
99	3	36	-3	Fehlermeldung, da $z < 0$	

### 4. Implementierung

Die Aufgabe ist „eigentlich“ ganz einfach. Aber die Tücke liegt wie immer im Detail. Wir haben drei Werte zu unterscheiden:

1. die Zahl  $z$ , die der Nutzer eingegeben hat
2. den Index  $i$  und
3. die Summe  $s$  der kumulierten Kubikzahlen.

Da gemäß der Aufgabenstellung der Nutzer eine positive Zahl (einschließlich der Null) eingeben soll, sind negative Zahlen nicht erlaubt. Aus diesem Sachverhalt können wir schlussfolgern, dass der kleinstmögliche Index  $i$  und damit auch die Summe  $s$  null sind.

Wir müssen den Index und gleichzeitig auch die Summe  $s$  solange erhöhen, bis wir einen Wert größer als die Zahl  $z$  gefunden haben. Aber in diesem Fall sind beide Werte bereits zu groß; gemäß Aufgabenstellung sollen wir ja den nächst kleineren Wert finden. Wir haben nun zwei Möglichkeiten:

1. In diesem Fall machen wir alles wieder einen Schritt rückgängig. Entsprechend müssen wir den Index um eins und die kumulierte Summe um den entsprechenden Wert reduzieren.
2. Wir führen den Test vorausschauend um eine Kubikzahl höher durch. Sollte also die gemerkte Summe  $s$  plus der nächsten Kubikzahl  $i+1^3$  größer als die eingegebene Zahl  $z$  sein, müssen wir aufhören. Auch wenn dies erst einmal etwas umständlicher erscheint, brauchen wir in diesem Fall anschließend nichts mehr zu korrigieren.

Beide Varianten haben ihre Berechtigung und sind für uns erst einmal gleichwertig. Wir haben uns für die zweite entschieden, um euch einen Ansatz zu zeigen, den ihr vermutlich nicht gewählt habt; ihr sollt ja etwas lernen.

### 5. Kodierung

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int z, i, summe, qubic;
6      scanf( "%d", & z );
7      if ( z >= 0 )
8      {
9          i = summe = 0;
10         qubic = 1;
11         while( summe + qubic <= z )
12         {
13             summe = summe + qubic;
14             i = i + 1;
15             qubic = (i + 1)*(i + 1)*(i + 1);
16         }
17         printf( "Zahl z=%d\n", z );
18         printf( "Index i=%d\n", i );
19         printf( "Summe s=%d\n", summe );
20     }
21     else printf( "Negative Zahlen sind nicht zulaessig\n" );
22     return 0;
23 }

```

# Übungspaket 12

## Der Datentyp char

---

### Übungsziele:

1. Umgang mit dem Datentyp `char`,
2. Deklarationen von `char`-Variablen,
3. `char`-Konstanten
4. und `char`-Rechenoperationen.

### Skript:

Kapitel: 29 bis 31 sowie 24, 25 und 27

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket gibt es einiges zum Thema Zeichen. Die Struktur der Aufgaben ähnelt sehr den vorherigen. Neben der eigentlichen Zeichenverarbeitung ist die Umwandlung von Werten des Datentyps `char` in Werte des Datentyps `int` und umgekehrt Teil dieses Übungspaketes.



# Teil I: Stoffwiederholung

## Aufgabe 1: Notation von Zeichenkonstanten

In Vorlesung und Skript haben wir gesehen, dass in der Programmiersprache C Zeichen in drei verschiedenen Formen angeben können. Vervollständige folgende Tabelle:

Zeichen	C-Notation			
	Zeichen	Escape/Oktal	Ganzzahl	Hexadezimal
Großbuchstabe A	'A'	'\101'	65	0x41
Semikolon	';'	'\73', '\073'	59	0x3B
Ziffer 3	'3'	'\63', '\063'	51	0x33
Nullbyte		'\0', '\00', '\000'	0	0x0

## Aufgabe 2: Ein- und Ausgabe

Erläutere anhand von zwei Beispielen, wie Zeichen ein und ausgelesen werden können?

Format: `%c, %nc`    Eingabe: `scanf( "%c", & c )`    Ausgabe: `printf( "%3c", c )`

## Aufgabe 3: Zeichen und deren ASCII-Werte

Schreibe ein kleines Testprogramm, das die Zeichen 'A' bis 'Z' nebst ihrer zugehörigen ASCII Werte in tabellarischer Form ausgibt. Frage: Muss man bei der Aufgabe die konkrete ASCII-Kodierung der einzelnen Zeichen selber in der ASCII-Tabelle nachschlagen oder kann das der Compiler dies erledigen? nein, der Compiler kann es: 'Zeichen'

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     char c;
6     printf( "Zeichen | Dezimalwert\n" );
7     for( c = 'A'; c <= 'Z'; c = c + 1 )
8         printf( "%7c | %2d\n", c, c );
9 }
```

## Teil II: Quiz

---

### Aufgabe 1: Datentyp char vs. Datentyp int

Der Datentyp `char` gehört zu den Ganzzahldatentypen, da die einzelnen Zeichen einzeln aufgeführt werden können. Aber dennoch gibt es einige Unterschiede zum Datentyp `int`. Diese Unterschiede betreffen in erster Linie die Zahl der verwendeten Bytes im Arbeitsspeicher und damit die Zahl der verfügbaren Bits und dem daraus resultierenden Wertebereich. Ergänze folgende Tabelle:

Datentyp	# der Bytes	# der Bits	Wertebereich
<code>int</code>	4	32	-2 147 483 648 bis 2 147 483 647
<code>char</code>	1	8	-128 bis 127 bzw. 0 bis 255

### Aufgabe 2: Ausdrücke

Welche Werte erhalten die Variablen `char c` und `int i` durch die angegebenen Ausdrücke?

Ausdruck	Resultat
<code>c = 'a' + 1</code>	<code>c = 'b'</code>
<code>c = 'm' - 1</code>	<code>c = 'l'</code>
<code>c = 'a' + 4</code>	<code>c = 'e'</code>
<code>c = 'm' - 'a' + 'A'</code>	<code>c = 'M'</code>

Ausdruck	Resultat
<code>c = 'n' - 'z' + 'Z'</code>	<code>c = 'N'</code>
<code>c = 'P' - 'A' + 'a' + 1</code>	<code>c = 'q'</code>
<code>i = 'z' - 'a'</code>	<code>i = 25</code>
<code>i = '9' - '0'</code>	<code>i = 9</code>

### Aufgabe 3: Datentypen

Im Zusammenhang mit Variablen und Werten vom Typ `char` gibt es immer wieder Fragen bezüglich des resultierenden Datentyps. In den folgenden Ausdrücken gilt folgende Variablendefinition: `char c` und `int i`

Ausdruck	Datentyp	Ausdruck	Datentyp
<code>c</code>	<code>char</code>	<code>c + 1</code>	<code>int</code>
<code>'a'</code>	<code>int</code>	<code>c - '0'</code>	<code>int</code>
<code>'\101'</code>	<code>int</code>	<code>c + i</code>	<code>int</code>

**Hinweis:** Die Anweisung `printf("%d\n", Ausdruck)` gibt einen Hinweis auf den resultierenden Typ. `Ausdruck` kann eine Variable, ein Ausdruck oder ein Typ sein.

## Teil III: Fehlersuche

---

### Aufgabe 1: Summenberechnung mit kleinen Fehlern

Das folgende Programm soll die Summe der Zeichen 'a' bis 'z', 'A' bis 'Z' und '0' bis '9' bestimmen. Leider sind dem Programmierer wieder einmal ein paar kleine Fehler unterlaufen. Finde die 14 Fehler und korrigiere sie:

```
1  #include 'stdio.h'
2
3  int main( int argc, char **argv )
4  {
5      char c,
6      int s.
7      for( s = 'a', c = 'b'; c < 'z'; c = c + 1 )
8          s = c;
9      for( c = 'A'; c >= 'Z"; c =    + 1 )
10         s = s + c;
11      for( c    "9'; c > '0', c = c + 1 )
12         s = s + c;
13      printf( "summe= %d\n", s );
14  }
```

Zeile	Fehler	Erläuterung	Korrektur
1	'...' statt <...>	Die standard .h-Dateien müssen in <...> angegeben werden.	<stdio.h>
5/6	,/. statt ;	Alle Anweisungen müssen mit einem Semikolon abgeschlossen werden.	char c; int x;
7	< statt <=	Gemäß Aufgabentellung muss auch das 'z' mit zur Summe gezählt werden. Daher muss dort der <= Operator stehen.	c <= 'z'
8	s + fehlt	Es fehlt die Summenbildung.	s = s + c
9	" statt '	Einzelne Zeichen gehören in einfache ' '	'Z'
9	c fehlt	So wird c nicht weitergeschaltet.	c = c + 1
11	Die for-Schleife ist mehrfach fehlerhaft	Entweder soll innerhalb der for-Schleife aufwärts oder abwärts gezählt werden. Für keine dieser beiden Möglichkeiten passen die Angaben zusammen. Wir entscheiden uns fürs Abwärtszählen.	for( c = '9'; c >= '0'; c = c - 1 )

Zeile	Fehler	Erläuterung	Korrektur
13	' statt "	Die Ausgabeanweisung benötigt die Textausgaben in Gänsefüßchen "... " eingeschlossen werden.	"su ... "
13	] statt }	Eine Funktion wird durch eine geschweifte Klammer- zu } beendet.	]

### Programm mit Korrekturen:

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char c;
6      int s;
7      for( s = 'a', c = 'b'; c <= 'z'; c = c + 1 )
8          s = s + c;
9      for( c = 'A'; c <= 'Z'; c = c + 1 )
10         s = s + c;
11     for( c = '9'; c >= '0'; c = c - 1 )
12         s = s + c;
13     printf( "summe= %d\n", s );
14 }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Umwandlung von Ziffern in Zahlen

Ein echter Klassiker: Schreibe ein kleines Programm, das *eine einzelne* Ziffer, also die Zeichen von '0' bis '9', in ihren entsprechenden Zahlenwerte, also ints von 0 bis 9, umwandelt. Das folgende Programmschnipsel zeigt, wie ihr es *nicht* machen sollt ...

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char zeichen;
6      int zahl;
7      zeichen = '5';
8      switch( zeichen )
9      {
10         case '0': zahl = 0; break;
11         case '1': zahl = 1; break;
12         case '2': zahl = 2; break;
13         case '3': zahl = 3; break;
14         case '4': zahl = 4; break;
15         case '5': zahl = 5; break;
16         case '6': zahl = 6; break;
17         case '7': zahl = 7; break;
18         case '8': zahl = 8; break;
19         case '9': zahl = 9; break;
20     }
21     printf( "Zeichen= '%c' Zahl= %d\n", zeichen, zahl );
22 }
```

Eine derartige Umwandlung geht mit *einer einfachen Anweisung*. Überlege zunächst, in welchem Zusammenhang die ASCII-Zeichen und deren Kodierung zueinander stehen. Formuliere und erkläre die entsprechende C-Anweisung, und schreibe ein entsprechendes Testprogramm:

Alle Ziffern sind in der ASCII-Tabelle nacheinander angeordnet, sodass sich die Werte benachbarter Ziffern um eins unterscheiden. Beispielsweise ist der (ASCII-) Wert der Ziffer '7' um genau eins größer als der (ASCII-) Wert der Ziffer '6'. Den Wert einer beliebigen Ziffer bekommt man, in dem man vom ASCII-Wert den Wert des Zeichens '0' subtrahiert. Daher kann man einfach formulieren:

```
int_value = ascii_char - '0';
```

**Testprogramm zur einfachen Wandlung von Ziffern in Zahlenwerte:**

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char c;
6      int  value;
7      printf( "Zeichen | Dezimalwert\n" );
8      for( c = '0'; c <= '9'; c = c + 1 )
9      {
10         value = c - '0';
11         printf( "%7c | %2d\n", c, value );
12     }
13 }

```

## Aufgabe 2: Klassifizierung von Zeichen: <ctype.h>

Die Include-Datei <ctype.h> stellt eine Reihe von nützlichen Makros zur Klassifizierung von Zeichen zur Verfügung. Nenne mindestens sechs von ihnen:

Bedeutung	Makroname	Bedeutung	Makroname
Buchstaben	isalpha()	Ziffern	isdigit()
Kleinbuchstaben	islower()	Hex-Ziffern	isxdigit()
Großbuchstaben	isupper()	„Leerzeichen“	isspace()

## Aufgabe 3: Wandeln von hexadezimalen Ziffern

### 1. Aufgabenstellung

Erweiter das Programm aus Aufgabe 1 dieses Teils so, dass es auch hexadezimale Ziffern, 'A' bis 'F' bzw. 'a' bis 'f' korrekt wandelt.

### 2. Pflichtenheft

Aufgabe	: Umwandlung hexadezimaler Ziffern in die entsprechenden Zahlenwerte
Eingabe	: eine hexadezimale Ziffer
Ausgabe	: eingegebene Ziffer und ihr Zahlenwert
Sonderfälle	: Fehlermeldung bei Eingabe nicht-hexadezimaler Zeichen

### 3. Testdaten

Alle Ziffern von '0' bis '9', 'A' bis 'F' und 'a' bis 'f'

## 4. Implementierung

Das Wesentliche haben wir ja schon in der vorherigen Übungsaufgabe gehabt. Jetzt müssen wir nur überprüfen, ob wir eine Ziffer '0'-'9', einen Kleinbuchstaben 'a'-'f' oder einen Großbuchstaben 'A'-'F' vorliegen haben. Je nach dem ändert sich die Umwandlung geringfügig. Im Kern sieht die Änderung also wie folgt aus:

Umwandlung von hexadezimalen Ziffern in Zahlenwerte

```
Variablen: Integer: zahl
           Char: zeichen

wenn  ziffer ≥ '0' und ziffer ≤ '9'
dann  zahl = ziffer - '0'
sonst wenn ziffer ≥ 'a' und ziffer ≤ 'f'
dann  zahl = ziffer - 'a' + 10
sonst wenn ziffer ≥ 'A' und ziffer ≤ 'F'
dann  zahl = ziffer - 'A' + 10
sonst Ausgabe einer Fehlermeldung
```

## 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char c;
6      int  val, error = 0;
7      scanf( "%c", & c );
8      if (c >= '0' && c <= '9')
9          val = c - '0';
10     else if (c >= 'a' && c <= 'f')
11         val = c - 'a' + 10;
12     else if (c >= 'A' && c <= 'F')
13         val = c - 'A' + 10;
14     else error = 1;
15     if ( error )
16         printf( "'%c' ist kein hex-zeichen\n", c );
17     else printf( "%c hat den wert %d\n", c, val );
18 }
```

# Aufgabe 4: Umwandlung von Klein in Großbuchstaben

## 1. Aufgabenstellung

Entwickle ein Programmstückchen, das Kleinbuchstaben in Großbuchstaben umwandeln kann und umgekehrt. Auch diese Umwandlung geht wieder in einer Zeile.

## 2. Pflichtenheft

Aufgabe	: Umwandlung von Klein- in Großbuchstaben und umgekehrt
Eingabe	: keine; die entwickelten Anweisungen werden direkt in Schleifen aufgerufen
Ausgabe	: Paare von Klein- und Großbuchstaben in Tabellenform
Sonderfälle	: keine

## 3. Testdaten

Alle Buchstaben von 'a' bis 'z' und 'A' bis 'Z'.

## 4. Implementierung

Da diese Aufgabe eigentlich keine größeren Schwierigkeiten bereiten sollte, konzentrieren wir uns hier auf die eigentliche Umwandlung der Zeichen. Wie lautet die jeweiligen Anweisungen zum Umwandeln der Buchstaben?

Klein- in Großbuchstaben: `upper = lower - 'a' + 'A'`

Groß- in Kleinbuchstaben: `lower = upper - 'A' + 'a'`

## 5. Kodierung

Schreibe ein entsprechendes Testprogramm für die beiden obigen Anweisungen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char c;
6      printf( "klein | gross\n" );
7      for( c = 'a'; c <= 'z'; c = c + 1 )
8          printf( "%5c | %c\n", c, c - 'a' + 'A' );
9      printf( "\ngross | klein\n" );
10     for( c = 'A'; c <= 'Z'; c = c + 1 )
11         printf( "%5c | %c\n", c, c - 'A' + 'a' );
12 }
```



# Aufgabe 5: Zeichen-Matrix

## 1. Aufgabenstellung

Zur Wiederholung: Eine Variable vom Typ `char` besitzt in der Regel acht Bits, womit sich 256 verschiedene Werte darstellen lassen. Entwickle ein Programm, das diese 256 Möglichkeiten in Form einer 16×16 Matrix darstellt, und zwar nicht als Zahlenwerte sondern als Zeichen. Da einige Zeichen „merkwürdig“ aussehen, ist es günstig " %2c" als Formatierung zu nehmen. Mit etwas Glück wird folgende Matrix ausgegeben.

Gewünschte Ausgabe:

0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	
	i	ç	£	€	¥	§	§	©	®	«	¬		®		
°	±	²	³	¼	½	¶	·	¸	¹	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

## 2. Pflichtenheft

Aufgabe	: Darstellung aller 256 Zeichen in Form einer 16×16 Matrix
Eingabe	: keine, die Tabelle wird direkt ausgegeben
Ausgabe	: Ausgabe der Tabelle
Sonderfälle	: keine

## 3. Implementierung

Ausgabe aller 256 Zeichen als Matrix
setze zeichen = 0
für i = 0 bis 16 schrittweite 1
wiederhole
für j = 0 bis 16 schrittweite 1
wiederhole
setze zeichen = zeichen + 1
Ausgabe zeichen
Ausgabe Zeilenumbruch

#### 4. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j, c;
6      for( c = i = 0; i < 16; i++ )
7      {
8          for( j = 0; j < 16; j++, c++ )
9              printf( " %2c", c );
10             printf( " \n" );
11     }
12 }
```

#### 5. Alternative-I: Berechnung des auszugebenen Zeichens in Zeile 9:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j;
6      for( i = 0; i < 16; i++ )
7      {
8          for( j = 0; j < 16; j++ )
9              printf( " %2c", i * 16 + j );
10             printf( " \n" );
11     }
12 }
```

#### 6. Alternative-II: Test auf die notwendigen Zeilenumbrüche in den Zeilen 9 und 10:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int c;
6      for( c = 0; c < 256; c++ )
7      {
8          printf( " %2c", c );
9          if ( c % 16 == 15 )
10             printf( " \n" );
11     }
12 }
```

# Übungspaket 13

## Der Datentyp `double`

---

### Übungsziele:

1. Umgang mit dem Datentyp `double`,
2. Deklarationen von `double`-Variablen,
3. `double`-Konstanten
4. und `double`-Rechenoperationen.

### Skript:

Kapitel: 32

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket gibt es einiges zum Thema Fest- bzw. Fließkommazahlen. Die Struktur der Aufgaben ähnelt sehr den vorherigen. Neben der eigentlichen Verarbeitung von Fließkommazahlen ist die Umwandlung von Werten des Datentyps `int` in den Datentyp `double` und umgekehrt Gegenstand der Aufgaben.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Konstanten

In Skript und Vorlesung haben wir zwei Möglichkeiten (Festkomma- und Exponentialform) kennen gelernt, mittels derer man Gleitkommazahlen notieren kann. Aus welchen vier Komponenten kann eine Gleitkommazahl bestehen?

- |                              |             |
|------------------------------|-------------|
| 1. Vorzeichen der Zahl       | 2. Mantisse |
| 3. Vorzeichen des Exponenten | 4. Exponent |

Durch die Möglichkeit, einzelne Teile quasi weg zu lassen bzw. stark zu verkürzen gibt es reichlich Kombinationsmöglichkeiten, die teilweise fast schon „abstrus“ aussehen. Bilde mindestens zehn Zahlen, anfangs einfache, am Ende möglichst merkwürdig aussehende. Im Zweifelsfalle immer mal das Syntaxdiagramm aus Kapitel 32 konsultieren.

C-Syntax	Wert	C-Syntax	Wert	C-Syntax	Wert
123.456	123,456	-4.5E1	-45,0	-.1	-0,1
-.123	-0,123	20E-2	0,2	-1E-4	-0,0001
1.2E-3	0,0012	0.1E2	10,0	.2E2	20,0
-.056E3	-56,0	1E2	100,0	-2E2	-200,0

## Aufgabe 2: Ein- und Ausgabe

In Skript und Vorlesung haben wir kurz erwähnt, dass `double`-Werte mittels `%lf` und `%e` ein- und ausgegeben werden können. Die Möglichkeiten sind vielfältig, wie man beispielsweise auf der Manual Page (`man 3 printf`) sehen kann. Ziel dieser Aufgabe ist es, diese Möglichkeiten ein wenig auszuloten. Schreibe dazu ein kleines Testprogramm, das mindestens zehn unterschiedliche Ausgabeformatierungen mit jeweils einem aktuellen Wert ausgibt. Notiere die verwendeten Formatierungen und Ausgaben in folgender Tabelle.

Format	Wert	Ausgabe	Format	Wert	Ausgabe
%8.1f	123.456	___123.4	%8.1e	123.456	__1.2e+02
%8.2f	123.456	__123.45	%8.2e	123.456	1.23e+02
%8.3f	123.456	_123.456	%8.3e	123.456	1.235e+02
%8.4f	123.456	123.4560	%8.4e	123.456	1.2346e+02
%5.3f	123.456	123.456	%8.5e	123.456	1.23456e+02

## Teil II: Quiz

---

### Aufgabe 1: Ausdrücke: double versus int

Selbst bei fortgeschrittenen Programmierern gibt es immer wieder Fragezeichen im Gesicht, wenn in Ausdrücken `double` und `int`-Werte bzw. Variablen gemischt auftauchen. Die beiden Grundregeln lauten:

1. Vor Ausführung einer arithmetischen Operation werden beide Operanden in den selben (größeren) Datentyp gewandelt.
2. Beim Zuweisen eines Gleitkommawertes an eine `int`-Variable werden die Nachkommastellen immer abgeschnitten.

Für das folgende Quiz gilt: `int i1 = 2`, `i2 = 3` und `double d1 = 2.0`, `d2 = 5.0`.

Ausdruck	Typ	Wert
<code>1 + 2</code>	<code>int</code>	<code>3</code>
<code>2.0 + 3.0</code>	<code>double</code>	<code>5.0</code>
<code>2 + 4.0</code>	<code>double</code>	<code>6.0</code>
<code>2.0*(3 + 4)</code>	<code>double</code>	<code>14.0</code>
<code>i1 + i2</code>	<code>int</code>	<code>5</code>
<code>2 + d2</code>	<code>double</code>	<code>7.0</code>
<code>d1 - d2 - i1</code>	<code>double</code>	<code>-5.0</code>

Ausdruck	Typ	Wert
<code>16 / 5</code>	<code>int</code>	<code>3</code>
<code>17 / 6</code>	<code>int</code>	<code>2</code>
<code>15 / 6.0</code>	<code>double</code>	<code>2.5</code>
<code>i2 / 4</code>	<code>int</code>	<code>0</code>
<code>(i1 + i2) / 2.0</code>	<code>double</code>	<code>2.5</code>
<code>i1/(i2 - i1 - 1)</code>	---	Fehler
<code>d1/(i2 - i1 - 1)</code>	<code>double</code>	<code>∞</code>

### Aufgabe 2: Ausgaben

Wie sehen die resultierenden Ausgaben aus?

Anweisung	Ausgabe
<code>printf("%f",123.456)</code>	<code>123.456000</code>
<code>printf("%e",12.34)</code>	<code>1.234000e+01</code>
<code>printf("%6.2f",123.4)</code>	<code>123.40</code>
<code>printf("%9.4f",123.4)</code>	<code> 123.4000</code>
<code>printf("%6.2f",-1.2)</code>	<code> -1.20</code>

Anweisung	Ausgabe
<code>printf("%.2f",-0.1234)</code>	<code>-0.12</code>
<code>printf("%.2f",-1234.)</code>	<code>-1234.00</code>
<code>printf("%.3e",12.34)</code>	<code>1.234e+01</code>
<code>printf("%.1e",-12.34)</code>	<code> -1.2e+01</code>
<code>printf("%.1e",12.34)</code>	<code> 1.2e+01</code>

## Teil III: Fehlersuche

---

### Aufgabe 1: Verarbeitung von Fließkommazahlen

Das folgende Programm soll alle Zahlen von 1.0, 1.1, 1.2, ..., 1.8 aufsummiert und das Ergebnis in Form von `xxx.xx` ausgeben. Leider sind wieder ein paar Fehler versteckt, die ihr finden sollt.

```
1 #incLude <stdio h>
2
3 int main( int argc; char **argv )
4 {
5     double x. deltax, sum;
6     sum = 0; deltax = 0,1;
7     for( x = 1.0; x < 1.75, x = x    deltax )
8         sum = sum * x;
9     printf( "summe= %6.2e\n", summe );
10 }
```

Zeile	Fehler	Erläuterung	Korrektur
1	incLude	Die <code>#include</code> -Direktive schreibt sich mit kleinem „l“.	<code>include</code>
3	; statt ,	Die Parameter müssen mit einem Komma voneinander getrennt werden.	,
5	. statt ,	Die Parameter müssen mit einem Komma voneinander getrennt werden.	,
6	, statt .	Gleitpunktzahlen werden nicht mit einem Komma sondern einem Punkt notiert, da es sich bei C um eine englischsprachige Programmiersprache handelt.	.
7	1.75	Da die Schleifenbedingung <code>x &lt; 1.75</code> lautet, wird die 1.8 nicht, wie eigentlich gefordert, berücksichtigt.	1.85
7	, statt ;	In der <code>for</code> -Schleife werden die einzelnen Teile mittels Semikolons voneinander getrennt.	;
7	fehlendes +	Es fehlt der Operator zum aktualisieren des <code>x</code> -Wertes.	+
8	* statt +	Zum Bilden der Summe muss der <code>+</code> -Operator verwendet werden.	+
9	falsche Formatierung	Da die Ausgabe im Festkommaformat erfolgen soll, muss ein <code>f</code> zur Formatierung verwendet werden.	<code>%6.2f</code>

**Programm mit Korrekturen:** Das folgende Programm entspricht dem vorherigen, in dem die vorhandenen Fehler behoben wurden.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4      {
5          double x, deltax, sum;
6          sum = 0.0; deltax = 0.1;
7          for( x = 1.0; x < 1.85; x = x + deltax )
8              sum = sum + x;
9          printf( "summe= %6.2f\n", sum );
10     }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Umwandlung von double nach int

### 1. Aufgabenstellung

Jedem sollte klar sein, dass der Datentyp `int` nur eine Untermenge des Datentyps `double` ist. Entsprechend muss man sich bei der Konvertierung von `double` nach `int` Gedanken darüber machen, ob man rundet oder abschneidet. Wie im Skript erläutert, wird in der Programmiersprache C generell abgeschnitten. Daher sollt Ihr in dieser Übungsaufgabe ein Stück Programm entwickeln, das bei der Konvertierung *rundet*. Dabei ist darauf zu achten, dass auch negative Zahlen richtig gerundet werden.

### 2. Pflichtenheft

Aufgabe	: Wandlung von <code>double</code> -Werte in <code>int</code> -Werte <i>mit</i> Rundung
Eingabe	: ein <code>double</code> -Wert
Ausgabe	: ein <code>int</code> -Wert
Vorüberlegungen:	Beim Runden von Zahlen muss man zwischen positiven und negativen Zahlen unterscheiden. Positive Zahlen werden ab .5 aufsonst abgerundet. Beispiel: 3.5 wird auf 4 aufgerundet, wohingegen 3.3 auf 3 abgerundet wird. Negative Zahlen werden genau andersherum behandelt. Beispielsweise wird eine -2.6 auf -3 <i>abgerundet</i> , wohingegen eine -2.45 auf -2 <i>aufgerundet</i> wird.

### 3. Testdaten

Eingabe:	0	0.1	-0.1	3.4	3.5	-5	-5.4	-5.5
Ausgabe:	0	0	0	3	4	-5	-5	-6

### 4. Implementierung

Das Runden positiver Zahlen kann man erreichen, wenn man zuerst 0.5 addiert und <i>anschließend</i> die Nachkommastellen abschneidet. Bei negativen Zahlen muss man statt dessen 0.5 subtrahieren. Für die Konvertierung eignet sich folgender Ansatz:
Runden von double-Zahlen
Variablen: Double: in Integer: out
wenn in >= 0.0
dann out = in + 0.5
sonst out = in - 0.5

### 5. Kodierung



```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      double in;
6      int out;
7      printf( "Bitte reelle Zahl eingeben: " );
8      scanf( "%lf", & in );
9      if ( in >= 0.0 )
10         out = in + 0.5;
11     else out = in - 0.5;
12     printf( "in: %6.3f out: %d\n", in, out );
13 }

```

## Aufgabe 2: Drucken einer tabellierten Sinus-Funktion

### 1. Aufgabenstellung

In der Schule konnten wir die Sinus-Werte für gegebene Winkelargumente in verschiedenen Tafeln nachschlagen. Heutzutage sind diese Tafeln nicht mehr so verbreitet wie früher. Daher entwickeln wir in dieser Übungsaufgabe ein kleines Programm, das uns die Sinus-Werte in einem gegebenen Bereich tabellarisch ausgibt.

Entwickle ein Programm, das  $n$  Wertepaare, bestehend aus dem Argument und dem korrespondierendem Funktionswert, im Intervall  $[x_{\min}..x_{\max}]$  in Tabellenform ausgibt. Für  $x_{\min} = 0$ ,  $x_{\max} = 90^\circ$  und  $n = 10$  könnte die Ausgabe wie folgt aussehen:

x	sin(x)
0.0000	0.00000
10.0000	0.17365
20.0000	0.34202
30.0000	0.50000
40.0000	0.64279
50.0000	0.76604
60.0000	0.86603
70.0000	0.93969
80.0000	0.98481
90.0000	1.00000

Bei Interesse kann das Programm auch dahingehend erweitert werden, dass es die Parameter  $x_{\min}$ ,  $x_{\max}$  und  $n$  interaktiv einliest.

### 2. Pflichtenheft

Aufgabe : tabellarische Darstellung von  $x$  und  $\sin x$   
 Eingabe :  $x_{\min}$ ,  $x_{\max}$  und  $n$   
 Ausgabe : Tabelle mit Tabellenkopf  
 Sonderfälle: 1. Test ob  $n > 0$  und 2. Test ob  $x_{\min} < x_{\max}$   
 ggf. Fehlermeldung ausgeben und Programm abbrechen

### 3. Testdaten

$x_{\min}$	$x_{\max}$	$n$
0	90	10
0	90	2
-90	90	20

### 4. Implementierung

Ausgehend von der Aufgabenstellung können wir folgende Überlegungen anstellen. Erstens müssen wir die notwendigen Parameter  $x_{\min}$ ,  $x_{\max}$  und  $n$  einlesen und diese auf ihre Plausibilität prüfen. Sollte alles ok sein, müssen wir die Tabelle drucken. Da diese neben dem Tabellenkopf aus genau  $n$  Zeilen besteht, sollten wir hierfür eine `for`-Schleife nehmen. Das Berechnen der Funktionswerte und Ausgeben beider Zahlen ist für uns mittlerweile recht einfach.

Erstellen einer tabellierten Sinus-Funktion

```

Variablen: Double: x, x_min, x_max, f
           Integer: n, i

Lese  x_min und x_max
wenn  x_min < x_max
dann  Lese n
      wenn  n > 1
      dann  Tabellenkopf ausgeben
            für i = 0 bis n schrittweite 1
            wiederhole setze x = x_min + i*(x_max - x_min)/(n - 1)
                      setze f = sin(x)
                      Ausgabe x und f
      sonst Fehlermeldung ausgeben
sonst Fehlermeldung ausgeben
  
```

Das ist wieder nur eine von vielen Lösungsmöglichkeiten. Am besten vergleicht ihr wieder eure Lösung mit der hier vorgestellten und analysiert die Unterschiede.

### 5. Kodierung

Hinweise:

1. Die `sin()`-Funktion und die Konstante `M_PI` (für  $\pi$ ) sind in `math.h` definiert.
2. Beim Binden müsst Ihr die Option `-lm` angeben, damit die `sin()`-Funktion auch dazu gebunden wird. Beispiel: `gcc supi-ich.c -o supi-ich -lm`
3. Schaut euch in der Dokumentation an, was für Argumente die `sin()`-Funktion erwartet.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main( int argc, char **argv )
5  {
6      double x, f, x_min, x_max;
7      int    i, n;
8      printf( "Bitte folgende Parameter eingeben\n" );
9      printf( "xmin: " ); scanf( "%lf", & x_min );
10     printf( "xmax: " ); scanf( "%lf", & x_max );
11     if ( x_min < x_max )
12     {
13         printf( "n      : " ); scanf( "%d", & n );
14         if ( n > 1 )
15         {
16             printf( "\n" );
17             printf( " x          | sin(x)\n" );
18             printf( "-----+-----\n" );
19             for( i = 0; i < n; i = i + 1 )
20             {
21                 x = x_min + i*(x_max - x_min)/(n - 1);
22                 f = sin( x * 2.0 * M_PI / 360.0 );
23                 printf( "%8.4f | %8.5f\n", x, f );
24             }
25         }
26         else printf( "Sorry, n > 1 muss gelten\n" );
27     }
28     else printf( "Sorry, xmin < xmax muss gelten\n" );
29     return 0;
30 }
```

Die auf der vorherigen Seite gezeigte Lösung entspricht dem „traditionellen“ Vorgehen des Software Engineering, nach dem ein Programm oder eine Funktion eine Stelle hat, an der sie ihre Arbeit beendet hat. Dies ist die Letzte Zeile. Viele C-Programmierer bevorzugen einen anderen Stil, den wir beispielhaft auf der nächsten Seite zeigen. Das Schema ist, dass die Tests nach Fehlern schauen und das Programm ggf. mitten drin beenden. Dies erkennt man daran, dass es nach einer Fehlerausgabe eine entsprechende **return**-Anweisung gibt.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main( int argc, char **argv )
5  {
6      double x, f, x_min, x_max;
7      int i, n;
8      printf( "Bitte folgende Parameter eingeben\n" );
9      printf( "xmin: " ); scanf( "%lf", & x_min );
10     printf( "xmax: " ); scanf( "%lf", & x_max );
11     if ( x_min >= x_max )
12     {
13         printf("Sorry, es muss gelten: xmin<xmax!\n");
14         return 1;
15     }
16     printf( "n : " ); scanf( "%d", & n );
17     if ( n <= 1 )
18     {
19         printf( "Sorry, es muss gelten: n > 1 !\n" );
20         return 1;
21     }
22     printf( "\n" );
23     printf( " x          | sin(x)\n" );
24     printf( "-----+-----\n" );
25     for( i = 0; i < n; i = i + 1 )
26     {
27         x = x_min + i*(x_max - x_min)/(n - 1);
28         f = sin( x * 2.0 * M_PI / 360.0 );
29         printf( "%8.4f | %8.5f\n", x, f );
30     }
31 }
```

Probiert beide Stile einfach aus und findet denjenigen, mit dem ihr am besten zurecht kommt.

# Aufgabe 3: Euro-Dollar Konvertierung

## 1. Aufgabenstellung

Manchmal fliegt man in die USA um Urlaub zu machen oder um zu arbeiten. Entsprechend muss man auch Geld wechseln ;-) Entwickle ein Programm, das zuerst den Wechselkurs einliest, und anschließend nacheinander Dollar-Beträge einliest, in Euro-Beträge wandelt und diese ausgibt.

## 2. Pflichtenheft

Da mehrere Euro-Beträge verarbeitet werden sollen, ist die Schleifenbedingung eine wesentliche Frage des Pflichtenheftes.

Aufgabe	: tabellarisches Darstellen von Euro und Dollar-Beträgen
Eingabe	: Wechselkurs und beliebig viele Euro-Beträge
Ausgabe	: Euro und Dollarbeträge
Sonderfälle	: Negative Wechselkurse sind nicht zulässig
Abbruchbedingung	: Negative Euro-Beträge beenden das Programm

## 3. Testdaten

Wechselkurs	Euro-Betrag	1.00	10.00	11.11	86.12	13.24
1.34	Dollar-Betrag	1.34	13.40	14.89	115.14	17.74

## 4. Implementierung

Gemäß Aufgabenstellung muss ein Wechselkurs eingelesen werden. Dieser muss positiv sein. Anschließend sollen beliebig viele Dollar-Beträge gelesen und in Euro-Beträge umgerechnet werden. Dies wird solange wiederholt, bis ein negativer Dollar-Betrag eingegeben wird. Da sich also das Ende der Schleife erst durch die Nutzereingaben ergibt, wählen wir eine `while`-Schleife. Der Rest ist eigentlich relativ einfach.

Programm zum ermitteln von Eurobeträgen

```
Variablen: Double: wechsellkurs, euro, dollar
lese wechsellkurs
wenn wechsellkurs > 0
dann lese dollar
    solange dollar >= 0
        wiederhole setze euro = dollar * wechsellkurs
                    ausgabe euro und dollar
                    lese dollar
sonst Ausgabe Fehlermeldung
```

## 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      double exrate, euro, dollar;
6      printf( "Bitte Wechselkurs eingeben: " );
7      scanf( "%lf", & exrate );
8      printf( "Bitte Dollar-Betrag eingeben: " );
9      scanf( "%lf", & dollar );
10     while( dollar >= 0.0 )
11     {
12         euro = dollar * exrate;
13         printf( "euro= %6.2f dollar= %6.2f\n",
14             euro, dollar );
15         printf( "Bitte Dollar-Betrag eingeben: " );
16         scanf( "%lf", & dollar );
17     }
18 }
```

## Aufgabe 4: Reihenschaltung von Widerständen

### 1. Aufgabenstellung

Die Aneinanderreihung von  $n$  Widerständen  $R_1, \dots, R_n$  führt zu einem Reihenwiderstand  $R = R_1 + \dots + R_n = \sum_{i=1}^n R_i$ . Entwickle ein Programm, dass der Reihe nach verschiedene Widerstandswerte einliest, den Gesamtwiderstand ausrechnet und diesen am Ende ausgibt. Bei dieser Aufgabe steht die Zahl der einzulesenden Werte *nicht* fest, sondern soll durch die Eingabe eines geeigneten Wertes erkannt werden.

**Hinweis:** Die Wahl der Schleifenbedingung und der verwendeten Datentypen sollte pragmatisch und nicht esoterisch erfolgen ;-). Das betrifft auch die Zahl der Nachkommastellen.

### 2. Pflichtenheft

Aufgabe	: Berechnen eines resultierenden Reihenwiderstandes
Eingabe	: Eine beliebige Zahl von Widerstandswerten
Ausgabe	: Der resultierende Reihenwiderstand wird ausgegeben
Sonderfälle	: Negative Widerstandswerte sind nicht zulässig
Abbruchbedingung	: Die Eingabe eines negativen Widerstandswertes beendet das Programm

### 3. Testdaten

Widerstandswerte						Resultat
1.0	14.0	101.0				116.0
13.4						13.4
1.0	1.0	2.0	3.0	3.0	4.0	14.0

#### 4. Implementierung

Diese Aufgabenstellung ist äußerst einfach: Wir müssen nur der Reihe nach Widerstandswerte einlesen und aufaddieren. Da sich die Zahl der Wiederholungen erst durch die Nutzereingaben ergibt, wählen wir wieder eine **while**-Schleife. Negative Widerstandswerte beenden die Schleife. Am Ende geben wir noch eine schöne Ergebniszeile aus.

Programm zur Berechnung in Reihe geschalteter Widerstände

```

Variablen: Double: r, total
setze total = 0.0
lese r
solange r >= 0
wiederhole setze total = total + r
           lese r
Ausgabe: Text: "Gesamtwiderstand" Wert: total

```

#### 5. Kodierung

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      double r, total;
6      printf( "Bitte Widerstandswert eingeben: " );
7      total = 0.0;
8      scanf( "%lf", & r );
9      while( r >= 0 )
10     {
11         total = total + r;
12         printf( "Bitte Widerstandswert eingeben: " );
13         scanf( "%lf", & r );
14     }
15     printf( "Gesamtwiderstand: %6.1f\n", total );
16 }

```

# Aufgabe 5: Parallelschaltung von Widerständen

## 1. Aufgabenstellung

Die Parallelschaltung von  $n$  Widerständen  $R_1, \dots, R_n$  führt zu einem Gesamtwiderstand  $R = 1/(1/R_1 + \dots + 1/R_n) = 1/(\sum_{i=1}^n 1/R_i)$ . In der Elektrotechnik ist auch folgende Schreibweise üblich:  $1/R = (1/R_1 + \dots + 1/R_n) = \sum_{i=1}^n 1/R_i$ . Diese Aufgabe ist naturgemäß deutlich schwieriger als die vorherige. Da sie aber zu den Standardaufgaben eines Elektrotechnikers gehört, sollten wir sie unbedingt bearbeiten. Entwickle ein Programm, das der Reihe nach verschiedene Widerstandswerte einliest, den Gesamtwiderstand der Parallelschaltung dieser Widerstände ausrechnet und diesen am Ende ausgibt. Bei dieser Aufgabe steht die Zahl der einzulesenden Werte *nicht* fest, sondern soll durch die Eingabe eines geeigneten Wertes erkannt werden.

**Hinweis:** Die Wahl der Schleifenbedingung und der verwendenden Datentypen sollte pragmatisch und nicht esoterisch erfolgen ;-)

## 2. Pflichtenheft

Aufgabe	: Berechnen eines resultierenden Parallelwiderstandes
Eingabe	: Eine beliebige Zahl von Widerstandswerten
Ausgabe	: Der resultierende Parallelwiderstand wird ausgegeben
Sonderfälle	: Widerstandswerte, die kleiner oder gleich null sind, sind nicht zulässig
Abbruchbedingung	: Die Eingabe eines Widerstandswertes, der kleiner oder gleich null ist, beendet das Programm

## 3. Testdaten

Widerstandswerte						Resultat
2.0	2.0	2.0				0.7
13.4						13.4
10.0	67.0	35.0	13.0	89.0	40.0	3.9



#### 4. Implementierung

Diese Aufgabe so ähnlich wie die vorherige. Entsprechend gelten die dort gemachten Vorüberlegungen auch hier. Da wir aber eine Division durch Null vermeiden müssen, sind jetzt nur noch strikt positive Widerstandswerte erlaubt.

Programm zur Berechnung einer Parallelschaltung

```
Variablen: Double: r, total
setze total = 0.0
lese r
solange r > 0
wiederhole setze total = total + 1.0 / r
           lese r
wenn total > 0.0
dann  ausgabe 1.0 / total
sonst Ausgabe Fehlermeldung
```

#### 5. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      double r, total = 0.0;
6      printf( "Bitte Widerstandswert eingeben: " );
7      scanf( "%lf", & r );
8      while( r > 0 )
9      {
10         total = total + 1.0 / r;
11         printf( "Bitte Widerstandswert eingeben: " );
12         scanf( "%lf", & r );
13     }
14     if ( total > 0.0 )
15         printf( "Gesamtwiderstand: %6.1f\n", 1/total );
16     else printf( "Leider wurde kein Wert eingeben\n");
17 }
```

# Übungspaket 14

## Eindimensionale Arrays

---

### Übungsziele:

Deklaration und Verwendung eindimensionaler Arrays

### Skript:

Kapitel: 33

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Nach dem wir nun die wesentlichen Datentypen kennengelernt haben, kommt etwas neues dazu; die eindimensionalen Arrays. Mit Arrays kann man mehrere Variablen gleichen Typs zusammenfassen, so wie wir es aus der Mathematik in Form von Vektoren, (indizierten) Koeffizienten und dergleichen kennen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Aufbau eines eindimensionalen Arrays

Aus welchen Komponenten besteht eine Array-Definition (ohne Initialisierung)?

1.
2.
3.
4.

## Aufgabe 2: Array-Größe und gültige Indizes

Nehmen wir an, wir hätten ein Array der Größe  $n$  (eines mit  $n$  Elementen), wobei  $n$  stellvertretend für eine ganzzahlige Konstante wie beispielsweise 14 ist.

Frage: Was sind die gültigen Indizes?

## Aufgabe 3: Speicherbelegung

Wie viele Bytes belegt ein Array mit  $n$  Elementen mindestens im Arbeitsspeicher (RAM), wenn der Elementtyp genau  $b$  Bytes belegt?

Skizziere ein Array `a` mit 6 Elementen vom Typ `int` unter der Annahme, dass ein `int` genau 4 Bytes belegt und die Startadresse des Arrays `0x1000` ist.

Definition:

Adresse	Variable	Typ	Größe
0x1018			
0x1014	a[ 5 ]	int	4 Bytes
0x1010	a[ 4 ]	int	4 Bytes
0x100C	a[ 3 ]	int	4 Bytes
0x1008	a[ 2 ]	int	4 Bytes
0x1004	a[ 1 ]	int	4 Bytes
0x1000	a[ 0 ]	int	4 Bytes

## Teil II: Quiz

---

### Aufgabe 1: „Standard“ Arrays

Gegeben seien die folgenden vier Definitionen:

1. `int a[ 3 ];`
2. `char b[ 4 ];`
3. `double c[ 2 ];`
4. `int d[ 1 ];`

Welche Indizes sind bei den jeweiligen Arrays erlaubt und welche Werte haben die Array-Elemente unmittelbar nach der Definition?

Array	Indizes	Werte
a	0, 1, 2	nicht initialisiert
b	0, 1, 2, 3	nicht initialisiert
c	0, 1	nicht initialisiert
d	0	nicht initialisiert

### Aufgabe 2: Implizit definierte Arrays

Die Größe eines Arrays kann man auch „implizit“ durch eine zusätzliche Initialisierung definieren. Gegeben seien die folgenden vier Definitionen:

1. `int e[] = { 3, 4, 5 };`
2. `int f[] = { -1, -3 };`
3. `double g[] = { , , 23.0, };`
4. `double h[] = { -3,0 , , 2.0 };`

Welche Größen haben die Arrays, welche Indizes sind jeweils erlaubt und welche Werte haben die einzelnen Elemente unmittelbar nach der Definition?

Array	Größe	Indizes	Werte
e	3	0, 1, 2	3, 4, 5
f	2	0, 1	-1, -3
g	4	0, 1, 2, 3	n.i., n.i., 23.0, n.i.
h	4	0, 1, 2, 3	3.0, 0.0, n.i., 2.0

n.i.  $\hat{=}$  nicht initialisiert

## Teil III: Fehlersuche

---

### Aufgabe 1: Arbeiten mit mehreren Zahlen

Das folgende Programm soll zehn Zahlen einlesen und in einem Array ablegen, anschließend die Summe ausrechnen und diese am Ende ausgeben. Leider hat DR. ONE-NEURON wieder diverse Fehler gemacht, die ihr finden und korrigieren sollt.

```
1  #include <stdioh>
2
3  #define N=10
4
5  int man( int argc, char **argv )
6  {
7      int i, sum, a[ N ];
8      for( i = 0, i < N, i = i + 1 )
9      {
10         printf( Bitte Wert fuer a[ %d ] eingeben: , i );
11         scanf( "%d", & (a[ ] ) );
12     }
13     for( i = 1; sum = 0; i <= N; i = i + 1 )
14         sum = sum + a{ i };
15     printf( "Die Summe der Zahlen betraegt %d\n", N, sum );
16 }
```

Zeile	Fehler	Erläuterung	Korrektur
1	. fehlt	Der Name der Datei ist <code>stdio.h</code>	<code>&lt;stdio.h&gt;</code>
3	= zu viel	Die <code>#define</code> -Direktive hat zwei „Argumente“, erst das, was definiert werden soll, und dann die eigentliche Definition; ein <code>=</code> ist hier <i>nicht</i> erlaubt.	<code>#define N 10</code>
5	Tippfehler: <code>man</code>	Das Hauptprogramm heißt nicht <code>man()</code> sondern <code>main()</code> .	<code>main</code>
8	, statt ;	Die einzelnen Teile der <code>for</code> -Anweisung werden mittels Semikolons getrennt.	<code>for( ...; ...; ...)</code>
10	" fehlen	Die Texte in der Ausgabeanweisung müssen in Gänsefüßchen eingeschlossen sein.	<code>"..."</code>
11	Index <code>i</code> fehlt	Beim Zugriff auf die einzelnen Arrayelemente muss der jeweilige Index angegeben werden.	<code>a[ i ]</code>

Zeile	Fehler	Erläuterung	Korrektur
13	; statt ,	Werden bei der <b>for</b> -Anweisung in einem der drei Teile mehr als ein Ausdruck verwendet, müssen diese untereinander mit Kommas getrennt werden.	<b>i = 1,</b> <b>sum = 0</b>
13/14	Laufindex	Da in Zeile 14 auf die Array-Elemente direkt mittels <b>a[ i ]</b> zugegriffen wird, muss der Laufindex <b>i</b> von 0 bis <b>N-1</b> laufen.	<b>i = 0</b> <b>i &lt; N</b>
14	{ } statt [ ]	Beim Zugriff auf die einzelnen Elemente eines Arrays müssen eckige Klammern verwendet werden.	<b>a[ i ]</b>
15	N, zu viel	Der Parameter <b>N</b> ist zu viel.	<b>", sum );</b>

### Programm mit Korrekturen:

```

1  #include <stdio.h>
2
3  #define N      10
4
5  int main( int argc, char **argv )
6  {
7      int i, sum, a[ N ];
8      for( i = 0; i < N; i = i + 1 )
9      {
10         printf( "Bitte Wert fuer a[ %d ] eingeben: ", i );
11         scanf( "%d", & (a[ i ] ) );
12     }
13     for( i = 0, sum = 0; i < N; i = i + 1 )
14         sum = sum + a[ i ];
15     printf("Die Summe der %d Zahlen betraegt %d\n",N,sum);
16 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Deklaration einfacher Arrays

Die folgende Übungsaufgabe ist nicht wirklich sinnvoll, sondern dient hauptsächlich dem Einüben des Hantierens mit eindimensionalen Arrays.

### 1. Aufgabenstellung

Ziel der Aufgabe ist es, ein Programm zu erstellen, das folgende Arrays enthält:

1. ein Array mit 10 Elementen vom Typ `int`,
2. ein Array mit 3 Elementen vom Typ `char` und
3. ein Array mit 100 Elementen vom Typ `double`.

Des Weiteren sollen die Elemente dieser drei Arrays initialisiert werden. Dies geschieht vorzugsweise in einer Schleife. Diese Schleife soll so aufgebaut sein, dass eine Größenänderung des Arrays keine weiteren Änderungen in den Initialisierungsschleifen nach sich ziehen sollte; mit anderen Worten: Eine Größenänderung sollte durch eine *einzig*e Modifikation vollständig realisiert sein.

### 2. Kodierung

```
1  #include <stdio.h>
2
3  #define A_SIZE    10
4  #define B_SIZE    3
5  #define C_SIZE   100
6
7  int main( int argc, char **argv )
8  {
9      int i, a[ A_SIZE ];
10     char b[ B_SIZE ];
11     double c[ C_SIZE ];
12     for( i = 0; i < A_SIZE; i = i + 1 )
13         a[ i ] = 0;
14     for( i = 0; i < B_SIZE; i = i + 1 )
15         b[ i ] = 'A';
16     for( i = 0; i < C_SIZE; i = i + 1 )
17         c[ i ] = 0.0;
18 }
```

## Aufgabe 2: Implizite Größendefinitionen

Wir haben gelernt, dass man in der Programmiersprache C die Größe eines Arrays auch indirekt und zwar durch die initialisierte Definition festlegen kann. Gegenstand dieser Aufgabe ist das aktive Einüben dieses Konzeptes.

### 1. Aufgabenstellung

Entwickle ein kleines C-Programm, in dem die Größen von mindestens vier Arrays implizit durch ihre Initialisierung definiert werden. Zwei Beispielformulierungen sind bereits weiter unten angegeben.

Zur weiteren Verwendung der Größen, beispielsweise in Initialisierungsschleifen, ist die Größe jedes Arrays durch ein geeignetes `#define`-Makro zu „berechnen“.

Zur Bestätigung, dass alles korrekt abgelaufen ist, sind die einzelnen Elemente mittels geeigneter Schleifen auszugeben. In diesen Schleifen sind natürlich oben erwähnte `#define`-Makros zu verwenden.

### 2. Pflichtenheft

Aufgabe	: Korrekte Definition von Arrays mittels Initialisierung
Eingabe	: Eingaben werden nicht benötigt
Ausgabe	: Größe und Inhalt der Arrays als „Gegenprobe“
Sonderfälle	: Da es keine Eingaben gibt, können auch keine Problemfälle auftreten

### 3. Testdaten

Array	Typ	Initialwerte	C-Definitionen
a	int	1, 2, 3, 4	<pre>int    a[] = {1, 2, 3, 4 } #define A_SIZE sizeof( a )/sizeof(a[ 0 ])</pre>
x	double	1.0, 2.0	<pre>double x[] = { 1.0, 2.0 } #define X_SIZE sizeof( x )/sizeof(x[ 0 ])</pre>
c	char	'H', 'i'	<pre>char   c[] = { 'H', 'i' } #define C_SIZE sizeof( c )/sizeof(c[ 0 ])</pre>
m	char	98, 99, 100	<pre>char   m[] = { 98, 99, 100 } #define M_SIZE sizeof( m )/sizeof(m[ 0 ])</pre>



#### 4. Kodierung

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int a[] = { 1, 2, 3, 4 };
6      #define A_SIZE    sizeof(a)/sizeof(a[0])
7
8      double x[] = { 1.0, 2.0 };
9      #define X_SIZE    sizeof(x)/sizeof(x[0])
10
11     char c[] = { 'H', 'i' };
12     #define C_SIZE    sizeof(c)/sizeof(c[0])
13
14     char m[] = { 98, 99, 100 };
15     #define M_SIZE    sizeof(m)/sizeof(m[0])
16
17     int i;
18     printf( "a[ %d ]:", A_SIZE );
19     for( i = 0; i < A_SIZE; i = i + 1 )
20         printf( " %d", a[ i ] );
21     printf( "\n" );
22     printf( "x[ %d ]:", X_SIZE );
23     for( i = 0; i < X_SIZE; i = i + 1 )
24         printf( " %e", x[ i ] );
25     printf( "\n" );
26     printf( "c[ %d ]:", C_SIZE );
27     for( i = 0; i < C_SIZE; i = i + 1 )
28         printf( " %c", c[ i ] );
29     printf( "\n" );
30     printf( "m[ %d ]:", M_SIZE );
31     for( i = 0; i < M_SIZE; i = i + 1 )
32         printf( " %c", m[ i ] );
33     printf( "\n" );
34 }
```

# Übungspaket 15

## Einfaches Sortieren, Suchen und Finden

---

### Übungsziele:

1. Erarbeiten der Funktionsweise eines einfachen Sortieralgorithmus
2. Implementierung von Bubble Sorts
3. Suchen und Finden in sortierten und unsortierten Arrays

### Skript:

Kapitel: 27 und 33

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Sortieren, Suchen und Finden gehören zu den Standardaufgaben in der Informatik, die auch in vielen Anwendungen für uns Elektrotechniker relevant sind. Beim Sortieren von Daten gehen wir im Rahmen dieses Übungspaketes davon aus, dass bereits bei der Programmerstellung fest steht, wie groß das Datenarray sein soll. Für das Sortieren verwenden wir den einfachsten Algorithmus, der auch unter dem Namen **Bubble Sort** bekannt ist. Um ein Element zu suchen und nach Möglichkeit auch zu finden, verwenden wir ebenfalls einen sehr einfachen Algorithmus: das lineare Durchsuchen eines Arrays. Diese Themen werden wir wieder in Übungspaket 21 aufgreifen.

Da wir die „Zutaten“ für diese Algorithmen schon hatten, werden wir in den ersten drei Abschnitten (Stoffwiederholung, Quiz und Fehlersuche) keinen neuen Stoff einüben, sondern bereits Gelerntes nochmals kurz wiederholen. Wer hier auf Schwierigkeiten stößt, sollte sich zunächst noch einmal die entsprechenden Übungspakete anschauen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Definition eindimensionaler Felder

Zunächst ein paar allgemeine Fragen bzw. Teilaufgaben zu Arrays. Zeige anhand zweier Beispiele, wie man eindimensionale Felder definiert.

- `int feld[ 10 ]; // 10 elemente vom typ int`
- `double coeff[ 4 ]; // 4 elemente vom typ double`

Welche Indizes haben die Elemente eines Arrays mit 12 Elementen? `0...11`

Nehmen wir mal an, wir wollen im weiteren Verlauf unseres Programms auf die einzelnen Elemente eines zuvor definierten Feldes zugreifen. Hierfür müssen wir vor allem die Größe des Feldes wissen. Wie führen wir günstigerweise die Definition durch, damit das Programm möglichst änderungsfreundlich ist? Zeige dies anhand eines Beispiels einer Definition nebst einer Initialisierung:

```
1 #define SIZE      22                      // size definition
2
3 int i, array[ SIZE ];                      // our array
4
5 for( i = 0; i < SIZE; i = i + 1 )         // the initialization
6     array[ i ] = 0;
```

## Aufgabe 2: Ein-/Ausgabe von Arrays

Da es in der Programmiersprache C keine besonderen Formatierungen für Arrays gibt, müssen diese immer elementweise ein- bzw. ausgelesen werden. Zeige dies anhand eines Arrays der Größe SIZE:

**Einlesen eines Arrays:**

```
10 for( i = 0; i < SIZE; i = i + 1 )        // reading it in
11     scanf( "%d", & array[ i ] );
```

**Ausgeben eines Arrays:**

```
20 for( i = 0; i < SIZE; i = i + 1 )        // printing it
21     printf( "%d ", array[ i ] );
```

## Aufgabe 3: Arbeiten mit Array-Elementen

Nehmen wir wieder an, wir haben ein Array `feld` vom Typ `int`, das 20 Elemente haben soll. Wie sieht die Definition aus?

```
#define SIZE    20
int feld[ SIZE ];
```

Beantworte die folgenden Fragen anhand eines frei gewählten Beispiels.

Wie vergleicht man zwei *benachbarte* Elemente? `if ( feld[ i ] < feld[ i + 1 ] )`

Wie vertauscht man sie? `h = feld[i]; feld[i] = feld[i + 1]; feld[i + 1] = h;`

Nehmen wir an, wir führen obige Vergleiche etc. mit folgender Schleife durch:

```
1 for( i = 0; ..... ; i = i + 1 )      // die Schleife
2     if ( ..... )                    // der Vergleich
3     { ..... }                        // die Anweisungen
```

Wie muss obige Schleifenbedingung richtig lauten? `i < SIZE - 1`

Begründe kurz deine Antwort

Da wir bei obigem Vergleich die Elemente `i` und `i+1` miteinander vergleichen, darf der Laufindex `i` maximal den Wert `SIZE - 2` annehmen. Andernfalls würde die Array-Grenze überschritten werden, was nicht erlaubt ist. Zusammenfassend würden wir schreiben:

```
for( i = 0; i < SIZE - 1; i = i + 1 )
```

Zum Abschluss formulieren wir die Frage noch einmal anders: Nehmen wir an, wir haben ein Array der Größe `SIZE`. Nehmen wir ferner an, wir vergleichen nacheinander alle benachbarten Elemente `i` und `i+1`.

Wie viele Vergleiche werden durchgeführt? `Anzahl = SIZE - 1`

## Teil II: Quiz

---

### Aufgabe 1: Schleifen und Arrays

Nehmen wir an, wir haben ein Array und vergleichen benachbarte Elemente miteinander, wie wir es gerade eben in Aufgabe 3 gemacht haben. Nehmen wir ferner an, dass wir das Array *aufsteigend* sortieren wollen. „Aufsteigend“ heißt hier, dass sich an der Stelle 0 das kleinste Element und an der Stelle `SIZE - 1` das größte Element befindet. Dafür verwenden wir folgende Definitionen und Anweisungen (rudimentär, ohne richtiges Programm):

```
1 #define SIZE      4
2 int i, h, a[ SIZE ];
3
4 for( i = 0; i < SIZE - 1; i = i + 1 )
5     if ( a[ i ] > a[ i + 1 ] )
6     {
7         h = a[ i ]; a[ i ] = a[ i + 1 ]; a[ i + 1 ] = h;
8     }
```

Für die folgende Handsimulation nehmen wir an, dass das Array wie folgend belegt ist:

Elemente	a[ 0 ]	a[ 1 ]	a[ 2 ]	a[ 3 ]
Werte	4711	27	42	2

Führe in der folgenden Handsimulation *schrittweise* alle Vergleiche und ggf. die resultierenden Tausch-Operationen durch.

i	Vergleich Indizes	Vergleich Werte	initial:	a[ 0 ]	a[ 1 ]	a[ 2 ]	a[ 3 ]
				4711	27	42	2
0	0 mit 1	4711 mit 27		27	4711		
1	1 mit 2	4711 mit 42		27	42	4711	
2	2 mit 3	4711 mit 2		27	42	2	4711

Diese Handsimulation war für genau einen Durchgang durch das Array. Dennoch sollten wir noch einmal genau hinschauen, was mit den beiden Extremwerten passiert ist:

Wo ist das kleinste Element hingekommen?	<input type="text" value="Eine Position nach links"/>
Wo ist das größte Element hingekommen?	<input type="text" value="Ganz nach rechts"/>
Nächster Durchlauf: i von/bis:	<input type="text" value="i von 0 bis 1"/>
Noch ein Durchlauf: i von/bis:	<input type="text" value="i von 0 bis 0"/>
Wie viele Durchläufe bei <code>SIZE</code> Elementen?	<input type="text" value="SIZE - 1"/>

## Teil III: Fehlersuche

---

### Aufgabe 1: Finden und Zählen von Anstiegen

Das folgende Programm besitzt ein Array, in dem einige Werte vorliegen. Das Programm soll feststellen, wann ein Wert  $a_{i+1}$  größer als  $a_i$  ist. Ferner soll das Programm ausgeben, wie viele dieser positiven Vergleiche es gefunden hat.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, cnt;
6      int a[] = { 1 2 4 0 -2 33 4711 815 };
7      #define SIZE      (sizeof(a)/sizeof(a[0]))
8
9      for( i = 0, i < SIZE, i = i + 1 )
10         if ( a[ i ] < a[ i + 1 ] )
11             cnt = cnt + 1;
12             printf( "%5d --> %5d\n", a[ i ], a[ i + 1 ] );
13
14     printf( "%d mal steigend\n", cnt );
15     return 0;
16 }
```

Zeile	Fehler	Erläuterung	Korrektur
6	Kommas fehlen	Bei der Initialisierung eines Arrays müssen die einzelnen Werte mittels Kommas getrennt werden.	1, 2, 4, ...
9	, statt ;	Die einzelnen Teile der <b>for</b> -Schleife müssen mittels Semikolons voneinander getrennt werden.	<b>for( ; ; )</b>
9/10	Schleifenbedingung	Die Schleife wird bis einschließlich $i == \text{SIZE} - 1$ hochgezählt. Da aber in Zeile 10 der Vergleich mit $a[i + 1]$ durchgeführt wird, führt dies zu einem Fehler, ggf. zu einem Programmabsturz.	$i < \text{SIZE} - 1$
11	cnt nicht initialisiert	Da die Variable <b>cnt</b> nirgends initialisiert wurde, kann sie jeden beliebigen Wert annehmen.	<b>cnt = 0</b> in Zeile 5 oder 9
11/12	{ } fehlen	Gemäß Aufgabenstellung und Formatierung sollen beiden Zeilen bei positiver Fallunterscheidung ausgeführt werden. Da dort aber nur eine Anweisung stehen darf, müssen sie geklammert werden.	{ } vor Zeile 11 und nach Zeile 12

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, cnt;
6      int a[] = { 1, 2, 4, 0, -2, 33, 4711, 815 };
7      #define SIZE      (sizeof(a)/sizeof(a[0]))
8
9      for( i = cnt = 0; i < SIZE - 1; i = i + 1 )
10         if ( a[ i ] < a[ i + 1 ] )
11         {
12             cnt = cnt + 1;
13             printf( "%5d --> %5d\n", a[ i ], a[ i + 1 ] );
14         }
15
16     printf( "%d mal steigend\n", cnt );
17     return 0;
18 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Sortieren mittels Bubble Sort

Ziel dieser Aufgabe ist die Entwicklung des wohl einfachsten Sortieralgorithmus, der auch unter dem Namen **Bubble Sort** bekannt ist. Wie bisher in diesem Übungspaket gehen wir davon aus, dass sich die zu sortierenden Daten in einem Array befinden und dass wir die Daten aufsteigend sortieren wollen; also das kleinste Element an die Stelle mit dem Index 0 und das größte Element an die Stelle mit dem Index **SIZE-1**.

### 1. Vorüberlegungen

**Bubble Sort** macht nichts anderes als wir uns im Quiz-Teil dieses Übungspakets erarbeitet haben: Er geht das zu sortierende Array elementweise durch, vergleicht zwei Nachbarn und vertauscht diese gegebenenfalls. Dadurch kommt das größte Element ans Ende. Dieser Vorgang wird dann insgesamt maximal **SIZE-1** mal wiederholt. Anschließend ist das Array definitiv sortiert. Um aus „Glauben“ Wissen zu machen, machen wir hier eben noch eine Handsimulation (verkürzt, jeweils am Ende eines vollständigen Durchlaufs):

Durchlauf	a[ 0 ]	a[ 1 ]	a[ 2 ]	a[ 3 ]
0	4711	27	42	2
1	27	42	2	4711
2	27	2	42	4711
3	2	27	42	4711

### 2. Aufgabenstellung

Entwickle ein Programm, das die Elemente eines eindimensionalen Arrays aufsteigend sortiert. Es kann davon ausgegangen werden, dass die Daten bereits im Array vorhanden sind und dass die Größe des Arrays bereits zum Zeitpunkt der Programm-erstellung feststeht. Zur Kontrolle soll das Array am Ende ausgegeben werden. In der ersten Implementierung können die einzelnen Durchläufe immer komplett durch das Array gehen, auch wenn diese im Laufe des Sortierens „hinten“ nichts bringen.

### 3. Pflichtenheft

Aufgabe	: Programm zum Sortieren eines Arrays
Eingabe	: keine, die Daten liegen bereits im Array
Ausgabe	: Die Elemente des Arrays; diese sind (hoffentlich) am Ende sortiert
Sonderfälle	: keine



#### 4. Testdaten

Als Testdaten können wir die Beispiele aus dem Quiz-Teil übernehmen.

#### 5. Implementierung

Nach all den Vorarbeiten ist die Programmentwicklung sehr einfach; sie kann im Grunde genommen vom Quiz-Teil abgeleitet werden:

Bubble Sort

```
Konstante: Integer: SIZE
Variablen: Integer: i, j, h
Variablen: Integer Array: a[ 0 .. SIZE-1 ]
für i = 0 bis SIZE - 2 Schrittweite 1
  wiederhole für j = 0 bis SIZE - 2 Schrittweite 1
    wiederhole wenn a[ j ] > a[ j + 1 ]
      dann h = a[ j ]
      a[ j ] = a[ j + 1 ]
      a[ j + 1 ] = h
```

#### 6. Kodierung

```
1  #include <stdio.h>
2
3  #define SIZE      4
4
5  int main( int argc, char **argv )
6  {
7      int i, j, h, a[ SIZE ] = { 4711, 27, 42, 2 };
8      printf( "Das Array:" );           // testausgabe
9      for( i = 0; i < SIZE; i = i + 1 )
10         printf( " %d", a[ i ] );
11     printf( " unsortiert\n" );
12     for( i = 0; i < SIZE-1; i = i+1 )    // bubble
13         for( j = 0; j < SIZE-1; j = j+1 ) // bubble
14             if ( a[ j ] > a[ j + 1 ] )    // sort
15                 {
16                     h = a[j]; a[j] = a[j + 1]; a[j + 1] = h;
17                 }
18     printf( "Das Array:" );           // testausgabe
19     for( i = 0; i < SIZE; i = i + 1 )
20         printf( " %d", a[ i ] );
21     printf( " sortiert\n" );
22     return 0;                          // done
23 }
```

## 7. Erweiterung I

Die erste Erweiterung besteht darin, dass das Programm die zu sortierenden Daten selbstständig einliest. Wie das geht, haben wir bereits besprochen.

## 8. Erweiterung II

Im Quizteil haben wir gesehen, dass wir die Zahl der Vergleiche in jedem Durchlauf um eins reduzieren können. Hierzu müssen wir die Laufvariablen der äußeren Schleife in der Bedingung der inneren Schleife berücksichtigen. In der Musterlösung geschieht dies in Zeile 13 durch die Variable `i`. Führe die entsprechenden Erweiterungen durch.

### Bubble Sort mit Erweiterungen:

```
1  #include <stdio.h>
2
3  #define SIZE      4
4
5  int main( int argc, char **argv )
6  {
7      int i, j, h, a[ SIZE ];
8
9      printf( "Bitte die Elemente eingeben\n" );
10     for( i = 0; i < SIZE; i = i + 1 )
11     {
12         printf( "a[%d]: ", i ); scanf( "%d",& a[ i ] );
13     }
14
15     printf( "Das Array:" );
16     for( i = 0; i < SIZE; i = i + 1 )
17         printf( " %d", a[ i ] );
18     printf( " unsortiert\n" );
19
20     for( i = 0; i < SIZE-1; i = i+1 )
21         for( j = 0; j < SIZE-1-i; j = j+1 )
22             if ( a[ j ] > a[ j + 1 ] )
23             {
24                 h = a[j]; a[j] = a[j + 1]; a[j + 1] = h;
25             }
26
27     printf( "Das Array:" );
28     for( i = 0; i < SIZE; i = i + 1 )
29         printf( " %d", a[ i ] );
30     printf( " sortiert\n" );
31     return 0;
32 }
```

## Aufgabe 2: Suchen in unsortierten Tabellen

Eine zweite Standardaufgabe besteht darin, dass ein Programmteil überprüft, ob ein bestimmter Wert in einem Array vorhanden ist oder nicht. Diese Aufgabe ist mit dem Nachschlagen einer Telefonnummer in einem (elektronischen) Telefonbuch vergleichbar. In dieser Übungsaufgabe wenden wir uns einem sehr einfachen Fall zu, um uns das Grundkonzept einmal zu erarbeiten.

### 1. Aufgabenstellung

Wir nehmen an, dass wir ein Array haben, das mit Daten gefüllt ist. Wir nehmen ferner an, dass dieses Array unsortiert ist (ja, wir könnten es mit unserem **Bubble Sort** Algorithmus mal eben sortieren). Entwickle nun ein Programmstück, das überprüft, ob ein angegebener Wert in diesem Array vorhanden ist oder nicht.

**Beispiel:** Wir nehmen an, wir haben folgende Daten: 4711, 27, 42 und 2

Eingabe: 27    Ausgabe: 27 ist vorhanden, Index=1

Eingabe: 25    Ausgabe: 25 ist nicht vorhanden

**Quizfrage:** Welche Anweisung bricht eine **for**-Schleife frühzeitig ab?

### 2. Pflichtenheft

Aufgabe	: Entwickle ein Programm, das überprüft, ob ein angegebener Werte in einem unsortierten Array vorhanden ist.
Eingabe	: ein Testwert
Ausgabe	: im Erfolgsfalle den gefundenen Array-Index, sonst eine Fehlermeldung.
Sonderfälle	: keine

### 3. Testdaten

Als Testdaten nehmen wir obige Werte zuzüglich weiterer beliebiger Ganzzahlen.

### 4. Implementierung

Nach unseren Vorarbeiten ist die Implementierung eigentlich wieder ganz einfach. Wir müssen nur das Array elementweise durchgehen und jedes Mal schauen, ob der Elementwert mit dem Testwert übereinstimmt. Sollten beide Werte übereinstimmen, können wir die <b>for</b> -Schleife abbrechen und haben den Index. Sollte der Testwert nicht im Array vorhanden sein, würde die <b>for</b> -Schleife bis zum Ende durchlaufen und der Index hätte einen Wert, der größer als die Zahl der Array-Einträge ist. Daran sehen wir dann, dass der Testwert nicht gefunden wurde. Das vorzeitige Verlassen einer <b>for</b> -Schleife ist sehr C-bezogen und entspricht nicht der Philosophie des Software Engineerings, nach der die Zahl der Iterationen einer <b>for</b> -Schleife fest steht. Daher präsentieren wir auf der nächsten Seite zwei Lösungen.
--

### C-orientierte Lösung:

Suchen eines Testwertes in einem Array

```
Variablen: Integer: i, testwert
           Array of Integer: a[ 0 .. SIZE-1 ]
für i = 0 bis SIZE-1 Schrittweite 1
wiederhole wenn testwert = a[ i ]
           dann verlasse die for-schleife

wenn i = SIZE
dann Ausgabe Text "Testwert nicht gefunden"
sonst Ausgabe Text "Testwert gefunden; Index=", i
```

**Anmerkung:** Im Software Engineering und vielen anderen Programmiersprachen ist der Wert der Laufvariablen *nach* Verlassen der *for*-Schleife zusätzlich *undefiniert* und *nicht* mehr verfügbar.

### Software-Engineering-konforme Lösung:

Suchen eines Testwertes in einem Array

```
Variablen: Integer: i, testwert, found
           Array of Integer: a[ 0 .. SIZE-1 ]

setze found = -1
setze i = 0
solange i < SIZE und found = -1
wiederhole wenn testwert = a[ i ]
           dann setze found = i
           sonst setze i = i + 1

wenn found = -1
dann Ausgabe Text "Testwert nicht gefunden"
sonst Ausgabe Text "Testwert gefunden; Index=", i
```

**Anmerkung:** Durch die spezielle Konstruktion der Schleife und die Verwendung einer weiteren Variablen *found* vermeiden wir in jedem Fall, dass auf nicht vorhandene Array-Elemente zugegriffen wird. Um es nochmals anders zu formulieren: in den meisten Programmiersprachen würde der Ausdruck `while( i < SIZE && a[ i ] != test)` in jedem Fall bis zum Schluss ausgewertet werden. Entsprechend würde in anderen Programmiersprachen in *jedem* Fall auch das Element `a[ SIZE ]` getestet werden, was einen unerlaubten Zugriff bedeuten würde. Dies würde in den meisten Fällen zu einem Programmabsturz führen. Auch wenn diese Software-Engineering Diskussion nicht auf die Programmiersprache C zutrifft, so wollten wir es dennoch der Vollständigkeit halber erwähnen ;-).

Die beiden konkreten Kodierungen zeigen wir auf der nächsten Seite.

## 5. Kodierung

### Suchen eines Testwertes in einem Array:

```
1  #include <stdio.h>
2
3  #define SIZE    4
4
5  int main( int argc, char **argv )
6  {
7      int i, test, a[ SIZE ] = { 4711, 27, 42, 2 };
8      printf( "Bitte Testwert eingeben: " );    // input
9      scanf( "%d", & test );
10     for( i = 0; i < SIZE; i = i + 1 )          // search
11         if ( test == a[ i ] )                  // found !
12             break;
13     if ( i != SIZE )                            // found ?
14         printf( "%d gefunden, Index=%d\n", test, i );
15     else printf( "%d nicht gefunden\n", test );
16     return 0;                                  // done
17 }
```

### Suchen eines Testwertes in einem Array (software-engineering-konform):

```
1  #include <stdio.h>
2
3  #define SIZE    4
4
5  int main( int argc, char **argv )
6  {
7      int i, found, test, a[SIZE] = { 4711, 27, 42, 2 };
8      printf( "Bitte Testwert eingeben: " );    // input
9      scanf( "%d", & test );
10     i = 0; found = -1;
11     while( i < SIZE && found == -1 )            // search
12         if ( a[ i ] == test )
13             found = i;                          // found !
14         else i = i + 1;
15     if ( found != -1 )                          // found ?
16         printf( "%d gefunden, Index=%d\n", test, i );
17     else printf( "%d nicht gefunden\n", test );
18     return 0;                                  // done
19 }
```

So, das war's erst einmal :-)

# Übungspaket 16

## Gemischte Datentypen

---

### Übungsziele:

1. Sicherer Umgang mit gemischten Ausdrücken
2. Herleiten der unterschiedlichen Datentypen in gemischten Ausdrücken
3. Kenntnis über die impliziten Durchführung von Typanpassungen

### Skript:

Kapitel: 21, 30 und 32, sowie insbesondere Kapitel 37 und die Präzedenztabelle

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Solange nur Operanden und Operatoren eines Typs in einem Ausdruck erscheinen, entstehen normalerweise keine Fragen. Dies ändert sich aber, wenn in einem Ausdruck Typen miteinander gemischt werden. Im Gegensatz zu anderen Programmiersprachen, wie beispielsweise Pascal, Modula-2 oder Ada, ist dies in C erlaubt, doch führt der C-Compiler von sich aus einige Typanpassungen durch. Dies führt bei vielen Programmierern häufig zu unangenehmen Überraschungen und fehlerhaften Programmen. Egal wie, Um derartige Probleme zu vermeiden, ist es für uns als Programmierer sehr wichtig, diese impliziten Typanpassungen zu verstehen; also nicht *raten* sondern *wissen*.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Welche Typen

Leider haben wir keine direkte Möglichkeit, zu „kontrollieren“, was für Typen der Compiler in gemischten Ausdrücken annimmt; wir müssen es schlichtweg wissen. Mit der Compiler-Funktion `sizeof()` haben wir aber eine einfache Möglichkeit, dies relativ gut herauszufinden. Beschreibe kurz mit eigenen Worten, was diese Funktion genau macht:

Die Compiler-Funktion `sizeof()` nimmt als Argument entweder einen Datentyp (per Name) oder einen Ausdruck; bei letzterem bestimmt der Compiler den resultierenden Datentyp selbst. Nach dem der Compiler den Datentyp festgestellt hat, ermittelt er, wie viele Bytes er hierfür verwenden würde und gibt diesen Wert als Ergebnis zurück.

Zu Erinnerung: Die Funktion `sizeof()` kann man sowohl mit Datentypen als Ausdrücken und Daten, die ja auch Ausdrücke sind, aufrufen. Gegeben seien folgende Definitionen:

```
double d; char c; int i;
```

Vervollständigt nachfolgende Tabelle, in dem ihr die Compiler-Funktion `sizeof()` mit den entsprechenden Argumenten aufruft und das Ergebnis ausgibt.

**Beispiel:** Die Anweisung `printf( "size= %d\n", sizeof( int ) )` gibt aus, wie viele Bytes der Compiler für den Datentyp `int` verwendet.

Typ/Ausdruck	char	c	int	i	double	d	
<code>sizeof(Typ/Ausdruck)</code>	1	1	4	4	8	8	(Größe in Bytes)

**Wichtiger Hinweis:** Obige Ergebnisse hängen immer vom verwendeten Rechner (CPU) und dem Compiler ab; der C-Standard *definiert explizit nicht* die Größe von Datentypen!

Vervollständige die beiden folgenden Tabellen, wobei obige Variablendefinitionen weiterhin gelten:

Typ/Ausdruck	4711	3*8 - 1	6 / 5	c	c + 1	c + c
<code>sizeof(Typ/Ausdruck)</code>	4	4	4	1	4	1
vermuteter Datentyp	int	int	int	char	int	char

Typ/Ausdruck	'\101'	c + i	3.5	3.5 + 1	3.5 + i	3.1 + c
<code>sizeof(Typ/Ausdruck)</code>	4	4	8	8	8	8
vermuteter Datentyp	int	int	double	double	double	double

## Teil II: Quiz

---

### Aufgabe 1: Wert und Typ gemischter Ausdrücke

Die folgende Tabelle präsentiert einige gemischte Ausdrücke. Gefragt ist jeweils nach dem resultierenden Wert und dem resultierenden Typ. Versucht die Lösungen erst selbst zu finden und dann ggf. mittels eines kleinen Testprogramms zu überprüfen. Im Zweifelsfalle ist es sicherlich sinnvoll, noch einmal das Kapitel [37](#) und dort insbesondere Abschnitt [37.4](#) zu konsultieren. Ihr könnt davon ausgehen, dass das ASCII-Zeichen 'a' den Wert 97 hat.

Ausdruck	Wert	Typ	Kommentar/Begründung
<code>1 + 3</code>	4	<code>int</code>	
<code>1 + 3.0</code>	4.0	<code>double</code>	
<code>1 + 'a'</code>	'b' bzw. 98	<code>int</code>	Zeichen werden intern als <code>int</code> abgelegt
<code>1.0 + 'a' - 'b'</code>	0.0	<code>double</code>	
<code>1 + 3 / 5</code>	1	<code>int</code>	3 / 5 ergibt 0 (Ganzzahldivision)
<code>1.0 + 3 / 5</code>	1.0	<code>double</code>	Zuerst wird 3 / 5 = 0 ausgewertet. Erst dann wird das Ergebnis in ein <code>double</code> umgewandelt
<code>1.0 + 3.0 / 5</code>	1.8	<code>double</code>	3.0 / 5 = 0.6 dann die Addition mit 1.0
<code>3 / 5 + 1.0</code>	1.0	<code>double</code>	Auch hier ergibt 3 / 5 = 0, erst dann die Konvertierung nach <code>double</code>
<code>(1.0 + 3) / 5</code>	0.8	<code>double</code>	Die Klammer 1.0 + 3 hat bereits den Datentyp <code>double</code>
<code>2 * ('a' - 'b')</code>	-2	<code>int</code>	
<code>2 * 'b' - 'a'</code>	'c' bzw. 99	<code>int</code>	
<code>2.0 * ('a' - 'b')</code>	-2.0	<code>double</code>	



# Teil III: Fehlersuche

---

## Aufgabe 1: Gemischte Berechnungen

Für einen Labortest werden Vielfache der rationalen Zahl  $7/11$  für die Werte  $x \in \{1.0, 1.1, 1.2, \dots, 2.0\}$  benötigt. Somit werden die Zahlen  $7/11 \times 1.0, 7/11 \times 1.1, \dots, 7/11 \times 2.0$  gebraucht. Diese Werte sollen berechnet und in einer Tabelle mit elf Einträgen abgespeichert werden. Hierfür hat unser Starprogrammierer DR. ZERO CHECKER folgendes Programm entworfen, wobei die letzten drei Programmzeilen nur zur Überprüfung der Tabelle dienen. Leider sind in diesem Programm fünf Fehler enthalten. Finde und korrigiere sie.

```
1  #include <stdio.h>
2
3  #define X_MIN    1.0
4  #define X_MAX    2.0
5  #Define X_DIV    0.1
6  define X_SIZ     ((X_MAX - X_MIN)/X_DIV + 1) // total is 11.0
7
8  int main( int argc, char **argv )
9  {
10     double table[ X_SIZ ];
11     int i;
12     for( i = 0; i < X_SIZ; i = i + 1 )
13         table[ i ] = 7 / 11 * (X_MIN + i * X_DIV);
14     for( i = 0; i < 11; i = i+1 ) // print table to check
15         printf( "%4.2f ", table[ i ] );
16     printf( "\n" );
17 }
```

Zeile	Fehler	Erläuterung	Korrektur
3/5	#define #Define	Alle Präprozessordirektiven müssen in Kleinbuchstaben geschrieben werden.	#define
6	# fehlt	Die Präprozessordirektiven fangen immer mit einem das Doppelkreuz # an.	#define
10/6	falsche Arraygröße	Die Zahl der Arrayelemente <i>muss</i> eine Ganzzahl (int) sein. Daher muss in Zeile 10 entweder direkt 11 stehen oder der Ausdruck in Zeile 6 muss durch eine explizite Typumwandlung (siehe auch Skriptkapitel 37.5) in ein int umgewandelt werden.	table[ 11 ] oder table [(int) X_SIZ]

Zeile	Fehler	Erläuterung	Korrektur
13	Rundungsfehler	Der Ausdruck 7/11 liefert leider immer 0, sodass die Tabelle nur Nullen enthält.	7.0/11.0

### Korrigiertes Programm:

```

1  #include <stdio.h>
2
3  #define X_MIN    1.0
4  #define X_MAX    2.0
5  #define X_DIV    0.1
6  #define X_SIZ    11
7
8  int main( int argc, char **argv )
9  {
10     double table[ X_SIZ ];
11     int i;
12     for( i = 0; i < X_SIZ; i = i + 1 )
13         table[ i ] = 7.0 / 11 * (X_MIN + i * X_DIV);
14     for( i = 0; i < 11; i = i+1 )    // print table to check
15         printf( "%4.2f ", table[ i ] );
16     printf( "\n" );
17 }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Konvertierung von double nach int

Wir benötigen ein Programm zur Verarbeitung von Messwerten, die alle vom Typ `double` sind. Die Messwerte haben vor dem Komma immer eine 0 und danach genau zwei Ziffern. Das Programm soll aus den beiden Nachkommastellen eine Nummer berechnen, die dem Ganzzahlwert der beiden Nachkommastellen entspricht. Zur Illustration folgen hier beispielhafte Ein- und Ausgaben:

Eingabe:	0.01	0.82	0.83	0.74	0.65	0.86	0.47	0.48	0.29	0.10
Ausgabe:	01	82	83	74	65	86	47	48	29	10

Hierfür haben wir folgendes Programm entworfen, wobei die Berechnung in Zeile 10 noch fehlt:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i, nummer;
6     double wert;
7     for( i = 0; i < 10; i = i + 1 )
8     {
9         scanf( "%lf", & wert );
10        nummer =          ; // hier wird berechnet
11        printf( "wert=%6.4f nummer=%2d\n", wert, nummer );
12    }
13 }
```

Wenn wir nun einfach `nummer = wert * 100` rechnen, kommt leider folgendes falsche Ergebnis heraus:

Eingabe:	0.01	0.82	0.83	0.74	0.65	0.86	0.47	0.48	0.29	0.10
Ausgabe:	01	81	82	73	65	85	46	47	28	10

Beantworte folgende Fragen:

Warum sind die Ergebnisse falsch?

Es entstehen Rundungsfehler
-----------------------------

Wie wäre die Berechnung korrekt?

<code>nummer = wert * 100 + 0.5</code>
--

Überprüfe deine Vermutung durch Abtippen und Testen obigen Programms :-)

## Aufgabe 2: Konvertierung von int nach double

Wir benötigen ein kleines Programm zum Erstellen folgender (sinnloser) Tabelle:

1	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
2	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
3	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9
4	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9
5	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9
6	5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9
7	6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9
8	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9
9	8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	8.9
10	9.0	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9

Hierfür haben wir folgendes Programm entworfen:

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i, j;
6      ..... // definition der variablen wert
7      for( i = 0; i < 10; i = i + 1 )
8      {
9          for( j = 0; j < 10; j = j + 1 )
10         {
11             wert = .....; // hier soll berechnet werden
12             printf( ".....", wert );
13         }
14         printf( "\n" );
15     }
16 }
```

In den Zeilen 6, 11 und 12 fehlen noch eine geeignete Variablendefinition, die eigentliche Berechnung und das Ausgabeformat. Ergänze diese fehlenden Stellen und überlege dir mindestens zwei verschiedene Berechnungsarten:

Zeile	Aktion	Lösung	Kommentar
6	Definition	<code>double wert;</code>	
11	Berechnung 1	<code>wert = i + j / 10.0;</code>	j/10.0 ergibt einen double-Wert
11	Berechnung 2	<code>wert = j;</code> <code>wert = wert/10.0 + i;</code>	wert = j ergibt einen double-Wert
12	Format	<code>%3.1f</code>	

# Übungspaket 17

## Der gcc Compiler

---

### Übungsziele:

1. Sicherer Umgang mit gemischten Ausdrücken
2. Herleiten der unterschiedlichen Datentypen in gemischten Ausdrücken
3. Kenntnis über die implizite Durchführung von Typanpassungen

### Skript:

Kapitel: 38, 39 und 40

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Eines haben wir alle bis jetzt sicherlich gelernt, durch Eintippen von `gcc quelltext.c` wird ein C-Programm in Maschinencode übersetzt und damit lauffähig gemacht. Doch wenn etwas schiefgeht, fangen die Probleme an. Der Compiler selbst ist wie die meisten Programmierwerkzeuge ein recht komplexes Programm. Daher fällt es vielen Programmieranfängern sehr schwer, den Überblick über die einzelnen Teile des Compilers zu behalten und die aufgetretenen Fehler*ursachen* zu lokalisieren. Um hier Abhilfe zu schaffen, schauen wir uns in diesem Übungspaket den Compiler `gcc` und seine Komponenten ein wenig genauer an.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Grobaufbau des gcc Compilers

Durch den Aufruf `gcc datei.c` werden eigentlich vier größere Programme *nacheinander* aufgerufen. Benenne diese vier Programme und erläutere kurz, was ihre Aufgaben sind.

<b>1. Programm:</b>	Präprozessor
<b>Funktion:</b>	<p>Der Präprozessor erfüllt im Wesentlichen drei Funktionen:</p> <ol style="list-style-type: none"><li>1. Der Präprozessor ersetzt all <b>#include</b>-Direktiven durch die angegebenen Dateien (diese Dateien werden <i>vollständig</i> in den Quelltext eingefügt).</li><li>2. Ersetzen aller <b>#define</b>-Makros durch ihre entsprechenden Definitionen.</li><li>3. Übersetzen bzw. entfernen aller Anweisungen zwischen den <b>#ifdef</b>, <b>#ifndef</b>, <b>#else</b> und <b>#endif</b> Direktiven.</li></ol>
<b>2. Programm:</b>	Eigentlicher Compiler (der C-Übersetzer)
<b>Funktion:</b>	<p>In dieser Phase wird das C-Programm in denjenigen Assembler-Code übersetzt, der zum gewählten Prozessor gehört. Das Ergebnis ist also eine Datei, die prozessorspezifisch ist.</p> <p><i>Ergebnis:</i> eine Datei mit der Endung <code>.s</code></p>
<b>3. Programm:</b>	Assembler
<b>Funktion:</b>	<p>Der Assembler wandelt den (prozessorspezifischen) Assembler-Code, der noch „lesbare“ Anweisungen enthält in Maschinencode um. Das Ergebnis ist eine Datei, die nur noch aus unverständlichen Nullen und Einsen besteht, die üblicherweise zu hexadezimalen Zahlen zusammengefasst werden.</p> <p><i>Ergebnis:</i> eine Datei mit der Endung <code>.o</code></p>
<b>4. Programm:</b>	Linker
<b>Funktion:</b>	<p>Der Linker fügt den Maschinencode und alle verwendeten Bibliotheken zu einem einzigen lauffähigen Programm zusammen. Erst dieses Programm kann vom Prozessor (in Zusammenarbeit mit dem Betriebssystem) auch wirklich ausgeführt werden.</p> <p><i>Ergebnis:</i> eine ausführbare Datei mit dem angegebenen Namen oder <code>a.out</code></p>

## Aufgabe 2: Die Syntax der Präprozessor-Direktiven

Erkläre kurz in eigenen Worten die Syntax der C-Präprozessor-Direktiven:

Alle C-Präprozessor-Direktiven fangen mit einem Doppelkreuz `#` an, werden von einem der Schlüsselwörter `include`, `define`, `ifdef`, `ifndef`, `else` oder `endif` gefolgt und meistens mit einem Argument (Parameter) abgeschlossen. Zwischen dem Zeilenanfang, dem Doppelkreuz `#`, dem Schlüsselwort und den Argumenten dürfen beliebig viele Leerzeichen, Tabulatoren und Kommentare eingefügt werden. Ferner ist zu beachten, dass eine C-Präprozessor-Direktive in einer Zeile abgeschlossen wird, es sei denn, die Zeile wird mit einem Backslash `'\'` abgeschlossen.

## Aufgabe 3: Die Präprozessor-Direktiven

Erkläre jede Präprozessor-Direktive anhand je eines oder zweier Beispiele:

### 1. `#include`

Mittels der Direktive `#include` kann man andere Dateien einbinden. Die `#include` wird dadurch vollständig durch den Inhalt der angegebenen Datei ersetzt.

```
#include <stdio.h>    // Einbinden der Standard Ein-/Ausgabe
#include "my.h"       // Einbinden der eigenen Datei my.h
```

### 2. `#define`

Mittels `#define` kann man einfache Labels und ganze Makros definieren:

```
#define NAME          Cool          // mein Name
#define HiThere( x )  Hi Daddy x    // meine Anrede
HiThere( NAME )

Ergibt: Hi Daddy Cool
```

### 3. `#ifdef ... #else ...#endif`

Die `#ifdef`-Direktive erlaubt es, Dinge in Abhängigkeit anderer zu übersetzen.

```
#ifdef __my_header
#else
#include "my-header.h"
#endif
```

In diesem Falle wird die Datei `my_header.h` nur dann eingebunden, sofern das Label `__my_header` noch nicht definiert wurde. Zwischen `#if`, `#else` und `#endif` können beliebige Anweisungen stehen. In den `.h`-Dateien werden oftmals Labels der obigen Form definiert, um das mehrmalige Ausführen eines `#include` zu vermeiden.

## Teil II: Quiz

---

### Aufgabe 1: Die Definition von „Namen“

Gegeben sei folgendes Programmstück:

```
1 #define Tag      Montag          /* mein Arbeitstag */
2 #define WOCHEN   34              /* meine Urlaubswoche */
3
4 #define Telefon  0381 498 72 51
5 #define FAX      /* 0049 */ 0381 498 118 72 51
6
7 #define Ferien
8
9 #define MSG      "heute geht es mir sehr gut"
10
11 #define NAME1    ein name
12 #define NAME2    NAME1
13
14 #define ZWEI_ZEILEN  ZEILE-1 /* das ist die erste Zeile
15 #define ZWEITE_ZEILE  hier ist Zeile 2 */
16
17 #define ENDE     "jetzt ist schluss"
```

Finde heraus, welche Namen in obigem Programmstück definiert werden und welche Werte diese haben. Trage die definierten Namen nebst ihrer Werte in folgende Tabelle ein.

Zeile	Name	Wert
1	Tag	Montag
2	WOCHEN	34
4	Telefon	0381 498 72 51
5	FAX	0381 498 118 72 51
7	Ferien	
9	MSG	"heute geht es mir sehr gut"
11	NAME1	ein name
12	NAME2	ein name
14	ZWEI_ZEILEN	ZEILE-1
15	-----	Hier wird kein Name definiert, denn diese Zeile ist auskommentiert
17	ENDE	"jetzt ist schluss"



## Aufgabe 2: Definition von Makros

Gegeben sei folgendes Programmstück:

```
1 // macro implementation of the formula: 2*x + 3*y
2
3 #define Formula_1( x, y )      2 * x + 3 * y
4 #define Formula_2( x, y )      2 * (x) + 3 * (y)
5
6 i = Formula_1( 1, 2 );          j = Formula_2( 1, 2 );
7 i = Formula_1( 4, 2 );          j = Formula_2( 4, 2 );
8 i = Formula_1( 1+2, 2+3 );      j = Formula_2( 1+2, 2+3 );
9 i = Formula_1( 2+1, 3+2 );      j = Formula_2( 2+1, 3+2 );
```

Notiert in der folgenden Tabelle die Werte der Parameter  $x$  und  $y$ , das erwartete Ergebnis  $res = 2 * x + 3 * y$ , sowie die Resultate, die in den Variablen  $i$  und  $j$  abgelegt werden:

Zeile	x	y	res	i	j	Zeile	x	y	res	i	j
6	1	2	8	8	8	8	3	5	21	13	21
7	4	2	14	14	14	9	3	5	21	16	21

Wie werden die Makros vom Präprozessor expandiert (ersetzt)?

```
6 i = 2 * 1 + 3 * 2;          j = 2 * (1) + 3 * (2);
7 i = 2 * 4 + 3 * 2;          j = 2 * (4) + 3 * (2);
8 i = 2 * 1+2 + 3 * 2+3;      j = 2 * (1+2) + 3 * (2+3);
9 i = 2 * 2+1 + 3 * 3+2;      j = 2 * (2+1) + 3 * (3+2);
```

Überprüft eure Annahme durch Aufruf des Präprozessors. Nehmen wir an, ihr habt die paar Zeilen abgetippt und in der Datei `quiz.c` gespeichert. Dann einfach `cpp quiz.c` oder alternativ `gcc -E quiz.c` aufrufen; die Ergebnisse erscheinen dann direkt auf dem Bildschirm. Überprüft nochmals eure Ergebnisse, wie ihr sie in obiger Tabelle eingetragen habt. Erklärt, sofern sich Unterschiede auftun, was hierfür die Ursachen sind. Welche Schlussfolgerungen zieht ihr daraus für die Definition von Makros?

**Ursachen:** In der Makro-Definition `Formula_1` werden die beiden Parameter  $x$  und  $y$  nicht geklammert. Dadurch kann es sein, dass bei Übergabe von einfachen arithmetischen Ausdrücken die einzelnen Bestandteile aufgrund der Präzedenzregeln nicht so ausgewertet werden, wie man es erwartet.

**Problembehebung:** Bei der Definition (der Implementierung) von Makros sollten die Parameter stets geklammert werden. So werden die Parameter *immer* zuerst ausgewertet und dann mit anderen Parametern verknüpft.

## Aufgabe 3: Einbinden von (Header-) Dateien

Gegeben sei folgendes Programmstück:

```
1 #include <stdio.h>
2 #include <math.h>
3
4 #define C_TYPES <ctype.h>
5 #include C_TYPES
6
7 // #include <signal.h>
```

Welche Dateien werden dadurch vom Präprozessor eingebunden?

Folgende Systemdateien (aus /usr/include) werden eingebunden:

stdio.h, math.h und ctype.h

Die Datei signal.h wird nicht eingebunden, da diese Zeile auskommentiert ist.

## Aufgabe 4: Bedingtes Übersetzen

Gegeben sei folgendes Programmstück:

```
1 #define A          4711
2
3 #ifndef A
4     #define N1      17
5     #ifndef B
6         #define N2    2
7     #else
8         #define N2    4
9     #endif
10 #else
11     #define N1      12
12     #define N2      -3
13 #endif
14
15 int i = N1 * N2;
```

Welche Labels werden mit welchen Werten definiert, welchen Wert erhält die Variable i?

Zeile	Label	Wert
1	A	4711
4	N1	17

Zeile	Label	Wert
8	N2	4
15	i	68

# Teil III: Fehlersuche

---

## Aufgabe 1: Praktische Fehlersuche

Das folgende Programm soll den Text **Fehler beheben** drei Mal ausgeben, jeweils eine Ausgabe pro Zeile. Diesmal war unser Starprogrammierer DR. BIT-BYTE zugange. Aber auch ihm sind einige Fehler unterlaufen. Finde und korrigiere diese. Zeige anschließend mittels einer Handsimulation, dass dein Programm korrekt arbeitet.

```
1 #define ANZAHL          3;
2 #define LOOP_CONDITION  (fehler >= 0)
3 #define MSG              "Fehler "eheben"
4
5 int main( int argc, char **argv );
6 {
7     int fehler = ANZAHL;
8     do
9         printf( MSG ); fehler = - 1;
10    while ( LOOP_CONDITION );
11    return 0;
12 }
```

Zeile	Fehler	Erläuterung	Korrektur
0	#include fehlt	Die Datei <code>stdio.h</code> wird nicht eingebunden. Wir brauchen sie aber, da wir etwas ausgeben wollen.	#include <stdio.h>
1		Das Semikolon scheint zu viel zu sein. Aber in der Initialisierung in Zeile 7 ist es dennoch ok (aber unschön).	3
2	>= statt >	Im Verbindung mit der <code>do-while</code> -Schleife in den Zeilen 8 bis 10 ergeben sich vier statt drei Schleifendurchläufe.	(fehler>0)
3	" statt b	Hier scheint ein einfacher Tippfehler vorzuliegen.	beheben
5	; zu viel	Bei der Funktions <i>definition</i> darf am Ende <i>kein</i> ; stehen. Dieses ist nur bei Funktionsdeklarationen erlaubt, die dem Compiler den Namen und die Signatur der Funktion bekannt machen.	argv )
8/10	{ } fehlen	Da die <code>do-while</code> -Schleife mehr als eine Anweisung ausführen soll, müssen sie durch { } geklammert werden.	{... }
9	fehler fehlt	Da der Wert der Variablen <code>fehler</code> um eins reduziert werden soll, muss sie auch auf der rechten Seite auftauchen.	fehler = fehler - 1

### Programm mit Korrekturen:

```
1 #include <stdio.h>
2
3 #define ANZAHL          3
4 #define LOOP_CONDITION  (fehler > 0)
5 #define MSG             "Fehler beheben\n"
6
7 int main( int argc, char **argv )
8 {
9     int fehler = ANZAHL;
10    do {
11        printf( MSG );
12        fehler = fehler - 1;
13    } while ( LOOP_CONDITION );
14    return 0;
15 }
```

Vor der Handsimulation ist es eine gute Übung, sich das Programm zu überlegen, das der Präprozessor aus der .c-Datei generiert:

### Ergebnis nach Abarbeitung durch den Präprozessor:

```
7 int main( int argc, char **argv )
8 {
9     int fehler = 3;
10    do {
11        printf( "Fehler beheben\n" );
12        fehler = fehler - 1;
13    } while ( (fehler > 0) );
14    return 0;
15 }
```

### Handsimulation:

Zeile	9	11	12	13	11	12	13	11	12	13	14
fehler	3		2	2		1	1		0	0	
printf()		Fehler beheben		Fehler beheben			Fehler beheben				

## Teil IV: Anwendungen

---

### Aufgabe 1: Fehler Finden und Eliminieren

Es ist ganz normal, dass man beim Entwickeln und Eintippen eines Programms viele Tippfehler macht. Da der Compiler ein recht pingeliger Zeitgenosse ist, findet er viele dieser Tippfehler und gibt entsprechende Fehlermeldungen und Hinweise aus. Die Kunst besteht nun darin, dieser Ausgaben richtig zu deuten *und* die wirklichen Fehlerursachen zu finden. Um dies ein wenig zu üben, greifen wir nochmals das fehlerhafte Programm aus dem vorherigen Abschnitt (vorherige Seite) auf, wobei wir davon ausgehen, dass ihr sowieso alle Fehler gefunden habt. Arbeite nun wie folgt:

#### Arbeitsanleitung:

1. Tippe das *fehlerhafte* Programm ab und speichere es in einer Datei deiner Wahl.
2. Übersetze das fehlerhafte Programm mittels `gcc`.
3. Fahre mit Arbeitsschritt **6** fort, falls der Compiler keine Fehlermeldung oder Warnung ausgegeben hat.
4. Lese die erste(n) Fehlermeldung(en) aufmerksam durch und korrigiere *einen* Fehler.
5. Gehe zurück zu Arbeitsschritt **2**.
6. Starte das Programm und überprüfe, ob es korrekt arbeitet.
7. Sollte das Programm korrekt arbeiten gehe zu Schritt **10**
8. Korrigiere einen inhaltlichen Fehler (Semantikfehler).
9. Gehe zurück zu Arbeitsschritt **2**.
10. Fertig!

**Hinweis:** Diese Herangehensweise empfiehlt sich auch in Zukunft :-) !

## Aufgabe 2: Eigene Makro-Definitionen

Definiere je ein Makro für die folgenden drei Formeln (Lösungen siehe unten):

$$\begin{aligned}f(x) &= 3x^2 + x/2 - 1 \\g(x, y) &= x^2 - 3xy + y^2 \\h(x, y, z) &= 2x^3 - 3y^2 + 2z\end{aligned}$$

Zu welchen Ergebnissen führen die folgenden drei Aufrufe (Einsetzen)?

Aufruf	Resultat Mathematik	Resultat C-Programm
$f(1 + z)$	$3(1 + z)^2 + (1 + z)/2 - 1$	<code>3*(1 + z)*(1 + z) + (1 + z)/2 - 1</code>
$g(x, A + 1)$	$x^2 - 3x(A + 1) + (A + 1)^2$	<code>x*x - 3*x*(A + 1) + (A + 1)*(A + 1)</code>
$h(a, b, a + b)$	$2a^3 - 3b^2 + 2(a + b)$	<code>2*a*a*a - 3*b*b + 2*(a + b)</code>

Überprüft eure Makro-Definitionen durch Eintippen eines kleinen Programmstücks und Aufruf des Präprozessors (entweder mittels `cpp <datei>.c` oder `gcc -E <datei>.c`).

### Programmstückchen:

```
1 #define f(x)          3*(x)*(x) + (x)/2 - 1
2 #define g(x, y)       (x)*(x) - 3*(x)*(y) + (y)*(y)
3 #define h(x, y, z)     2*(x)*(x)*(x) - 3*(y)*(y) + 2*(z)
4
5 f(1 + z)
6 g(x, A + 1)
7 h(a, b, a + b)
```

### Ausgabe des Präprozessors:

```
1 3*(1 + z)*(1 + z) + (1 + z)/2 - 1
2 (x)*(x) - 3*(x)*(A + 1) + (A + 1)*(A + 1)
3 2*(a)*(a)*(a) - 3*(b)*(b) + 2*(a + b)
```

# Übungspaket 18

## Ausdrücke

---

### Übungsziele:

1. Ausdrücke in vielfältiger Form
2. Formulierung vereinfachender Kurzformen
3. Pre-/Post-Dekrement/Inkrement
4. Bedingte Auswertung
5. Besonderheiten logischer Ausdrücke

### Skript:

Kapitel: 43

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In Kapitel 43 haben wir gelernt, dass Ausdrücke in ganz unterschiedlichen Formen auftreten können. Die Programmiersprache C bietet verschiedene Kurzschreibweisen an, mittels derer sich Ausdrücke sehr kompakt formulieren lassen. Dies führt bei vielen Programmierern, insbesondere bei Anfängern, zumindest anfänglich zu Unverständnis, Frust und Ärger. Um diese Dinge zu vermeiden, schauen wir uns in diesem Übungspaket die verschiedenen Kurzformen und Besonderheiten systematisch an.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Kurzformen für Zuweisungen

Wie kann eine Zuweisung vereinfacht werden, wenn die selbe Variable sowohl auf der rechten als auch der linken Seite auftaucht?

Vereinfachung bei Verwendung des Operators op: `var = var op exp ⇒ var op= exp`

Notiere drei unterschiedliche Beispiele zur Illustration:

1. `i = i + 1 ⇒ i += 1`    2. `j = j * 3 ⇒ j *= 3`    3. `d=d/2.5 ⇒ d /= 2.5`

## Aufgabe 2: Die Pre- und Post-Operatoren

Zuerst sollten wir uns nochmals die hier relevanten Begriffe klar machen.

### 1. Pre-Inkrement:

Syntax	<code>++variable</code>
Beispiel	<code>++i</code>
Aktualisierungszeitpunkt	<i>vor</i> dem Variablenzugriff

### 2. Pre-Dekrement:

Syntax	<code>--variable</code>
Beispiel	<code>--i</code>
Aktualisierungszeitpunkt	<i>vor</i> dem Variablenzugriff

### 3. Post-Inkrement:

Syntax	<code>variable++</code>
Beispiel	<code>i++</code>
Aktualisierungszeitpunkt	<i>nach</i> dem Variablenzugriff

### 4. Post-Dekrement:

Syntax	<code>variable--</code>
Beispiel	<code>i--</code>
Aktualisierungszeitpunkt	<i>nach</i> dem Variablenzugriff

## Aufgabe 3: Bedingte Ausdrücke (Zuweisungen)

Zur Erinnerung: Hin und wieder will man einer Variablen einen Wert zuweisen, der wiederum von einer Bedingung abhängig ist. Beispiel: `if (condition) i = 1; else i = 2;` Je



nachdem, ob die Bedingung `condition` erfüllt ist oder nicht, wird der Variablen `i` der Wert 1 oder 2 zugewiesen. Etwas allgemeiner spricht man auch von bedingten Ausdrücken. Illustriere die Syntax der bedingten Zuweisung bzw. des bedingten Ausdrucks anhand zweier Beispiele:

Syntax	<code>condition? true_value: false_value</code>
Beispiel 1	<code>i = (j &gt; 3)? 1: -1</code>
Beispiel 2	<code>printf( "expression= %d\n", (j &gt; 3)? 1: -1 )</code>

## Aufgabe 4: Logische Ausdrücke

**Logische Werte:** Wie sind die beiden folgenden logischen Werte in der Programmiersprache C definiert?

wahr/true	alle Werte ungleich null
falsch/false	Wert gleich null

**Auswertung (Berechnung) logischer Ausdrücke:** Die Programmiersprache C wertet arithmetische und logische Ausdrücke grundsätzlich unterschiedlich aus. Bei arithmetischen Ausdrücken gilt, dass die Reihenfolge, in der die einzelnen Terme ausgewertet (berechnet) werden, dem Compiler überlassen ist. Der Compiler garantiert *nur*, dass er die Einzelteile gemäß der gültigen Rechenregeln korrekt zusammensetzt. Welche beiden Regeln gelten für die Auswertung logischer Ausdrücke?

Regel 1:	Logische Ausdrücke werden von links nach rechts ausgewertet.
Regel 2:	Das Auswerten wird abgebrochen, sobald das Ergebnis feststeht.

## Teil II: Quiz

---

### Aufgabe 1: Kurzformen für Zuweisungen

Gegeben sei folgendes Programm:

```
1 int main( int argc, char **argv )
2     {
3         int i = 1, j = 2, k = 3;
4         i += k; j -= k; k *= 3;
5         i *= j + k; j *= 100; j /= k; k -= j - i;
6         i -= (k = 4); j += 1;
7     }
```

Notiere in der folgenden Tabelle, welche Werte die einzelnen Variablen in den jeweiligen Programmzeilen annehmen:

Zeile	Werte von		
	i	j	k
3	1	2	3
4	4	-1	9
5	32	-11	52
6	28	-10	4

### Aufgabe 2: Bedingte Ausdrücke (Zuweisungen)

In den folgenden Anweisungen sind die beiden Variablen `i` und `j` vom Typ `int` und haben *immer* die Werte: `i = 1` und `j = 2`. Welche Werte haben sie anschließend?

Ausdruck	i	j
<code>i = (j == 2)? 1: 3;</code>	1	2
<code>i = (2)? 1: 3;</code>	1	2
<code>i = (0)? 1: 3;</code>	3	2
<code>i = (j = 1)? 5: 7;</code>	5	1
<code>i = (j = 1)? (j = 2) + 1: (j = 3) - 6;</code>	3	2
<code>i = (j++ &gt; 1)? 2: 4;</code>	2	3

### Aufgabe 3: Die Pre- und Post-Operatoren

Gegeben sei folgendes Programm:

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i = 1, j = 2, k = 3;
6      printf( "i= %d j= %d k= %d\n", i++, ++j, k-- );
7      printf( "i= %d j= %d k= %d\n", i++, ++j, k-- );
8      printf( "i= %d j= %d k= %d\n", i++, ++j, k-- );
9      printf( "i= %d j= %d k= %d\n", i, j, k );
10 }

```

Notiere in der folgenden Tabelle die Ausgaben für die einzelnen Variablen:

Zeile	Ausgabe von		
	i	j	k
6	1	3	3
7	2	4	2

Zeile	Ausgabe von		
	i	j	k
8	3	5	1
9	4	5	0

Im Folgenden werden die beiden Pre- und Post-Operatoren innerhalb etwas komplexerer Ausdrücke verwendet. Dabei sind die drei Variablen *i*, *j* und *k* vom Typ *int*, wobei am Anfang jeder Anweisung *immer* gilt: *i* = 1 und *j* = 2. Welche Ergebnisse liefern die folgenden Anweisungen?

Ausdruck	i	j	k	Bemerkung
<i>k</i> = <i>i</i> ++	2	2	1	
<i>k</i> = <i>i</i> --	0	2	1	
<i>k</i> = ++ <i>i</i>	2	2	2	
<i>k</i> = -- <i>i</i>	0	2	0	
<i>k</i> = <i>i</i> -- * ++ <i>j</i>	0	3	3	
<i>k</i> = (1 + ++ <i>i</i> )* <i>j</i> ++	2	3	6	
<i>k</i> = (1 + ++ <i>i</i> )* <i>i</i>	2	2	?	<i>k</i> ist undefiniert, da die Evaluationsreihenfolge von ++ <i>i</i> und <i>i</i> undefiniert ist
<i>k</i> = (1 + ++ <i>i</i> )* <i>i</i> --	1	2	?	<i>k</i> ist undefiniert, da die Evaluationsreihenfolge von ++ <i>i</i> und <i>i</i> -- undefiniert ist

**Hinweis:** Bei der Beurteilung bzw. Auswertung obiger Ausdrücke sollte insbesondere Abschnitt 43.4 des Skriptes zu Rate gezogen werden.

## Aufgabe 4: Logische Ausdrücke

In den folgenden Aufgaben sind die beiden Variablen `i` und `j` vom Typ `int` und haben zu Beginn jeder Anweisung immer die Werte: `i = 1` und `j = 2`. Welche Ergebnisse liefern die folgenden Ausdrücke?

Ausdruck	Ergebnis	i	j
<code>i == 1</code>	1	1	2
<code>i &gt; -2</code>	1	1	2
<code>i == 1 &amp;&amp; j != 0</code>	1	1	2
<code>i &amp;&amp; j</code>	1	1	2
<code>i++ &amp;&amp; j++</code>	1	2	3
<code>i--    j++</code>	1	0	2
<code>i++ &gt; 2 &amp;&amp; j++</code>	0	2	2

## Aufgabe 5: Verschachtelte Zuweisungen

In den folgenden Beispielen sind alle Variablen vom Typ `int`. Welche Werte haben die Variablen nach den Zuweisungen?

Ausdruck	i	j
<code>i = (j = 5)</code>	5	5
<code>i = (j = 5) &lt; 10</code>	1	5
<code>i = j = 5 &lt; 10</code>	1	1
<code>i = (j = 5)*(i = 2)</code>	10	5
<code>i = ((j = 5) &gt; 3) * 2</code>	2	5

## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler in Kurzformen

Beim folgenden (sinnlosen) Programm hat DR. CHATTY wieder Einiges falsch gemacht. Finde und korrigiere die vorhandenen Syntaxfehler. Manchmal gibt es mehrere Korrekturmöglichkeiten. Entscheide dich für eine; Hauptsache jede Zeile wird korrekt.

```
1 int main( int argc, char **argv )
2     {
3         int i = 1--, j = -2;
4         i *= += 1;
5         i *= += j;
6         i *** 3;
7         j = ++i++;
8         j = 3 *= i;
9         j -= 3 * (i += 2) ,
10        i++ = 3 * j;
11        i-- += 3;
12    }
```

Zeile	Fehler	Erläuterung	Korrektur
3	1--	Die Pre- und Post-Increment/Decrement Operatoren kann man nur auf Variablen und nicht auf Konstanten anwenden; es wäre sinnlos, die 1 auf null zu reduzieren.	i = 1
4	*= bzw. +=	Bei einer Zuweisung kann man nur <i>einen</i> dieser Kurzoperatoren und nicht mehrere gleichzeitig anwenden. Wir entscheiden uns für den zweiten.	i += 1
5	dito.	Da rechts aber eine Variable steht, könnte man die Ausdrücke schachteln, denn der Wert einer Zuweisung ist das Ergebnis. Das ++ bezieht sich auf j, i wird um den neuen Wert von j erhöht. Einfach mal probieren.	i *= (++j)
6	***	Bei der Kurzfassung kommt erst das „Rechensymbol“ und dann das Gleichheitszeichen.	*=
7	++i++	Bei <i>einer</i> Variablen ist <i>entweder</i> eine Pre- <i>oder</i> eine Post-Operation anwendbar. Wir nehmen letzteres.	i++
8	*=	Vermutlich ist einfach nur das = zu viel.	j = 3 * i
9	, statt ;	Eigentlich ist alles richtig! Hinten fehlt aber ein ;	;

Zeile	Fehler	Erläuterung	Korrektur
10	++ und =	Es geht nicht beides gleichzeitig.	<code>i = 3*j+1</code>
11	-- +=	dito.	<code>i += 3 - 1</code>

### Programm mit Korrekturen:

```

1  int main( int argc, char **argv )
2      {
3          int i = 1, j = -2;
4          i += 1;
5          i *= ++ j;
6          i *= 3;
7          j = i++;          // oder j = ++i;
8          j = 3 * i;
9          j -= 3 * (i += 2);
10         i = 3 * j + 1;
11         i += 3 - 1;
12     }

```

## Teil IV: Anwendungen

### Aufgabe 1: Kurzformen

Vereinfache die folgenden Ausdrücke soweit möglich. Beachte, dass es manchmal unterschiedliche Lösungen gibt. Bei Unsicherheiten oder Zweifeln die Ergebnisse einfach mittels eines Mini-C-Programms überprüfen!

Ausdruck	Kurzform	Alternativen
$k = k + 1$	$k += 1$	$k -= -1$ , $k++$ , $++k$
$k = k * 2$	$k *= 2$	$k += k$
$k = k - 4711$	$k -= 4711$	$k += -4711$
$k = k - 23 - i$	$k -= 23 + i$	$k += -23 - i$
$k = 2 * k - i - 1$	$k *= 2$ ; $k -= i + 1$	$k += k - i - 1$ oder $k -= -k + i + 1$
$d = (123.0 + 0.123) * d$	$d *= 123.0 + 0.123$	$d *= 123.123$
$i = i / 123$	$i /= 123$	$i *= 1/123$ funktioniert nicht, da dies identisch mit: $i *= 0$ ist

### Aufgabe 2: Vereinfachung logischer Ausdrücke

Mit etwas Übung lassen sich logische Ausdrücke ein klein wenig vereinfachen. Dies üben wir anhand von vier unterschiedlichen Aufgaben. Fülle zunächst die Wahrheitstabelle aus und überlege dir dann eine mögliche Vereinfachung. In den folgenden Tabellen steht **a** immer für eine Bedingung, die die Werte **wahr** (1) und **falsch** (0) annehmen kann.

a „und“ wahr:	a „und“ falsch:	a „oder“ wahr:	a „oder“ falsch:																								
<table><tr><td>a</td><td>a &amp;&amp; 1</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	a	a && 1	0	0	1	1	<table><tr><td>a</td><td>a &amp;&amp; 0</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td></tr></table>	a	a && 0	0	0	1	0	<table><tr><td>a</td><td>a    1</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td></tr></table>	a	a    1	0	1	1	1	<table><tr><td>a</td><td>a    0</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	a	a    0	0	0	1	1
a	a && 1																										
0	0																										
1	1																										
a	a && 0																										
0	0																										
1	0																										
a	a    1																										
0	1																										
1	1																										
a	a    0																										
0	0																										
1	1																										
Vereinfachung <input type="checkbox"/> a	Vereinfachung <input type="checkbox"/> 0	Vereinfachung <input type="checkbox"/> 1	Vereinfachung <input type="checkbox"/> a																								

#### Zusammenfassung:

logisches und:	a „und“ wahr ergibt a, a „und“ falsch ergibt falsch.
logisches oder:	a „oder“ wahr ergibt wahr, a „oder“ falsch ergibt a.

# Übungspaket 19

## Programmieren eigener Funktionen

---

### Übungsziele:

1. Implementierung und Kodierung eigener Funktionen
2. Rekapitulation des Stack-Frames
3. Parameterübergabe mittels Stack und Stack-Frame

### Skript:

Kapitel: 44

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Funktionen sind eine schöne Möglichkeit, ein Programm zu strukturieren und übersichtlich zu gestalten. Doch bereitet dieses Konzept vielen Programmieranfängern eine Reihe von Problemen, was eigentlich völlig unnötig ist. Indem wir erste kleinere Funktionen selber schreiben, schauen wir uns die wesentlichen Konzepte im Detail an.



# Teil I: Stoffwiederholung

---

## Aufgabe 1: Deklaration und Definition

Für den sicheren Umgang mit Funktionen ist es wichtig, dass man die einzelnen Bestandteile einer Funktion gut kennt und sie voneinander abgrenzen kann. Gemäß Kapitel 44 können wir folgende Komponenten voneinander unterscheiden. Beschreibe in eigenen Worten, was jeder der folgenden Begriffe bedeutet und illustriere deine Erläuterungen jeweils anhand eines Minibeispiels.

### 1. Funktionskopf:

Beschreibung:	Der Funktionskopf besteht aus: Typ des Rückgabewertes, Namen der Funktion, der Liste der Formalparameter in runden Klammern. Name und Parameterliste werden als Signatur bezeichnet.
Beispiel:	<pre>int fnc( int a, char b ) int dcopy( double * p1, double * p2 )</pre>

### 2. Parameterliste:

Beschreibung:	Die formalen Parameter können innerhalb der Funktion wie lokale Variablen verwendet werden. Der <i>Aufrufer muss</i> beim Aufruf jedem <i>formalen</i> Parameter einen <i>aktuellen</i> Wert geben (call by value).
Beispiel:	<pre>fnc( 4, 'a' ) dcopy( &amp; d[ 3 ], d + 6 )</pre>

### 3. Funktionsrumpf:

Beschreibung:	Der Funktionsrumpf ist die algorithmische Beschreibung der Funktionalität der Funktion, diese ist in geschweifte Klammern eingeschlossen und ist Bestandteil der Funktionsdefinition.
Beispiel:	<pre>{ return c - 'a' + 'A'; } { printf( "Hi, guys\n"); return 1; }</pre>

### 4. Funktionsdeklaration:

Beschreibung:	Die Funktions <i>deklaration</i> ist wie der Funktions <i>kopf</i> , nur dass ihr noch ein Semikolon folgt. Durch sie wird dem Compiler bekannt gemacht, dass es irgendwo (egal wo) genau so eine Funktion gibt.
Beispiel:	<pre>int fnc( int a, char b ); int dcopy( double * p1, double * p2 );</pre>

## 5. Funktionsdefinition:

Beschreibung: Sie besteht aus dem Funktionskopf direkt gefolgt vom dazugehörenden Funktionsrumpf. Durch diese Kombination wird die Funktion auch tatsächlich realisiert.

Beispiel:

```
int fnc( int a, char c )
{ return a*(c - 'a' + 'A'); }
```

## Aufgabe 2: Stack-Frame

Gegeben sei die Funktionsdefinition

```
1 int mul_add( int a, int b, int c )
2 {
3     int res;
4     res = a * b + c;
5     return res;
6 }
```

sowie der Funktionsaufruf `i = mul_add( 10, 5, -15 )`. Erkläre anhand dieses Beispiels, was ein Stack-Frame ist und wie mit seiner Hilfe Parameter sowie Ergebnisse zwischen der aufrufenden Stelle und der Funktion übergeben werden.

Der Stack-Frame ist ein Speichersegment, das bei *jedem* Funktionsaufruf erneut auf dem Stack angelegt wird. Im Stack-Frame wird für folgende Daten Platz reserviert: die formalen Parameter, die die Werte der aktuellen Parameter zugewiesen bekommen, Platz für den Rückgabewert der Funktion, den aktuellen Wert des Programmzählers (PC) sowie die benötigten lokalen Variablen.

Durch den Funktionsaufruf `i = mul_add( 10, 5, -15 )` sieht der Stack-Frame in etwa wie folgt aus:

Variable	Wert
int c:	-15
int b:	5
int a:	10
int return:	35
CPU PC:	<alter PC>
int res:	35

In diesem Speicherbildchen ist der Rückgabewert bereits im Stack-Frame abgelegt. Dies bedeutet, dass die Funktion unmittelbar vor ihrem Ende steht.

## Aufgabe 3: Funktionsaufruf

Erkläre mit eigenen Worten, was bei einem Funktionsaufruf *der Reihe nach* passiert. Nimm dazu als Beispiel obigen Funktionsaufruf `i = mul_add( 10, 5, -15 )`.

### 1. Funktionsaufruf:

Beim Funktionsaufruf passiert der Reihe nach folgendes:

1. Der Stack-Frame wird angelegt.
2. Die aktuellen Parameter werden evaluiert (ausgerechnet) und ihre Werte den formalen Parametern zugewiesen (an die Speicherplätze der entsprechenden formalen Parameter kopiert). Im betrachteten Fall sind dies: `a = 10`, `b = 5` und `c = -15`.
3. Der aktuelle Wert des Programmzählers (PCs) wird auf dem Stack gerettet. Er dient am Ende des Funktionsaufrufs als Rücksprungsadresse.

### 2. Funktionsabarbeitung:

In diesem Schritt wird die Funktion Anweisung für Anweisung abgearbeitet. Im betrachteten Fall wird `res = 35` ausgerechnet und der `return`-Anweisung übergeben.

### 3. Funktionsende (Rücksprung):

Am Ende eines Funktionsaufrufs werden folgende Punkte abgearbeitet:

1. Bei Erreichen der `return`-Anweisung wird der zurückzugebende Wert auf den entsprechenden Speicherplatz im Stack-Frame kopiert.
2. Das Programm „springt“ an die zuvor gespeicherte Stelle (den alte Wert des Programmzählers) zurück.
3. Der Rückgabewert wird eventuell noch zur späteren Verwendung zwischengespeichert. Im betrachteten Fall wird der zurückgegebene Wert `35` der Variablen `i` zugewiesen.
4. Der Stack-Frame wird abgebaut.

## Aufgabe 4: Funktionsabarbeitung

Gegeben sei folgendes Programm:

```
1  #include <stdio.h>
2
3  int f( int a, int b )
4  {
5      int i;
6      i = a - b;
7      return 2 * i;
8  }
9
10 int main( int argc, char **argv )
11 {
12     int i = 4711, j;
13     j = f( i, 4700 );
14     printf( "j= %d\n", j );
15 }
```

Zeichne den zur Funktion `f()` gehörenden Stack-Frame zu den angegebenen „Zeitpunkten“:

1. Stack-Frame, wie er für `f()` vom Compiler definiert wird:

Variable	Wert
int b:	
int a:	
int return:	
CPU PC:	
int i:	

2. Stack zu Beginn des Funktionsaufrufs (Funktionsaufruf in Zeile 13):

Adresse	Variable	Wert	Änderung
0xFFEE1080	.....	.....	
0xFFEE107C	int i:	4711	
0xFFEE1078	int j:		
0xFFEE1074	int b:	4700	←
0xFFEE1070	int a:	4711	←
0xFFEE106C	int return:		
0xFFEE1068	CPU PC:	„Zeile 13“	←
0xFFEE1064	int i:		

3. Stack während des Funktionsaufrufs (Ende Zeile 6):

Adresse	Variable	Wert	Änderung
0xFFEE1080	.....		
0xFFEE107C	int i:	4711	
0xFFEE1078	int j:		
0xFFEE1074	int b:	4700	
0xFFEE1070	int a:	4711	
0xFFEE106C	int return:		
0xFFEE1068	CPU PC:	„Zeile 13“	
0xFFEE1064	int i:	11	←

4. Stack am Ende des Funktionsaufrufs (Ende Zeile 7):

Adresse	Variable	Wert	Änderung
0xFFEE1080	.....		
0xFFEE107C	int i:	4711	
0xFFEE1078	int j:		
0xFFEE1074	int b:	4700	
0xFFEE1070	int a:	4711	
0xFFEE106C	int return:	22	←
0xFFEE1068	CPU PC:	„Zeile 13“	
0xFFEE1064	int i:	11	

5. Stack nach Ende des Funktionsaufrufs (Anfang Zeile 14):

Adresse	Variable	Wert	Änderung
0xFFEE1080	.....		
0xFFEE107C	int i:	4711	
0xFFEE1078	int j:	22	←

Der Stack-Frame ist nicht mehr vorhanden, da er nach dem Funktionsende wieder aus dem Arbeitsspeicher entfernt wurde.

6. Was wird im Hauptprogramm ausgegeben? j= 22

## Teil II: Quiz

---

### Aufgabe 1: Programmanalyse

Gegeben sei das folgende kleine Programm:

```
1  #include <stdio.h>
2
3  int max_dif( int a, int b, int c, int d )
4  {
5      int d1 = a - b;
6      int d2 = c - d;
7      return (d1 > d2)? d1: d2;
8  }
9
10 int main( int argc, char **argv )
11 {
12     printf( "max_dif= %d\n", max_dif( 2, 1, 9, 8 ) );
13     printf( "max_dif= %d\n", max_dif( 2, 0, 8, 9 ) );
14     printf( "max_dif= %d\n", max_dif( 1, 3, -1, -1 ) );
15     printf( "max_dif= %d\n", max_dif( 10, -13, -2, -10 ) );
16 }
```

Erkläre in eigenen Worten, was die Funktion `max_dif()` macht:

Die Funktion `max_dif()` bekommt als Argumente zwei Paare von Parametern. Von jedem Paar bestimmt die Funktion die Differenz der beiden Zahlenwerte und gibt als Ergebnis die größere der beiden Differenzen zurück.

Erkläre im Detail, was die Anweisung und der Ausdruck in Zeile 7 macht und wie man sie mittels `if else` formulieren würde:

Der Operator `?:` besteht aus drei Ausdrücken: einer Bedingung, einem Ausdruck für den Fall, dass die Bedingung wahr ist, und einem Ausdruck für den Fall, dass die Bedingung falsch ist. Mit `if else` könnte man Zeile 7 folgendermaßen formulieren:

```
if (d1 > d2) return d1; else return d2;
```

Welche vier Zahlen werden in den Zeilen 12-15 ausgegeben? 1 2 0 23

Wie sieht der Stack-Frame aus, der durch den Funktionsaufruf in Zeile 14 angelegt wird?  
Wie sieht er am Ende von Zeile 7 unmittelbar vor dem Ende der Funktion aus?

Begin des Funktionsaufrufs:		Ende des Funktionsaufrufs:	
Variable	Wert	Variable	Wert
int d:	-1	int d:	-1
int c:	-1	int c:	-1
int b:	3	int b:	3
int a:	1	int a:	1
int return:		int return:	0
CPU PC:	„Zeile 14“	CPU PC:	„Zeile 14“
int d1:		int d1:	-2
int d2:		int d2:	0

## Aufgabe 2: Verschachtelte Funktionsaufrufe

Gegeben sei folgendes Programm:

```

1  #include <stdio.h>
2
3  int f( int i )
4      {
5          int j = i * 2 + 1;
6          printf( "    f: i= %3d j= %3d\n", i, j );
7          return j - i - 2;
8      }
9
10 int g( int i )
11     {
12         int j = i * 3 - 1;
13         printf( "    g: i= %3d j= %3d\n", i, j );
14         return j - 2 * (i - 3);
15     }
16
17 int main( int argc, char **argv )
18     {
19         int i = 1, j = 2;
20         printf( "main: i= %3d j= %3d\n", i, j );
21         i = f(i - g(i - j - 10)) + f(j + g(i + j));
22         j = g(i - f(i - j - 10)) - g(j + f(i + j));
23         printf( "main: i= %3d j= %3d\n", i, j );
24     }

```

Welche Ausgaben werden vom Programm gemacht? Gehe davon aus, dass bei der Addition der linke Operand vor dem rechten Operanden ausgewertet wird.

```

main: i=  1 j=  2
      g: i= -11 j= -34
      f: i=  7 j= 15
      g: i=  3 j=  8
      f: i= 10 j= 21
      f: i=  3 j=  7
      g: i= 13 j= 38
      f: i= 17 j= 35
      g: i= 18 j= 53
main: i= 15 j= -5

```

Zeichne die Stack-Frames der einzelnen Funktionen, wie sie unmittelbar vor Ausführung der `printf()`-Anweisung aussehen, wenn der Ausdruck aus Zeile 21 ausgewertet wird (vier Stack-Frames); trage auch die sich ergebenden `return`-Werte ein.

**Funktionsaufruf: g( -11 ):**

Variable	Wert
int i:	-11
int return:	-6
CPU PC:	„Zeile 21“
int j:	-34

**Funktionsaufruf g( 3 ):**

Variable	Wert
int i:	3
int return:	8
CPU PC:	„Zeile 21“
int j:	8

**Funktionsaufruf f(1-g(-11))=f(7):**

Variable	Wert
int i:	7
int return:	6
CPU PC:	„Zeile 21“
int j:	15

**Funktionsaufruf f(2+g(3))=f(10):**

Variable	Wert
int i:	10
int return:	9
CPU PC:	„Zeile 21“
int j:	21



## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler bei der Verwendung von Funktionen

In folgendem Programm befinden sich wieder einige Fehler. Finde und korrigiere sie. Die Funktion `hallo()` soll zwei Parameter vom Typ `int` bekommen und die Differenz zurückgeben, sofern der erste Parameter größer als der zweite ist; andernfalls soll die Summe beider Parameter zurückgegeben werden. Sollten Parameter fehlen, einfach welche ausdenken.

```
1  #include <stdio.h>
2
3  int hallo( int i, j );
4      [
5          if ( i > j )
6              return i - j;
7          else return;
8      )
9  int main( int argc, char **argv )
10     {
11         printf( "hallo= %d\n", hallo ( 1; 2 ) );
12         printf( "hallo= %d\n", hallo [ 2, 1 ] );
13         printf( "hallo= %d\n", hallo { hallo( 1, 1 ) }, 4711 );
14         printf( "hallo= %d\n", hallo ( 2 ) + 2 );
15     }
```

Zeile	Fehler	Erläuterung	Korrektur
3	Datentyp für j fehlt	Jeder formale Parameter muss einen Datentyp haben.	<code>int j</code>
3	Semikolon zu viel.	Bei der Definition darf dort kein Semikolon kommen.	<code>int hallo( ... )</code>
4	[ statt {.	Der Funktionsblock <i>muss</i> immer in {}-Klammern eingeschlossen sein.	<code>{</code>
7	Rückgabewert fehlt		<code>return i + j</code>
11	; statt ,	Die Parameter müssen durch Kommas getrennt werden.	<code>hallo( 1, 2 )</code>
12	[] statt ()	Die aktuellen Parameter müssen in runden Klammern stehen.	<code>hallo( 1, 2 )</code>
13	{ } statt ()	dito.	<code>hallo( 1, 2 )</code>
14	Parameter fehlt	Die Zahl der aktuellen und formalen Parameter müssen übereinstimmen.	<code>hallo( 3, 2 )</code>

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2
3  int hallo( int i, int j )
4  {
5      if ( i > j )
6          return i - j;
7      else return i + j;
8  }
9  int main( int argc, char **argv )
10 {
11     printf( "hallo= %d\n", hallo ( 1, 2 ) );
12     printf( "hallo= %d\n", hallo ( 2, 1 ) );
13     printf( "hallo= %d\n", hallo ( hallo( 1, 1 ), 4711 ) );
14     printf( "hallo= %d\n", hallo ( 3, 2 ) + 2 );
15 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Programmieren einer einfachen Funktion

### 1. Aufgabenstellung

Programmiere eine Funktion `int my_mult()`, die zwei Parameter vom Typ `int` hat und deren Produkt als Funktionswert zurückgibt. Wie immer bei solchen Aufgaben durchlaufen wir den regulären Software Life Cycle.

**Beispiel:** Aufruf: `res = my_mult( 4, 5)` Ausgabe: `res= 20`

### 2. Pflichtenheft

Aufgabe	: Multiplikation zweier Parameter mittels einer Funktion
Eingabe	: keine, die Parameter werden direkt kodiert
Entwurf	: Die Funktionalität soll als Funktion realisiert werden
Ausgabe	: Produkt aus beiden Parametern
Sonderfälle	: keine

### 3. Testdaten

Parameter 1:	3	0	2	-2	-100
Parameter 2:	2	4	0	1	-2
Ergebnis:	6	0	0	-2	200

### 4. Implementierung

Funktion <code>my_mult</code>
Parameter: Integer: <code>a, b</code>
Rückgabewert <code>a × b</code>

### 5. Kodierung

<pre>1 int my_mult( int a, int b ) 2     { 3         int res = a * b; 4         return res; 5         // return a * b would be much easier 6         // but we want to learn things 7     }</pre>
---

Ein vollständiges Testprogramm befindet sich am Ende dieses Übungspaketes.

## 6. Stack Frame

Zeichne den Stack-Frame der Funktion `my_mult( 4, 5 )`, wie er vor dem Funktionsaufruf, nach der Parameterübergabe (Beginn der Funktion) und bei Ausführung der `return`-Anweisung (Ende der Funktion) aussieht.

Vorher:		Zu Beginn:		Am Ende:	
Variable	Wert	Variable	Wert	Variable	Wert
int b:		int b:	5	int b:	5
int a:		int a:	4	int a:	4
int return:		int return:		int return:	20
CPU PC:		CPU PC:		CPU PC:	
int res:		int res:		int res:	20

## Aufgabe 2: Programmieren einer weiteren Funktion

### 1. Aufgabenstellung

Implementiere eine Funktion `int test_div()`, die zwei Parameter vom Typ `int` hat und überprüft, ob sich diese beiden Zahlen ohne Divisionsrest dividieren lassen. Das Ergebnis ist als Funktionswert zurückzugeben. Eine Division durch Null muss nicht überprüft werden (kann aber ;-)).

**Beispiele:**

Aufruf: `test_div( 4, 2 )`  $\Rightarrow$  1  
`test_div( 5, 3 )`  $\Rightarrow$  0

### 2. Pflichtenheft

Aufgabe : Test auf Teilbarkeit; Divisionsrest muss null sein  
 Eingabe : keine, die Parameter werden direkt kodiert  
 Ausgabe : Ergebnis des Divisionstests  
 Sonderfälle: gemäß Aufgabenstellung brauchen wir eine Division durch Null nicht überprüfen

### 3. Testdaten

Parameter 1:	4	5	4	-4	-8
Parameter 2:	2	3	-2	2	-4
Ergebnis:	1	0	1	1	1

### 4. Implementierung

Funktion `test_div`

Parameter: Integer: `a`, `b`

wenn Divisionsrest von `a / b` gleich Null

dann Rückgabewert 1

sonst Rückgabewert 0

## 5. Kodierung

```
1 int test_div( int a, int b )
2     {
3         return a % b == 0;
4     }
```

Ein vollständiges Testprogramm befindet sich am Ende dieses Übungspaketes.

## 6. Stack Frame

Zeichne den Stack-Frame der Funktion `test_div( 5, 4 )`, wie er vor dem Funktionsaufruf, nach der Parameterübergabe (Beginn der Funktion) und bei Ausführung der `return`-Anweisung (Ende der Funktion) aussieht.

Vorher:		Zu Beginn:		Am Ende:	
Variable	Wert	Variable	Wert	Variable	Wert
int b:		int b:	4	int b:	4
int a:		int a:	5	int a:	5
int return:		int return:		int return:	0
CPU PC:		CPU PC:		CPU PC:	

# Aufgabe 3: Funktion mit einem Array als Parameter

Im Ausblick von Kapitel 44 haben wir schon gesehen, wie man ein Array an eine Funktion übergeben kann. Entsprechend üben wir dies einfach ein wenig ein.

## 1. Aufgabenstellung

Definiere ein Array mit 15 Elementen vom Typ `int`. Die Elemente sollen als Wert `array[ index ] = 40 - 2 * index` erhalten. Implementiere eine Funktion, die ein beliebiges Array mit Elementen vom Typ `int` in der Form „Index, Wert“ ausgibt. Rufe diese Funktion mit dem oben angelegten Array auf und vergleiche, ob die Ausgabe mit der Initialisierung übereinstimmt.

## 2. Pflichtenheft

Aufgabe	: Anlegen, Initialisieren und Ausgeben eines Arrays
Eingabe	: keine, da alles vorgegeben
Ausgabe	: Tabelle mit den Array-Elementen
Sonderfälle	: keine

## 3. Testdaten

Keine notwendig, da das Array nur wie vorgegeben initialisiert und ausgegeben wird.

## 4. Implementierung

Array anlegen und initialisieren
Konstante: SIZE of Value 15
Variable: Array 0 .. SIZE-1 of Integer: vector
Variable: Integer: index
für index = 0 to SIZE-1 Schrittweite 1
wiederhole vector[ index ] = 40 - 2 * index
Funktion Array ausgeben
Parameter: Array of Integer: vector
Parameter: Integer: size
für index = 0 to size-1 Schrittweite 1
wiederhole ausgabe index, vector[ index ]

## 5. Kodierung

Array anlegen und initialisieren:

1	#define SIZE 15
2	int i, vector[ SIZE ];
3	for( i = 0; i < SIZE; i++ )
4	vector[ i ]= 40 - 2 * i;

Array ausgeben:

1	int print_array(int vector[], int size)
2	{
3	int i; // loop-index
4	for( i = 0; i < size; i++ )
5	printf( "Index=%2d, Wert=%2d\n", i, vector[i] );
6	return 1;
7	}

Ein vollständiges Testprogramm befindet sich am Ende dieses Übungspaketes.

## 6. Stack Frame

Zeichne den Stack-Frame der Funktion `print_array` am Ende ihrer Abarbeitung. Berücksichtige dabei die konkreten Adressen des eigentlichen Arrays (aktueller Parameter) und der Werte der formalen Parameter.

Adresse	Variable	Wert
0xFFEE10A8	.....	
0xFFEE10A4	int vector[ 14 ]:	12
.....	.....	.....
0xFFEE1070	int vector[ 0 ]:	40
0xFFEE106C	int size:	15
0xFFEE1068	int * vector:	0xFFEE1070
0xFFEE1064	int return:	1
0xFFEE1060	CPU PC:	<alter PC>
0xFFEE105C	int i:	15

## Aufgabe 4: Einfache Statistikaufgaben

Mit ein paar grundlegenden Programmierkenntnissen kann man in einfacher Weise statistische Parameter von Messdaten bestimmen. Stellen wir uns vor, wir haben  $n$  Messwerte  $x_0, \dots, x_{n-1}$ . Dann ist der Mittelwert  $\bar{x}$  definiert als:  $\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i$ . Hinzu kommt noch die Standardabweichung  $\sigma$ , die wie folgt definiert ist:

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2} \quad .$$

### 1. Aufgabenstellung

Entwickle ein Programm, dass sowohl den Mittelwert als auch die Standardabweichung einer gegebenen Messreihe bestimmt. Dabei *können* folgende Vorgaben berücksichtigt werden:

1. Die Berechnung von Mittelwert und Standardabweichung sollte nach Möglichkeit innerhalb einer eigenen Funktion geschehen.
2. Die Daten können bereits in einem Array vorliegen. Um das Programm ausreichend testen zu können, sollten gleich mehrere Arrays mit unterschiedlichen Größen verwendet werden. Der Elementtyp kann in allen Fällen `int` sein.
3. Entwickle eine weitere Funktion, die das kleinste und größte Element des übergebenen Arrays findet.

## 2. Wichtiger Hinweis

Bei Verwendung mathematischer Funktionen wie `sqrt()` muss im Programm die Datei `math.h` mittels `#include <math.h>` eingebunden werden, um dem Compiler diese und andere Funktionen *bekannt* zu machen. Ferner muss dem Compiler zusätzlich die Option `-lm` übergeben werden (einfach ans Ende des Kommandos anhängen), damit er die mathematische Bibliothek dazu bindet.

## 3. Pflichtenheft

Aufgabe	: Entwicklung einer Funktion zur Berechnung von Mittelwert und Standardabweichung von $n$ Messwerten vom Typ <code>int</code> . Ferner soll eine Funktion entwickelt werden, die sowohl das kleinste als auch größte Element dieser Messreihe bestimmt. Die Formeln stehen in der Aufgabenstellung.
Eingabe	: keine, die Daten liegen im Programm vor.
Ausgabe	: Mittelwert, Standardabweichung, kleinstes und größtes Element
Testfälle	: Das fertige Programm ist an mindestens drei Messreihen zu testen.
Sonderfälle	: keine.

## 4. Testdaten

Daten	$\bar{x}$	$\sigma$	Min.	Max.
1, 2, 3	2.000e+00	1.000e+00	1.000e+00	3.000e+00
-1, -1, -1, -1, -1	-1.000e+00	0.000e+00	-1.000e+00	-1.000e+00
1, 1, 2, 2	1.500e+00	5.774e-01	1.000e+00	2.000e+00

## 5. Entwurf

Für jede der einzelnen Aufgaben wie Mittelwert, Standardabweichung, kleinstes und größtes Element implementieren wir jeweils eine separate Funktion. Hierdurch werden die Funktionen sehr klein und damit übersichtlich.

Alle Funktionen sind vom Typ `double` und bekommen die folgenden beiden Parameter `int * data` und `int size` übergeben. Die Funktion zur Berechnung der Standardabweichung bekommt zusätzlich noch den (zuvor berechneten) Mittelwert vom Typ `double` übergeben.

Die einzelnen Formeln können wir direkt aus der Aufgabenstellung übernehmen.

## 6. Implementierung

Da die einzelnen Algorithmen mittlerweile sehr einfach sind, ersparen wir uns hier eine abstrakte algorithmische Beschreibung.



## 7. Kodierung

Unsere Lösung sieht wie folgt aus:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double minimum( int * data, int size )
5  {
6      int i, min = data[ 0 ];
7      for( i = 1; i < size; i++ )
8          if ( data[ i ] < min )
9              min = data[ i ];
10     return min;
11 }
12
13 double maximum( int * data, int size )
14 {
15     int i, max = data[ 0 ];
16     for( i = 1; i < size; i++ )
17         if ( data[ i ] > max )
18             max = data[ i ];
19     return max;
20 }
21
22 double average( int * data, int size )
23 {
24     int i;
25     double avg;
26     for( i = 0, avg = 0.0; i < size; i++ )
27         avg += data[ i ];
28     return avg / size;
29 }
30
31 double std_deviation( int * data, int size, double avg )
32 {
33     int i;
34     double dif;
35     for( i = 0, dif = 0.0; i < size; i++ )
36         dif += (data[ i ] - avg)*(data[ i ] - avg);
37     return sqrt(dif / (size - 1));
38 }
39
```

```

40 int analyze( int * data, int size )
41 {
42     double avg, std;
43     if ( size > 1 )
44     {
45         avg = average( data, size );
46         std = std_deviation( data, size, avg );
47         printf( "Mittelwert=          %9.3e\t", avg );
48         printf( "Minimum= %9.3e\n", minimum(data,size));
49         printf( "Standardabweichung= %9.3e\t", std );
50         printf( "Maximum= %9.3e\n", maximum(data,size));
51     }
52     else printf( "sorry, nicht genug Messwerte\n" );
53     printf( "\n" );          // eine Leerzeile
54     return size > 1;
55 }
56
57 int main( int argc, char **argv )
58 {
59     #define SIZE_1    3
60     int data_1[ SIZE_1 ] = { 1, 2, 3 };
61
62     #define SIZE_2    5
63     int data_2[ SIZE_2 ] = { -1, -1, -1, -1, -1 };
64
65     #define SIZE_3    4
66     int data_3[ SIZE_3 ] = { 1, 1, 2, 2 };
67
68     analyze( data_1, SIZE_1 );
69     analyze( data_2, SIZE_2 );
70     analyze( data_3, SIZE_3 );
71 }

```

## 8. Zusatzaufgabe

Man mag es kaum glauben, aber sowohl den Mittelwert als auch die Standardabweichung kann man gemeinsam in einem Schleifendurchlauf berechnen. Dies klingt etwas merkwürdig, da man doch den Mittelwert für die einzelnen Terme der Standardabweichung benötigt. Aber es reicht tatsächlich aus, wenn man den Mittelwert erst ganz zum Schluss weiß. Dies bekommt man heraus, in dem man sich die Terme der Standardabweichung etwas genauer anschaut und alles ein wenig umformt. Die Interessierten finden hierzu ein paar Infos auf der Webseite und können bzw. sollten versuchen, dies umzusetzen.

# Vollständige Testprogramme

## Aufgabe 1:

```
1  #include <stdio.h>
2
3  int my_mult( int a, int b )
4  {
5      int res = a * b;
6      return res;
7      // return a * b would be much easier
8      // but we want to learn things
9  }
10 int main( int argc, char **argv )
11 {
12     printf( "res= %3d\n", my_mult(    3,  2 ) );
13     printf( "res= %3d\n", my_mult(    0,  4 ) );
14     printf( "res= %3d\n", my_mult(    2,  0 ) );
15     printf( "res= %3d\n", my_mult(   -2,  1 ) );
16     printf( "res= %3d\n", my_mult( -100, -2 ) );
17 }
```

## Aufgabe 2:

```
1  #include <stdio.h>
2
3  int test_div( int a, int b )
4  {
5      return a % b == 0;
6  }
7
8  int main( int argc, char **argv )
9  {
10     printf( "res= %3d\n", test_div(  4,  2 ) );
11     printf( "res= %3d\n", test_div(  5,  3 ) );
12     printf( "res= %3d\n", test_div(  4, -2 ) );
13     printf( "res= %3d\n", test_div( -4,  2 ) );
14     printf( "res= %3d\n", test_div( -8, -4 ) );
15 }
```

### Aufgabe 3:

```
1  #include <stdio.h>
2
3  #define SIZE 15
4
5  // example for a forward declaration,
6  // just to make it known to main
7  int print_array(int *array, int size);
8
9  int main (int argc, char** argv)
10 {
11     int i, array[ SIZE ];    // our array
12
13     // initializing it
14     for( i = 0; i < SIZE; i++ )
15         array[ i ]= 40 - 2 * i;
16
17     // printing it
18     print_array( array, SIZE );
19
20     // successfully done
21     return 0;
22 }
23
24 // here we print the array
25 int print_array(int vector[], int size)
26 {
27     int i;    // loop-index
28     for( i = 0; i < size; i++ )
29         printf( "Index=%2d, Wert=%2d\n", i, vector[ i ] );
30     return 1;
31 }
```

Der folgende Quelltext zeigt eine zeigerbasierte Alternative für die Funktion `print_array()`. Weitere Erläuterungen findet ihr auch in den Kapiteln [45](#) und [46](#).

```
1  int print_array(int * vector, int size)
2  {
3      int i;                                // loop-index
4      int * end = vector + size;            // loop-end-pointer
5      for( i = 0; vector < end; i++, vector++ )
6          printf( "Index=%2d, Wert=%2d\n", i, * vector );
7      return 1;
8  }
```

# Übungspaket 20

## Zeiger und Zeigervariablen

---

### Übungsziele:

1. Definition von Zeigervariablen
2. Verwendung von Zeigern
3. Arrays und Adressberechnungen

### Skript:

Kapitel: 45 und 46

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Zeiger! Bei diesem Wort bekommen viele Programmieranfänger Pusteln, Hautausschlag, hektische Flecken, Schweißausbrüche und dergleichen. Aber das muss nicht sein! Ein bisschen gezielt üben und schon ist es kinderleicht... Na ja, vielleicht nicht ganz kinderleicht, aber ohne nennenswerte Hürde. In diesem Übungspaket schauen wir uns erst einmal ganz einfach an, was Zeiger eigentlich sind und was wir mit derartigen Variablen machen können. Dabei steht vor allem die Frage nach dem Typ des Zeigers im Vordergrund. Zeiger und Funktionen haben wir uns für Übungspaket 21 auf. Am Ende der Übung mag sich der eine oder andere fragen, was man denn nun eigentlich von Zeigern hat, denn Zeiger sind erst einmal nur kompliziert. Die Antwort ist sehr einfach: Außer ein paar Vereinfachungen bei Array-Zugriffen eigentlich nicht viel. Die wirklich interessanten Dinge kommen erst in Übungspaket 21 und bei den dynamischen Datenstrukturen (Übungspakete 31 bis 33). Hierfür ist dieses Übungspaket eine *sehr* wichtige Vorbereitung.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Allgemeines zu Zeigern

Erkläre kurz, was ein Zeiger ist und was in einer Zeigervariablen abgespeichert wird.

Ein Zeiger ist nichts anderes als eine Adresse des Arbeitsspeichers. Implizit geht man meist davon aus, dass sich dort auch tatsächlich etwas befindet. Ein Zeiger „zeigt“ also auf dieses Objekt. Entsprechend hat eine Zeigervariable eine Adresse als Inhalt.

## Aufgabe 2: Zeiger-Syntax

Woran erkennt man eine Zeigerdefinition?	Am * vor dem Namen
Wie definiert man einen Zeiger?	Typ * Name
Wie weist man einem Zeiger etwas zu?	Name=Wert; //'Wert' ist eine Adresse
Wie greift man auf den Zeigerinhalt zu?	Name //liefert die gespeicherte Adr.
Wie schreibt man etwas an die Stelle, auf die der Zeiger zeigt?	* Name = Wert
Wie liest man von einer Stelle, auf die der Zeiger zeigt?	* Name
Wie vergleicht man zwei Zeigervariablen miteinander? (Beispiel)	Name_1 != Name_2
Gibt es Werte, die eine Zeigervariable nicht annehmen darf?	Nein, alle Werte sind erlaubt
Gibt es Adressen, auf die man nicht zugreifen darf?	Ja, beispielsweise 0
Darf die Zeigervariable dennoch diese Werte annehmen?	int *p=0; /*ok*/ *p=... /*Absturz*/

## Aufgabe 3: Typen von Zeigervariablen

Nehmen wir einmal an, eine Adresse des Arbeitsspeichers benötigt vier Bytes. Wie viele Bytes werden dann bei jedem Zugriff auf eine Zeigervariable im Arbeitsspeicher gelesen?

4 (vier) (four) (quattro)

Gibt es Ausnahmen, bei der mehr oder weniger Bytes kopiert werden? Nein! (No) (non)

Wenn es nun wirklich so ist, dass es von dieser Regel keine Ausnahme gibt, warum in Gottes Namen legt dann das Lehrpersonal so viel Wert darauf, dass ein Zeiger auch einen Typ haben muss? Erkläre dies in eigenen Worten:

Solange wir nur auf die Zeigervariablen zugreifen würden, wäre es in C tatsächlich egal, was für einen Typ diese Zeiger haben. Bei einer vier-Byte Architektur würden dann immer nur 32 Bit irgendwo hingeschrieben oder von irgendwo gelesen werden. Dies ändert sich, sobald wir auf diejenige Speicheradresse zugreifen, auf die der Zeiger zeigt. Zeigt beispielsweise ein Zeiger `char *p` auf ein Zeichen, so wird beim Zugriff `*p = 'a'` ein Byte kopiert. Bei einem `int *p`-Zeiger wären dies hingegen `*p = 4711` vier Bytes. Aber in beiden Fällen würden bei der Anweisung `p = 0` vier Bytes an die Speicherstelle der Zeigervariablen `p` kopiert werden. Um hier also korrekt arbeiten zu können, *muss* der Compiler immer wissen, was sich am Ende des Zeigers befindet.

## Aufgabe 4: Zeiger und Arbeitsspeicher

Vervollständige für das folgende Programm das skizzierte Speicherbildchen.

1 <code>int i;</code>	Adresse	Variable	Wert
2 <code>int *p;</code>	0xFFEE107C	<code>int i:</code>	4711
3 <code>p = &amp;i;</code>	0xFFEE1078	<code>int *p:</code>	0xFFEE107C
4 <code>*p = 4711;</code>			

## Aufgabe 5: Zeiger, Arrays und Zeigeroperationen

Erkläre nochmals kurz, was ein Array ist und was der Name des Arrays symbolisiert:

Ein Array ist eine Zusammenfassung mehrerer Elemente des *selben* Typs, die alle nacheinander im Arbeitsspeicher abgelegt werden. Der Name des Arrays ist eine *Konstante* und symbolisiert die Anfangsadresse des Arrays.

Zeiger, Arrays und `int`-Terme kann man eingeschränkt miteinander verknüpfen. Ergänze folgende Tabelle, in der `n` ein `int`-Ausdruck, `a` ein Array und `p` und `q` beliebige Zeiger sind.

Ausdruck	Alternative	Effekt
<code>&amp;a[ n ]</code>	<code>a + n</code>	Bestimmt die Adresse des <code>n</code> -ten Elementes von <code>a</code>
<code>p + n</code>	<code>&amp;p[ n ]</code>	<code>p</code> plus <code>n</code> Elemente weiter „oben“
<code>a[ n ]</code>	<code>*(a + n)</code>	Inhalt des <code>n</code> -ten Elementes von <code>a</code>
<code>*(p + n)</code>	<code>p[ n ]</code>	Inhalt des <code>n</code> -ten Elementes von <code>p</code>
<code>p - n</code>	<code>&amp;p[ -n ]</code>	<code>p</code> plus <code>n</code> Elemente weiter „unten“
<code>p - q</code>		Bestimmt die Zahl der Basiselemente zwischen <code>p</code> und <code>q</code>

## Teil II: Quiz

---

### Aufgabe 1: Variablen und Zeiger

In dieser Quizaufgabe greifen wir nochmals das Miniprogramm der letzten Übungsaufgabe des ersten Teils auf. Demnach haben wir folgende Situation vorliegen:

	Adresse	Variable	Wert
1 <code>int i = 4711;</code>	0xFFEE107C	<code>int i:</code>	4711
2 <code>int *p = &amp;i;</code>	0xFFEE1078	<code>int *p:</code>	0xFFEE107C

Ferner gehen wir wieder davon aus, dass sowohl eine `int`-Variable als auch ein Zeiger genau vier Bytes im Arbeitsspeicher belegen.

Vervollständige folgende Tabelle. Gehe dabei immer davon aus, dass am Anfang jedes Ausdrucks alle Variablen wieder auf obige Initialwerte zurückgesetzt werden.

Ausdruck	Typ	Ergebnis	Kommentar
<code>i</code>	<code>int</code>	4711	Dies ist einfach nur der Wert von <code>i</code>
<code>i += 1</code>	<code>int</code>	4712	Zuweisungen sind auch Ausdrücke
<code>i--</code>	<code>int</code>	4711	Post-Dekrement, Ausdruck: 4711, aber <code>i = 4710</code>
<code>p += 1</code>	<code>int *</code>	0xFFEE1080	Ein Element ergibt hier vier Bytes
<code>--p</code>	<code>int *</code>	0xFFEE1078	<i>Pre</i> -Dekrement, ein Element, vier Bytes
<code>p[ 0 ]</code>	<code>int</code>	4711	Zugriff auf das erste Array-Element
<code>&amp; i</code>	<code>int *</code>	0xFFEE107C	Adresse der Variablen <code>i</code>
<code>&amp; p</code>	<code>int **</code>	0xFFEE1078	Der Ort, an dem wir den Zeiger <code>p</code> finden
<code>i &gt; 0</code>	<code>int</code>	1	Logisch wahr ergibt den Wert 1
<code>p &gt; 0</code>	<code>int</code>	1	wie oben
<code>p &gt; 10</code>	<code>int</code>	1	wie oben
<code>p &gt; p + 1</code>	<code>int</code>	0	Logisch falsch ergibt 0



## Aufgabe 2: Zeichenketten und Zeiger

Diese Quizaufgabe ist ähnlich der vorherigen. Nur behandeln wir jetzt nicht „normale“ Variablen sondern Zeichenketten. Zugegebenermaßen macht dies die Sache etwas komplexer. Wenn man sich aber *langsam* und *Schritt für Schritt* von innen nach außen vorarbeitet, ist die Sache doch recht überschaubar. Im Übrigen gehen wir wieder davon aus, dass alle Zeiger genau vier Bytes im Arbeitsspeicher belegen. Betrachten wir nun folgende Code-Zeilen:

```
1 char  buf1[] = "Fruehling";
2 char *buf2   = "Sommer";
3 char *buf3[] = { "Herbst", "Winter" };
```

Beschreibe kurz mit eigenen Worten, welchen Datentyp die drei Variablen haben:

buf1	Ein Array vom Typ char
buf2	Ein Zeiger auf ein char
buf3	Ein Array mit zwei Zeigern auf char

Vervollständige zunächst folgendes Speicherbildchen, denn es hilft drastisch beim Finden der richtigen Lösungen :-)

Segment: Stack

Adresse	Variable	Wert
0xFE34	char *buf3[ 1 ]:	0xF840
0xFE30	char *buf3[ 0 ]:	0xF838
0xFE2C	char *buf2 :	0xF830
0xFE28		'g' '\0'
0xFE24		'h' 'l' 'i' 'n'
0xFE20	char buf1[] :	'F' 'r' 'u' 'e'

Segment: Konstanten

Adresse	Wert
0xF844	'e' 'r' '\0'
0xF840	'W' 'i' 'n' 't'
0xF83C	's' 't' '\0'
0xF838	'H' 'e' 'r' 'b'
0xF834	'e' 'r' '\0'
0xF830	'S' 'o' 'm' 'm'

Vervollständige nun folgende Tabelle:

Ausdruck	Typ	Ergebnis	Kommentar
buf1	char []	0xFE20	Das komplette Array; Wert: Anfangsadresse
buf2	char *	0xF830	Eine ganz gewöhnliche Zeigervariable
buf3	char **	0xFE30	Ein Array mit zwei char * Zeigern
sizeof( buf1 )	int	10	Genau neun Nutzzeichen plus ein Nullbyte
sizeof( buf2 )	int	4	Ein klassischer Zeiger
sizeof( buf3 )	int	8	Ein Array bestehend aus zwei Zeigern
buf1[ 3 ]	char	'e'	Genau eines der Zeichen, nämlich das vierte
buf1 + 3	char *	0xFE23	Anfangsadresse plus drei Bytes, also die Adresse von 'e'
*(buf1 + 3)	char	'e'	Identisch mit buf1[ 3 ]
buf2[ 3 ]	char	'm'	Das vierte Zeichen der Zeichenkette
buf2 + 3	char *	0xF833	Adresse <i>in</i> buf2 plus 3 Bytes, identisch mit &buf2[ 3 ], also die Adresse des zweiten 'm'
*(buf2 + 3)	char	'm'	Identisch mit buf2[ 3 ]
buf3[ 0 ][ 2 ]	char	'r'	Das dritte Zeichen der ersten Zeichenkette
buf3[ 1 ][ 0 ]	char	'W'	Das erste Zeichen der zweiten Zeichenkette
& buf1	char **	0xFE20	Der Ort an dem sich buf1 befindet
& buf2	char **	0xFE2C	Der Ort an dem sich buf2 befindet
& buf3	*(char * [2])	0xFE30	Der Ort an dem sich buf3 befindet
& buf1[ 4 ]	char *	0xFE24	Adresse des 5. Zeichens (Stacksegment)
& buf2[ 4 ]	char *	0xF834	Adresse des 5. Zeichens (Konstantensegment)
& buf3[0][2]	char *	0xF83A	Adresse des 3. Zeichens (Konstantensegment)

## Teil III: Fehlersuche

---

### Aufgabe 1: Definition und Verwendung von Zeigern

Das folgende Programm enthält einige einfache Variablen und einige Zeiger. Der gewünschte Typ der Zeiger geht immer unmittelbar aus dem Namen hervor. Ferner gehen die gewünschten Zeigeroperationen aus dem Kontext hervor. Ab Zeile 5 ist in jeder Zeile ein Fehler vorhanden. Finde und korrigiere diese, damit DR. A. POINTER ruhig schlafen kann.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i = 4711, *i_ptr, *i_ptr_ptr*;
6      char c = 'x', **c_ptr;
7
8      i_ptr = i;
9      *i_ptr_ptr = & i_ptr;
10     c_ptr = * c;
11
12     printf( "i=%d\n", i_ptr );
13     printf( "c='%c'\n", c_ptr* );
14     printf( "i hat die Adresse      %p\n", * i_ptr );
15     printf( "c hat die Adresse      %p\n", & c_ptr );
16     printf( "i_ptr hat die Adresse %p\n", i_ptr );
17     printf( "i='%c\n", **i_ptr_ptr );
18 }
```

Zeile	Fehler	Erläuterung	Korrektur
5	*...*	Bei der Definition müssen die Sternchen für die Zeiger vor dem Namen stehen.	**i_ptr_ptr
6	* zu viel	Da wir nur einen einfachen Zeiger haben wollen, darf vor dem Namen auch nur ein * stehen.	*c_ptr
8	& fehlt	Da wir die Adresse der Variablen i benötigen, muss vor ihr der Adressoperator stehen.	& i
9	* falsch	Der Stern vor dem Zeiger i_ptr_ptr ist in diesem Falle unsinnig, er muss weg.	i_ptr_ptr
10	* statt &	Hier benötigen wir wieder die Adresse einer Variablen also ein & und nicht ein *.	& c

Zeile	Fehler	Erläuterung	Korrektur
12	* fehlt	Da wir hier auf den Inhalt derjenigen Speicherstelle zugreifen wollen, auf die <code>i_ptr</code> zeigt, benötigen wir hier ein * zum Dereferenzieren.	<code>* i_ptr</code>
13	* an der falschen Stelle	Der * zum Dereferenzieren muss vor dem Namen stehen nicht dahinter.	<code>* c_ptr</code>
14	* zu viel	Wir wollen die Adresse von <code>i</code> ausgeben, die bereits in <code>i_ptr</code> steht. Also dürfen wir nicht dereferenzieren, denn dann würden wir den Inhalt von <code>i</code> ausgeben.	<code>i_ptr</code>
15	& zu viel	Hier gilt das gleiche wie ein Fehler weiter oben. Der Ausdruck <code>&amp; c_ptr</code> würde uns fälschlicherweise die Adresse der Variablen <code>c_ptr</code> liefern.	<code>c_ptr</code>
16	falsche Variable oder & fehlt	Da wir die Adresse der Variablen <code>i_ptr</code> ausgeben wollen, dürfen wir nicht den Inhalt dieses Zeigers ausgeben, sondern benötigen dessen Adresse. Die steht in <code>i_ptr_ptr</code> oder kann mittels <code>&amp; i_ptr</code> gebildet werden.	<code>i_ptr_ptr</code> oder <code>&amp; i_ptr</code>
17	falsches Format	Hier ist eigentlich alles richtig. Nur muss das Format auf <code>%d</code> für Integer abgeändert werden.	<code>%d</code>

### Programm mit Korrekturen:

```

1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      int i = 4711, *i_ptr, **i_ptr_ptr;
6      char c = 'x', *c_ptr;
7
8      i_ptr = & i;
9      i_ptr_ptr = & i_ptr;
10     c_ptr = & c;
11
12     printf( "i=%d\n", *i_ptr );
13     printf( "c='%c'\n", *c_ptr );
14     printf( "i hat die Adresse      %p\n", i_ptr );
15     printf( "c hat die Adresse      %p\n", c_ptr );
16     printf( "i_ptr hat die Adresse %p\n", i_ptr_ptr );
17     printf( "i=%d\n", **i_ptr_ptr );
18 }
```

## Teil IV: Anwendungen

### Aufgabe 1: Definition von Variablen und Zeigern

In dieser Aufgabe geht es lediglich darum, „normale“ Variablen und Zeigervariablen zu definieren. Vervollständige dafür die folgende Tabelle. Wir gehen wieder davon aus, dass sowohl `int`-Variablen also auch Zeiger vier Bytes im Arbeitsspeicher belegen. Da die letzten beiden Definitionen recht schwierig sind, sollten sie mit den Betreuern besprochen werden.

Was	C-Definition	<code>sizeof(Variable)</code>
Eine <code>int</code> -Variable.....	<code>int i</code>	4
Eine <code>char</code> -Variable.....	<code>char c</code>	1
Ein Zeiger auf eine <code>int</code> -Variable.....	<code>int *p</code>	4
Ein Zeiger auf eine <code>char</code> -Variable.....	<code>char *p</code>	4 (ein Zeiger)
Ein Array mit drei <code>int</code> -Elementen.....	<code>int a[ 3 ]</code>	12 (drei <code>ints</code> )
Ein Array mit drei <code>int</code> -Zeigern.....	<code>int *a[ 3 ]</code>	12 (drei Zeiger)
Ein Zeiger auf ein Array mit drei <code>int</code> -Elementen	<code>int (*p)[ 3 ]</code>	4 (nur <i>ein</i> Zeiger)

Illustriere die letzten beiden Definitionen mittels eines kleinen Speicherbildchens und trage in die jeweiligen Variablen Phantasiewerte ein.

Definition: `int *a[3]`

Adresse	Variable	Wert
0xFFEE1028	<code>a[ 2 ] :</code>	0xFFFF3034
0xFFEE1024	<code>a[ 1 ] :</code>	0xFFFF44CC
0xFFEE1020	<code>a a[ 0 ] :</code>	0xFFFF2D14

Anmerkung:

Das Array `a` hat genau drei Elemente. Jedes dieser Elemente ist ein Zeiger, der seinerseits auf ein `int` zeigt.

Definition: `int (*p)[3]`

Adresse	Variable	Wert
0xFFEE1028	<code>[ 2 ] :</code>	-1
0xFFEE1024	<code>[ 1 ] :</code>	815
0xFFEE1020	<code>[ 0 ] :</code>	4711

0xFFEE0300	<code>p :</code>	0xFFEE1020
------------	------------------	------------

Anmerkung:

Die Variable `p` ist ein Zeiger auf ein Array. Dort befinden sich dann drei `int`-Werte. Bei diesem Array kann es sich um ein „gewöhnliches“ oder ein dynamisch angelegtes Array (siehe auch Übungspaket 29) handeln.

## Aufgabe 2: Zeiger auf Zeiger

Zeiger auf Zeiger, langsam wird's ernst. Nehmen wir diesmal den Basistyp `double`. Aus irgendeinem unerfindlichen Grund brauchen wir hiervon eine einfache Variable. Nennen wir sie `d` wie `double`. Ferner brauchen wir noch einen Zeiger `p` auf diese Variable, einen Zeiger `pp`, der auf diesen Zeiger zeigt und schließlich einen Zeiger `ppp` der auf letzteren zeigt. Definiere die entsprechenden Variablen und vervollständige unten stehendes Speicherbildchen. Bei Schwierigkeiten stehen die Betreuer für Diskussionen zur Verfügung.

```

1 double d;                // the single variable
2 double *p;               // a pointer to a double
3 double **pp;              // a pointer to a double pointer
4 double ***ppp;            // a pointer to a double ptr. ptr.
5
6 d = 3.141;               // giving it a well-known value
7 p = &d;                  // p now points to d;   &d -> double *
8 pp = &p;                  // pp now points to p;  &p -> double **
9 ppp = &pp;                // ppp now points to pp; &pp -> double ***

```

Adresse	Variable	Wert	Adresse	Variable	Wert
0xFFEE100C	ppp : 0xFFEE1008		0xFFEE1004	p : 0xFFEE1000	
0xFFEE1008	pp : 0xFFEE1004		0xFFEE1000	d : 3.141	

Vervollständige die folgenden Ausdrücke unter der Annahme, dass ein `double` acht und ein Zeiger vier Bytes im Arbeitsspeicher belegt.

Ausdruck	Typ	sizeof(Ausdruck)	Ergebnis
d	double	8	3.141
p	double *	4	0xFFEE1000
*p	double	8	3.141
pp	double **	4	0xFFEE1004
*pp	double *	4	0xFFEE1000
**pp	double	8	3.141
ppp	double ***	4	0xFFEE1008
*ppp	double **	4	0xFFEE1004
**ppp	double *	4	0xFFEE1000
***ppp	double	8	3.141

Zeige alle vier Möglichkeiten, dem Speicherplatz der Variablen `d` eine 2.718 zuzuweisen.

- `d = 2.718`
- `*p = 2.718`
- `**pp = 2.718`
- `***ppp = 2.718`

## Aufgabe 3: Verwendung von Zeigern

### 1. Aufgabenstellung

Definiere ein Array einer beliebigen Größe, eines beliebigen Typs. Schreibe zwei Programme. Das erste Programm soll alle Elemente dieses Arrays mittels einer `for`-Schleife und einer Indexvariablen initialisieren. Schreibe ein zweites Programm, dass diese Initialisierung mittels einer `for`-Schleife und Zeigern (also *nicht* mit einer Indexvariablen) durchführt.

### 2. Pflichtenheft

Aufgabe : Entwicklung zweier Programme, die ein Array mittels einer Indexvariablen bzw. mittels zweier Zeiger initialisieren.

Eingabe : keine

Ausgabe : keine

Sonderfälle: keine

### 3. Kodierung

Lösung mittels Indexvariable:

```
1 #define SIZE    10
2 #define INIT    0
3
4 int main( int argc, char **argv )
5 {
6     int arr[ SIZE ];
7     int i;
8     for( i = 0; i < SIZE; i++ )
9         arr[ i ] = INIT;
10 }
```

Lösung mittels Zeigern:

```
1 #define SIZE    10
2 #define INIT    0
3
4 int main( int argc, char **argv )
5 {
6     int arr[ SIZE ];
7     int *p;
8     for( p = arr; p < arr + SIZE; p++ )
9         *p = INIT;
10 }
```

# Übungspaket 21

## Funktionen mit Zeigern und Arrays als Parameter

---

### Übungsziele:

1. Funktionen mit Zeigern als Parameter
2. Emulation von Variablenparametern
3. Funktionen mit Arrays als Parameter
4. Zeigervariablen im Stack-Frame

### Skript:

Kapitel: 44 bis 47 sowie die Übungspakete 15, 19 und 20

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In den beiden vorherigen Übungspakten haben wir Funktionen und Zeiger getrennt voneinander geübt. Nun wollen wir beide Aspekte zusammenbringen. Entsprechend werden wir Zeigerarithmetik betreiben und Funktionen sowohl mit Zeigern als auch Arrays aufrufen. Neben einigen einfachen, grundlegenden Algorithmen greifen wir am Ende wieder das alte Thema „Suchen und Sortieren“ auf aus Übungspaket 15.

Dieses Übungspaket ist eher lang, schwierig und recht aufwändig. Aber wer hier einigermaßen durchkommt und die Dinge versteht, die er hier bearbeitet, der hat so ziemlich das Schlimmste hinter sich gebracht. Mit dem hier erworbenen Wissen lässt sich der Rest frohen Mutes angehen. Wer aber noch Schwierigkeiten mit Zeigern, Arrays oder allgemein mit Funktionsaufrufen hat, der sollte sich die entsprechenden Übungspakete nochmals anschauen. Denn je besser man hier vorbereitet ist, um so leichter geht's.



# Teil I: Stoffwiederholung

---

Zunächst wiederholen wir zum x-ten Male einige wesentliche Aspekte aus den vorherigen Übungspaketen. Auch wenn es einigen bereits aus den Ohren heraushängt, so kann man dennoch einige wesentliche Punkte anfangs nicht oft genug erwähnen.

## Aufgabe 1: Was sind Zeiger und Arrays

Beantworte die drei folgenden Fragen wieder kurz in eigenen Worten. Was ist ein Array?

Ein Array ist eine Zusammenfassung mehrerer Variablen des selben Typs. Die einzelnen Elemente spricht man über den Array-Namen und einen Index vom Typ <code>int</code> an.
---

Was repräsentiert der Array-Name? 

Die Anfangsadresse des Arrays
-------------------------------

Was ist ein Zeiger?

Ein Zeiger ist eine Variable, die lediglich eine Speicheradresse aufnehmen kann. Für den Compiler ist es äußerst wichtig zu wissen, von welchem Typ das Zielelement ist.
--

## Aufgabe 2: Grundlagen der Zeiger und Arrays

Für die Beantwortung der weiteren Fragen gehen wir davon aus, dass wir uns immer in demjenigen Block befinden, in dem das Array definiert wurde. Unter dem Begriff „Zielelement“ verstehen wir immer dasjenige Element, auf das der Zeiger zeigt.

Was repräsentiert der Array-Name	Die Anfangsadresse des Arrays
Ist der <i>Array-Name</i> eine Konstante oder Variable?	Ein Konstante
Kann man dem Array-Namen etwas zuweisen?	Nein, denn er ist eine Konstante!
Was ergibt <code>sizeof( Array-Name )</code> ?	Die Größe <i>aller</i> Elemente
Was ergibt <code>sizeof( Array-Name )</code> nicht?	Die Größe <i>eines</i> Elementes
Was ergibt <code>sizeof( * Array-Name )</code> ?	Die Größe <i>eines</i> Elementes
Ist ein Zeiger eine Konstante oder Variable?	Eine Variable
Kann man einem Zeiger etwas zuweisen?	Ja, denn er ist eine Variable
Was ergibt <code>sizeof( Zeiger )</code> ?	Den Platzbedarf des Zeigers
Was ergibt <code>sizeof( Zeiger )</code> <i>nicht</i> ?	Den Platzbedarf des Zielelementes
Was ergibt <code>sizeof( * Zeiger )</code> ?	Die Größe des Zielelementes

## Aufgabe 3: Fragen zu Funktionen

Wie nennt man die Platzhalter im Funktionskopf?	Formale Parameter
Wie nennt man die Werte beim Funktionsaufruf?	Aktuelle Parameter
Wo steht die Rücksprungadresse?	Im Stack-Frame
Wo steht der Funktionswert?	Im Stack-Frame
Wo stehen die lokalen Variablen?	Im Stack-Frame
Wann wird eigentlich der Stack-Frame angelegt?	Bei <i>jedem</i> Funktionsaufruf
Was lässt sich dadurch realisieren?	Rekursive Funktionsaufrufe
Bietet C die Parameterübergabe: Call-by-Value?	Ja
Was passiert dabei?	Die Parameterwerte werden kopiert
Bietet C den Call-by-Reference Mechanismus?	Nein, no, non, njet, $\acute{o}\chi\iota$

## Aufgabe 4: Zeiger und Arrays als formale Parameter

Gegeben seien die beiden Funktionen `int f(int * p)` und `int g(int arr[])`. Bei den folgenden Fragen befinden wir uns immer *innerhalb* der Funktionsrumpfe von `f()` und `g()`.

Welchen Typ hat der Formalparameter <code>p</code> ?	<code>int *</code>
Welchen Typ hat der Formalparameter <code>arr</code> ?	<code>int *</code>
Unterscheiden sich beide Parameter voneinander?	Nein, es gibt <i>keinen</i> Unterschied
Kennt <code>f()</code> die Größe des übergebenen Arrays?	Nein, keine Chance
Kennt <code>g()</code> die Größe des übergebenen Arrays?	Nein, auch keine Chance
Was müssen wir machen?	Wir geben die Größe <i>immer</i> mit an
Und wie machen wir das?	Mittels eines zweiten Parameters
Wie sähe das bei <code>f()</code> aus?	<code>int f( int * p, int size )</code>

## Aufgabe 5: Kurzformen von Zeigerausdrücken

Was bedeutet der Ausdruck `c = *p++`, wenn `c` und `p` ein `char` bzw. Zeiger darauf sind?

Der Zeiger `p` hat einen Wert und „zeigt“ in den Arbeitsspeicher. Das dortige Zeichen wird genommen und der Variablen `c` zugewiesen. Anschließend wird der Zeigerwert um eins erhöht. Es handelt sich also um eine Kurzform von: `c = *p; p = p + 1;`

## Teil II: Quiz

### Aufgabe 1: Zeiger und Arrays als Parameter

Für die folgenden Quizfragen betrachten wir folgende Definitionen und Speicherbild:

```
1 int i = 4711;      3 int a[] = { 1, 2, 4, 8, 16 };
2 int *p = &i;      4 int size_a = sizeof(a)/sizeof(a[0]);
```

Adresse	Variable	Wert	Adresse	Variable	Wert
0xFE7C	int i :	4711	0xFE6C	int a[ 3 ]:	8
0xFE78	int * p :	0xFE7C	0xFE68	int a[ 2 ]:	4
0xFE74	int size_a:	5	0xFE64	int a[ 1 ]:	2
0xFE70	int a[ 4 ]:	16	0xFE60	int a[ 0 ]:	1

Vervollständige obiges Speicherbild. `f()` ist nun eine Funktion, deren Parameter jedes Mal richtig spezifiziert sind. Notiere nun jeweils Typ und Wert der beiden aktuellen Parameter.

Aufruf	Parameter 1		Parameter 2		Was „sieht“ <code>f()</code>
	Typ	Wert	Typ	Wert	
<code>f( a, size_a )</code>	int *	0xFE60	int	5	Array <code>a[ 0 ] ... a[ 4 ]</code>
<code>f( a + 1, 3 )</code>	int *	0xFE64	int	3	Array <code>a[ 1 ] ... a[ 3 ]</code>
<code>f( &amp;(a[1]), 3 )</code>	int *	0xFE64	int	3	dito
<code>f( &amp;i, 1 )</code>	int *	0xFE7C	int	1	ein Array mit <code>i</code> als Element
<code>f( p, 1 )</code>	int *	0xFE7C	int	1	dito.
<code>f( &amp;p, 1 )</code>	int **	0xFE78	int	1	ein Array mit <i>einem</i> <code>int *</code>
<code>f( a, a+size_a )</code>	int *	0xFE60	int *	0xFE74	Array mit Start- und Endzeiger

**Anmerkung:** Im letzten Beispiel zeigt **endzeiger** *hinter* das Array `a`. Ein Zugriff auf diese Speicherstelle wäre undefiniert; der Zeigerwert ist erlaubt, das Dereferenzieren aber nicht. Daher müsste beispielsweise in Schleifen immer auf „`p < endzeiger`“ getestet werden.

Gegeben seien die beiden folgenden Programmzeilen. Vervollständige das Speicherbild für den Fall, dass beide abgearbeitet wurden.

	Adresse	Variable	Wert	Konstanten Segment:
1 char *p = "Ah!";	0xFE84	char c:	'A'	0xFE02 '!' '\0'
2 char c = *p++;	0xFE80	char *p:	0xFE01	0xFE00 'A' 'h'

# Teil III: Fehlersuche

---

## Aufgabe 1: Zeiger und Funktionen

Unser Grufti DR. DO-IT-RIGHT hat bei den Übungen zugeschaut und selbst ein wenig probiert. Bei seinen Versuchen handelt es sich nur um ein paar Testzeilen und nicht um ein sinnvolles Programm. Die Intention wird jeweils durch den Funktionsnamen bzw. die entsprechenden Begleitkommentare klar. Viel Spass beim Finden und Korrigieren der Fehler.

```
1 int who_is_larger( int arr_1[], int arr_2[] )
2     {
3         if ( sizeof(arr_1) > sizeof(arr_2) )
4             return 1;                // arr_1 is larger
5         else if ( sizeof(arr_1) < sizeof(arr_2) )
6             return 2;                // arr_2 is larger
7         else return 0;                // none
8     }
9 int set_to_zero( int a )
10    {
11        int *p;                      // one pointer
12        *p = & a;                    // get address of the actual parameter
13        *p = 0;                      // accessing actual parameter and set to zero
14    }
15 int main( int argc, char **argv )
16    {
17        int *p, a[ 10 ], b[ 5 ], i;
18        for( p = a + 10; a < p; a++ )    // the loop
19            *a = -1;                      // init with -1
20        a -= 10;                          // reset a to its beginning
21        set_to_zero( i );                  // set i to zero
22        i = who_is_larger( a, b );        // comparing sizes
23    }
```

Zeile	Fehler	Erläuterung	Korrektur
1-8	sizeof()	Die Funktion soll die beiden Arrays hinsichtlich ihrer Größe vergleichen. Aber was haben wir gelernt: Auch wenn der Formalparameter <i>augenscheinlich</i> als Array definiert ist, wird beim Funktionsaufruf nicht das Array sondern nur dessen <i>Anfangsadresse</i> übergeben. Der Aufruf von <code>sizeof()</code> innerhalb der Funktion ist also <i>nicht</i> sinnvoll: er bestimmt nicht die Größe des Arrays sondern nur die eines Zeigers.	Den Aufruf von <code>sizeof()</code> nach außen verlagern.

Zeile	Fehler	Erläuterung	Korrektur
9-14	Adressberechnung	Hier versucht DR. DO-IT-RIGHT die Adresse der aktuellen Parameter über den Trick eines zusätzlichen Zeigers herauszubekommen. Leider falsch, denn die formalen Parameter werden im Stack-Frame angelegt und erhalten ihre Werte als <i>Kopien</i> der aktuellen Parameter. Entsprechend wird hier der aktuelle Wert der Variablen <code>i</code> übergeben. In Zeile 13 erhält der Zeiger <code>p</code> lediglich die Adresse des formalen Parameters <code>a</code> , und nicht die des aktuellen Parameters. Klappt also nicht.	Die Funktion muss komplett umgeschrieben werden.
18-20	Veränderung von <code>a</code>	Die Idee ist einfach: Innerhalb der Schleife sollen alle Elemente mit dem Wert <code>-1</code> initialisiert werden. Die Vorgehensweise mit den Zeigern ist eigentlich richtig. Nur darf <code>a</code> nicht verändert werden, da es sich hier nicht um einen Zeiger sondern eine Konstante handelt, die den Anfang des Arrays symbolisiert.	Es müssen nur Zeiger und Array vertauscht werden.
21-22	Funktionsaufrufe	Beide Anweisungen sind aus obigen Gründen nicht sinnvoll, sie müssen entsprechend angepasst werden.	Siehe oben

### Programm mit Korrekturen:

```

1 int who_is_larger( int size_1, int size_2 )
2 {
3     if ( size_1 > size_2 )
4         return 1;                                // arr_1 is larger
5     else if ( size_1 < size_2 )
6         return 2;                                // arr_2 is larger
7     else return 0;                                // none
8 }
9 int set_to_zero( int *a )                        // one pointer
10 {
11     *a = 0;    // accessing actual parameter and set to zero
12 }
13 int main( int argc, char **argv )
14 {
15     int *p, a[ 10 ], b[ 5 ], i;
16     for( p = a; p < a + 10; p++ )                // the loop
17         *p = -1;                                // init with -1
18     set_to_zero( & i );                          // set i to zero
19                                                // comparing sizes
20     i = who_is_larger( sizeof( a ), sizeof( b ) );
21 }

```

# Teil IV: Anwendungen

---

## Vorbemerkung

Den meisten von euch sollte klar geworden sein, dass es hier irgendwie um Funktionen und Zeiger geht. Sollte dies im Einzelfalle doch nicht der Fall sein, dann unbedingt mindestens drei Übungspakete zurück gehen, das Skript nochmals lesen und sofort damit anfangen.

Ferner sollte jedem, der hier weiter arbeitet, folgende Dinge klar sein, und zwar glasklar:

1. In der Programmiersprache C gibt es *nur einen* Parameterübergabemechanismus, den man Call-by-Value nennt. Dabei werden die aktuellen Parameter ausgewertet und den formalen Parametern als Werte zugewiesen.
2. Es gibt in C *kein* Call-by-Reference, auch wenn dies an vielen Stellen immer wiederholt wird.
3. Wenn man Zeiger, beispielsweise die Adressen beliebiger Variablen, übergibt, dann wird dies immer noch mittels Call-by-Value abgearbeitet. Aber dadurch, dass die Funktion jetzt nicht die Variablenwerte sondern deren Adressen bekommt, kann sie quasi „aus der Funktion heraus schauen“ und im Block der aufrufenden Stelle frisch, fromm, fröhlich, frei ändern wie es ihr beliebt.

## Aufgabe 1: Tausch zweier char-Parameter

### 1. Aufgabenstellung

Ziel dieser Aufgabe ist es, eine Funktion `my_ch_swap()` zu entwickeln, die die Werte zweier `char`-Variablen nachhaltig vertauscht.

**Beispiel:** Im folgenden Programm soll erst `ko` und dann `ok` ausgegeben werden.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv )
4  {
5      char c1 = 'k', c2 = 'o';
6
7      printf( "vorher : %c%c\n", c1, c2 );
8      my_ch_swap( ... );
9      printf( "nachher: %c%c\n", c1, c2 );
10 }
```

### 2. Vorüberlegungen

Da die zu entwickelnde Funktion Variablenwerte vertauschen soll, ist es wenig sinnvoll, diese Werte einfach an die formalen Parameter der Funktion zu übergeben. Vielmehr benötigen wir Zeiger auf die konkreten Variablen, damit wir deren Inhalte nachhaltig tauschen können. Andernfalls würden zwar die Variablenwerte *innerhalb* der Funktion getauscht werden, was aber ausserhalb der Funktion keinen Effekt hätte. Also benötigen wir keine „normalen“ Parameter sondern die allseits geliebten Zeiger.

### 3. Pflichtenheft

Aufgabe : Entwicklung einer Funktion, die zwei **char**-Werte vertauscht.  
Parameter : Zwei Zeiger auf Zeichen  
Rückgabewert : Da wir keinen Rückgabewert benötigen, nehmen wir **void** als Typ.

### 4. Implementierung

Da die Aufgabe so einfach ist, können wir wie folgt verfahren:

Funktion zum Tauschen zweier Zeichen

Parameter: Pointer to Zeichen: p1, p2

Variable: Zeichen: c

setze c = p1 dereferenziert

setze p1 dereferenziert = p2 dereferenziert

setze p2 dereferenziert = c

### 5. Kodierung

```
1  #include <stdio.h>
2
3  void my_ch_swap( char *p1, char *p2 )
4  {
5      char c;
6      c = *p1; *p1 = *p2; *p2 = c;
7  }
8
9  int main( int argc, char **argv )
10 {
11     char c1 = 'k', c2 = 'o';
12
13     printf( "vorher : %c%c\n", c1, c2 );
14     my_ch_swap( & c1, & c2 );
15     printf( "nachher: %c%c\n", c1, c2 );
16 }
```

### 6. Stack Frame

Zeichne den Stack-Frame zu Beginn des Funktionsaufrufs:

Adresse	Variable	Wert
0xFFEE1080	.....	.....
0xFFEE4001	char c1:	'k'
0xFFEE4000	char c2:	'o'
0xFFEE1080	.....	.....
0xFFEE300C	char *p2:	0xFFEE4001
0xFFEE3008	char *p1:	0xFFEE4000
0xFFEE3004	CPU PC:	„Zeile 8“
0xFFEE3000	char c:	

**Anmerkung:** Da die Funktion keinen Rückgabewert hat, befindet sich hierfür auch kein reserviertes Feld im Stack-Frame.

## Aufgabe 2: Tausch zweier double-Parameter

### 1. Aufgabenstellung

Wiederhole die vorherige Aufgabe. Nur sollen diesmal zwei **double**-Werte vertauscht werden. Aufgrund der Vorarbeiten könnt ihr direkt mit der Kodierung beginnen.

### 2. Kodierung

```
1  #include <stdio.h>
2
3  void my_double_swap( double *p1, double *p2 )
4  {
5      double d;
6      d = *p1; *p1 = *p2; *p2 = d;
7  }
8
9  int main( int argc, char **argv )
10 {
11     double d1 = 47.11, d2 = 8.15;
12
13     printf( "vorher : %5.2f %5.2f\n", d1, d2 );
14     my_double_swap( & d1, & d2 );
15     printf( "nachher: %5.2f %5.2f\n", d1, d2 );
16 }
```



## Aufgabe 3: Ringtausch dreier int-Parameter

### 1. Aufgabenstellung

Entwickle eine Funktion, die die Werte dreier `int`-Parameter zyklisch vertauscht.

**Beispiel:** Die Werte der Parameter `a`, `b` und `c` sollen nach dem Funktionsaufruf in den Variablen `b`, `c` und `a` stehen.

Da auch diese Aufgabe den beiden vorherigen sehr ähnelt, reichen hier Implementierung und Kodierung.

### 2. Implementierung

In Anlehnung an die letzten beiden Aufgaben können wir direkt die Implementierung wie folgt formulieren:

Funktion zum zyklischen Tauschen dreier `int`-Werte

Parameter: Pointer to Integer: `p1`, `p2`, `p3`

Variable: Integer: `i`

setze `i` = `p1` dereferenziert

setze `p1` dereferenziert = `p2` dereferenziert

setze `p2` dereferenziert = `p3` dereferenziert

setze `p3` dereferenziert = `i`

### 3. Kodierung

```
1  #include <stdio.h>
2
3  void int_rotate( int *ap, int *bp, int *cp )
4      {
5          int i;
6          i = *ap; *ap = *bp; *bp = *cp; *cp = i;
7      }
8
9  int main( int argc, char **argv )
10     {
11         int a = 1, b = 2, c = 3;
12
13         printf( "vorher : %d %d %d\n", a, b, c );
14         int_rotate( & a, & b, & c );
15         printf( "nachher: %d %d %d\n", a, b, c );
16     }
```

# Aufgabe 4: Initialisierung von Arrays

## 1. Aufgabenstellung

Entwickle zwei Funktionen zur Initialisierung von Zeichen-Arrays. Die erste Funktion, `init_lower()` soll die Elemente des übergebenen Arrays mit den 26 Kleinbuchstaben initialisieren. Die zweite Funktion, `init_digits()` soll die Elemente des übergebenen Arrays mit den zehn Ziffern initialisieren. Ferner benötigen wir (zu Testzwecken) eine Funktion zum Ausgeben eines Zeichen-Arrays.

## 2. Pflichtenheft

Aufgabe	: je eine Funktion zum Initialisieren mit Kleinbuchstaben bzw. Ziffern und eine Funktion zum Drucken eines Zeichenarrays
Eingabe	: keine, da die Daten erzeugt werden
Ausgabe	: die Inhalte der beiden Arrays
Sonderfälle	: keine

## 3. Implementierung

Funktion zur Ausgabe eines Arrays

Wir wissen, dass beim Aufruf einer Funktion mit einem Array die Adresse des ersten Elementes übergeben wird. Daher muss der Parameter vom Typ <b>Pointer to Zeichen</b> sein. Da sich die Arrays in ihrer Größe unterscheiden können, benötigen wir noch einen zweiten Parameter <b>Integer: size</b> .
---

Funktion zur Ausgabe eines Zeichen-Arrays

Parameter: Pointer to Zeichen: array Integer: size Variable: Integer: i für i = 0 bis size-1 Schrittweite 1 wiederhole Ausgabe array[ i ]
---

Funktion zur Initialisierung eines Arrays mit Kleinbuchstaben

Die zu schreibende Funktion muss ein übergebenes Array mit Kleinbuchstaben initialisiert. Dieses Array ist der erste Parameter. Da wir dessen Größe bereits wissen, benötigen wir keinen zweiten Parameter, was aber die Flexibilität einschränkt ...
---

Funktion zur Initialisierung mit Kleinbuchstaben

Parameter: Pointer to Zeichen: array Variable: Integer: i für i = 0 bis size-1 Schrittweite 1 wiederhole setze array[ i ] = zeichen( 'a' + i )
---

#### 4. Kodierung

Unsere Lösung besteht aus den Funktionen `main()`, `init_digits()`, `init_lowers()` und `prt_array()`:

**Eine Funktion zur Ausgabe:**

```
1  #include <stdio.h>
2
3  void prt_array( char *name, char *p, int len )
4  {
5      int i;
6      printf( "kontrolle: %s:", name );
7      for( i = 0; i < len; i++ )
8          printf( " %c", p[ i ] );
9      printf( "\n" );
10 }
```

**Die Funktionen zum Initialisieren:**

```
11 void init_lowers( char *p )
12 {
13     int i = 0;
14     char c;
15     for( i = 0, c = 'a'; i < 26; i++, c++ )
16         p[ i ] = c;
17 }
18
19 void init_digits( char *p )
20 {
21     int i = 0;
22     char c;
23     for( i = 0, c = '0'; i < 10; i++, c++ )
24         p[ i ] = c;
25 }
```

**Das Hauptprogramm:**

```
26 int main( int argc, char **argv )
27 {
28     char lowers[ 26 ], digits[ 10 ];
29     int i;
30     init_lowers( lowers );
31     init_digits( digits );
32     prt_array( "lowers", lowers, 26 );
33     prt_array( "digits", digits, 10 );
34     return 0;
35 }
```

## 5. Kurzformen

Mittels der weiter oben wiederholten Kurzschreibweise `*p++` für Zeigerzugriffe können wir die Funktionen auch kompakter schreiben:

```
3 void prt_array( char *name, char *p, int len )
4     {
5         printf( "kontrolle: %s:", name );
6         while( len-- > 0 )
7             printf( " %c", *p++ );
8         printf( "\n" );
9     }
10
11 void init_lower( char *p )
12     {
13         char c;
14         for( c = 'a'; c <= 'z'; c++ )
15             *p++ = c;
16     }
17
18 void init_digits( char *p )
19     {
20         char c;
21         for( c = '0'; c <= '9'; c++ )
22             *p++ = c;
23     }
```

## 6. Stack Frame

Der folgende Speicherauszug zeigt das Arrays `digits` und den aktuelle Zustand des Stack-Frames nach dem vierten Schleifendurchlauf. Bei weiteren Unklarheiten empfehlen wir eine Handsimulation und/oder ein Gespräch mit den Betreuern.

Adresse	Variable	Wert
0xFFEE1088	.....	.....
0xFFEE1084	char digits[8-9]:	' ' ' ' X X
0xFFEE1080	char digits[4-7]:	' ' ' ' ' ' ' '
0xFFEE107C	char digits[0-3]:	'0' '1' '2' '3'
0xFFEE1078	char *p :	0xFFEE107C
0xFFEE1074	CPU PC :	„Zeile 30“
0xFFEE1070	int i :	4
0xFFEE106C	char c :	'4'

# Aufgabe 5: Generalisierte Initialisierungsfunktion

## 1. Aufgabenstellung

In der vorherigen Aufgabe haben wir zwei unterschiedliche Funktionen zum Initialisieren eines Zeichen-Arrays entwickelt. Ein genauer Blick auf die beiden Funktionen zeigt, dass sie sich sehr ähneln.

Was sind die beiden Unterschiede? Der Startwert und die Array-Größe

Ziel dieser Aufgabe ist es, beide Funktionen in eine generalisierte Initialisierungsfunktion zu überführen. Dazu müssen wir sie lediglich um zwei Parameter erweitern, die den Unterschieden Rechnung trägt. Zeige Anhand der Initialisierung mit Ziffern sowie mit Großbuchstaben, dass die neue Funktion korrekt funktioniert. Führe hierzu am besten eine Handsimulation durch.

## 2. Pflichtenheft

Aufgabe : eine generalisierte Funktion zur Initialisieren eines Zeichen-Arrays  
Parameter: drei Parameter: Anfangswert, Array, Größe  
Ausgabe : die Inhalte der beiden Arrays

## 3. Testdaten

Testdaten, wie in der Aufgabenstellung vorgegeben.

## 4. Implementierung

Generalisierte Funktion zur Initialisierung eines Arrays

Die Implementierung der generalisierten Initialisierungsfunktion ist ganz einfach. Wir müssen nur den Elementen den Initialwert zuweise bis wir das Ende des Arrays erreicht haben. Bei jeder Zuweisung müssen wir nur noch den Initialisierungswert um eins weiterschalten.

Funktion zur Initialisierung mit Kleinbuchstaben

```
Parameter: Zeichen: value
           Pointer to Zeichen: array
           Integer: size
Variable: Integer: i
für i = 0 bis size-1 Schrittweite 1
wiederhole setze array[ i ] = value
           setze value = value + 1
```

Das war's auch schon

## 5. Kodierung

### Die Funktion zum Initialisieren:

```
11 void init_array( char first, char *p, int len )
12     {
13         int i = 0;
14         for( i = 0; i < len; i++, first++ )
15             p[ i ] = first;
16     }
```

Fertig, einfach und kurz.

### Das Hauptprogramm:

```
17 int main( int argc, char **argv )
18     {
19         char uppers[ 26 ], digits[ 10 ];
20         int i;
21         init_array( '0', digits, 10 );
22         init_array( 'A', uppers, 26 );
23         prt_array( "digits", digits, 10 );
24         prt_array( "uppers", uppers, 26 );
25         return 0;
26     }
```

**Kurzform:** Unter Ausnutzung der Kurzschreibweise für Zeiger können wir die Initialisierungsfunktion wie folgt schreiben:

```
1 void init_array( char first, char *p, int len )
2     {
3         for( ; len-- > 0; first++ )
4             *p++ = first;
5     }
```

## Aufgabe 6: Berechnung von Partialsummen

### 1. Aufgabenstellung

In vielen Anwendungen liegen unterschiedliche Daten in einem Array vor. Für den Nutzer, beispielsweise einem Masterstudenten der Elektrotechnik, kann es nun interessant sein, die Daten unterschiedlich zu verknüpfen. Eine dieser Möglichkeiten ist das Bilden von Partialsummen, die jeweils einen spezifizierten Bereich des Arrays aufsummieren. In dieser Übungsaufgabe sind wir an Partialsummen interessiert, die sich jeweils symmetrisch um das mittlere Element befinden.

**Beispiel:** Daten im Array: 1 3 5 7 9, Ausgabe: 25, 15 und 5

Entwickle eine Funktion, die die entsprechende Summe der Elemente eines `int`-Arrays ermittelt. Zum Testen können obige Daten direkt in ein Array geschrieben werden. Das Hauptprogramm soll alle Partialsummen berechnen und ausgeben.

## 2. Vorüberlegungen

In der Aufgabenstellung sind schon genügend viele Hinweise für eine Implementierung gegeben. Dennoch gibt es grundsätzlich zwei unterschiedliche Möglichkeiten, die zu implementierende Funktion zu parametrisieren.

1. Wir rufen die Funktion mit drei Parametern auf: dem eigentlichen Array sowie dem Anfang und dem Ende der Partialsumme. Für die obigen Testdaten könnten diese Parameter die Werte 1 und 3 sein, sodass 15 als Ergebnis geliefert würde.
2. Wir rufen die Funktion mit nur zwei Parametern auf: dem eigentlichen Array und der Länge der Partialsumme. Jetzt müssten wir aber beim Funktionsaufruf den Anfang des Arrays „verschieben“, was wir aber bereits geübt haben.

## 3. Pflichtenheft

Aufgabe	: Schreiben einer Funktion zur Berechnung von Partialsummen, die symmetrisch zum mittleren Element eines Arrays gebildet werden.
Eingabe	: keine, die Eingabedaten direkt programmiert werden dürfen.
Ausgabe	: die jeweiligen Partialsummen gemäß Aufgabenstellung.
Parameter	: Das Daten-Array und die Zahl der zu berücksichtigenden Elemente.
Rückgabewert	: Die Funktion muss einen <code>int</code> -Wert zurück liefern.

## 4. Implementierung

Das Bilden der Partialsummen ist ebenso einfach wie das Aufsummieren der einzelnen Array-Elemente. Nur müssen wir diesmal die Parameter richtig bilden, was vermutlich die größere Herausforderung ist. Aber zuerst das Bilden der Summe:

Funktion zum Bilden einer Partialsumme

```
Parameter: Array of Integer: array
           Integer: size
Variablen: Integer: i, summe
setze summe = 0
für i = 0 bis size -1 Schrittweite 1
wiederhole setze summe = summe + array[ i ]
return summe
```

Nehmen wir an, wir haben die fünf Zahlen 1 3 5 7 9 in einem Array namens `zahlen` abgelegt. Dann könnten wir für den mittleren Fall unsere Funktion wie folgt aufrufen: `partial( zahlen + 1, 3 )` und schon sind wir fertig. In ähnlicher Weise können wir alle Partialsummen in einer geeigneten Schleife berechnen lassen.

## 5. Kodierung

Eine Funktion zum Drucken der betrachteten Array-Elemente:

```
1  #include <stdio.h>
2
3  void prt_arr( FILE *fp, int *a, int size )
4      {
5          int i;
6          for( i = 0; i < size; i++ )
7              fprintf( fp, " %d", a[ i ] );
8      }
```

Die Funktion zum Berechnen der Partialsumme:

```
9  int partial( int *a, int size )
10     {
11         int i, s;
12         for( i = s = 0; i < size; i++ )
13             s += a[ i ];
14         return s;
15     }
```

Das Hauptprogramm:

```
16 int main( int argc, char **argv )
17     {
18         int array[] = { 1, 3, 5, 7, 9 };
19         #define SIZE_A    (sizeof(array)/sizeof(array[0]))
20
21         int i, len;
22         for( i = 0, len = SIZE_A; len > 0; i++, len -= 2 )
23         {
24             printf( "partialsumme[" );
25             prt_arr( stdout, array + i, len );
26             printf( " ]: %d\n", partial( array + i, len ) );
27         }
28     }
```



## 6. Kurzformen

Mittels der weiter oben wiederholten Kurzschreibweise `*p++` für Zeigerzugriffe können wir die ersten beiden Funktionen auch kompakter schreiben:

```
1
2 void prt_arr( FILE *fp, int *a, int size )
3     {
4         while( size-- > 0 )
5             printf( " %d", *a++ );
6     }
7
8 int partial( int *a, int size )
9     {
10        int s = 0;
11        while( size-- > 0 )
12            s += *a++;
13        return s;
14    }
```

Bei weiteren Fragen stehen auch hier die Betreuer gerne zur Verfügung.

## Aufgabe 7: Bubble Sort als Funktion

Bereits in Übungspaket 15 haben wir den Bubble Sort Algorithmus entwickelt. Damals geschah alles im Hauptprogramm `main()`, was wir hier nun ändern wollen ;-)

### 1. Aufgabenstellung

Nimm den Bubble-Sort Algorithmus aus Übungspaket 15 und „kapsel“ ihn in einer Funktion. Ebenso sind die Anweisungen zum Drucken des Arrays in einer Funktion unterzubringen. Da es sich hier lediglich um das Verschieben einiger Anweisungen handelt, können wir gleich mit der Kodierung fortfahren.

### 2. Kodierung

```
1  #include <stdio.h>
2
3  void prt_array( int *p, int size )
4  {
5      while( size-- > 0 )
6          printf( " %d", *p++ );
7      printf( "\n" );
8  }
9
10 void bubble( int *p, int size )
11 {
12     int i, j, h;
13     for( i = 0; i < size - 1; i++ )
14         for( j = 0; j < size - 1; j++ )
15             if ( p[ j ] > p[ j + 1 ] )
16             {
17                 h = p[j]; p[j] = p[j + 1]; p[j + 1] = h;
18             }
19 }
20
21 int main( int argc, char **argv )
22 {
23     int a[] = { 4711, 27, 42, 2 };
24     #define SIZE_A    (sizeof(a)/sizeof(a[0]))
25
26     printf( "unsortiert:" ); prt_array( a, SIZE_A );
27
28     bubble( a, SIZE_A );
29
30     printf( " sortiert:" ); prt_array( a, SIZE_A );
31     return 0;
32 }
```

# Aufgabe 8: Suchen in Tabellen

## 1. Aufgabenstellung

Nimm den Algorithmus zum Suchen in Tabellen aus Übungspaket 15 und „kapsel“ ihn in einer Funktion `search()`. Diese Funktion benötigt die zu suchende Zahl, das Array und dessen Größe, die wir mit `size` bezeichnen. Ist die gesuchte Zahl im Array vorhanden, soll `search()` den entsprechenden Index zurückgeben. Da wir wieder nur einige Anweisungen verschieben, beginnen wir gleich mit der Kodierung.

Es bleibt jetzt aber noch die Frage zu klären, welcher Index im Fehlerfalle zurückgegeben werden soll. Dazu folgende Fragen:

Welche Indizes sind „gültig“?	0 .. size-1
Welcher Index bietet sich im Fehlerfalle an?	-1
Warum bietet sich dieser Wert an?	Er ist unabhängig von <code>size</code> .

## 2. Kodierung

```
1  #include <stdio.h>
2
3  int search( int value, int *a, int size )
4  {
5      int i;
6      for( i = 0; i < size; i++ )
7          if ( value == a[ i ] )
8              return i;                // found
9      return -1;                       // not found
10 }
11
12 int main( int argc, char **argv )
13 {
14     int i, test, a[] = { 4711, 27, 42, 2 };
15     #define SIZE_A    (sizeof(a)/sizeof(a[0]))
16
17     printf( "Bitte Testwert eingeben: " );    // input
18     scanf( "%d", & test );
19
20     if ((i = search( test, a, SIZE_A )) != -1 )
21         printf( "Testwert gefunden, Index=%d\n", i );
22     else printf( "Testwert nicht gefunden\n" );
23
24     return 0;                                // done
25 }
```

# Übungspaket 22

## Rekursive Funktionsaufrufe

---

### Übungsziele:

1. Technische Voraussetzungen für rekursive Funktionsaufrufe
2. Umsetzung mathematisch definierter Rekursionen in entsprechende C-Programme
3. Bearbeitung rekursiver Beispiele

### Skript:

Kapitel: 44 und 48 sowie insbesondere Übungspaket 19

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Funktionen haben wir nun endlich im vorherigen Übungspaket geübt. Jetzt gibt es aber noch Funktionen, die sich selbst aufrufen. Dies wird als Rekursion bezeichnet und bereitet vielen Programmieranfängern größere Schwierigkeiten, obwohl die Rekursion technisch gesehen nichts besonderes ist. Um hier Abhilfe zu schaffen, werden wir uns in diesem Übungspaket von verschiedenen Seiten der Rekursion nähern.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Rekursion am Beispiel der Fakultät

In wohl den meisten Lehrbüchern wird die Rekursion am Beispiel der Fakultät eingeführt; so haben auch wir es in unserem Skript gemacht. Wiederhole die beiden möglichen Definitionen der Fakultät:

Iterativ:  $n! = 1 \times 2 \times \cdots \times n$  für  $n \geq 0$

Rekursiv:  $n! = 1$  für  $n \leq 1$  bzw.  $n \times (n - 1)!$  für  $n > 1$

## Aufgabe 2: Technische Umsetzung der Rekursion

Was ist die wesentliche Voraussetzung für die technische Umsetzung der Rekursion in einer Programmiersprache?

Der wesentliche Punkt ist, dass bei *jedem* Funktionsaufruf ein *neuer* Stack-Frame angelegt wird, der alle formalen Parameter (mit aktuellen Werten) sowie alle lokalen Variablen beherbergt. Ferner steht in jedem Stack-Frame diejenige Stelle, an der das Programm das letzte Mal unterbrochen wurde. Dadurch kann die Bearbeitung am Ende *jedes* Funktionsaufrufs an derjenigen Stelle fortgesetzt werden, an der der reguläre Fluss unterbrochen wurde. Mit anderen Worten: Es gibt zwar nur eine Funktion im Sinne von Anweisungen aber jeder neue Funktionsaufruf arbeitet mit seinen eigenen lokalen Variablen.

Worin liegt die eigentliche Schwierigkeit der meisten Studenten?

Die Hauptschwierigkeit besteht darin, dass im Kopf nicht scharf genug zwischen den einzelnen Anweisungen und den Parametern bzw. lokalen Variablen unterschieden wird. In der Tat sind die Anweisungen nur einmal da, aber die Daten so oft, wie sich die Funktion selbst aufruft. Durch den Stack-Frame „weiß“ jeder Funktionsaufruf auch, wo er sich durch einen rekursiven Aufruf selbst unterbricht; genau dort wird er später seine Arbeit wieder aufnehmen.

Rekursive Funktionen weisen neben der Tatsache, dass sie sich selbst (direkt oder indirekt) aufrufen, ein ganz besonderes Charakteristikum auf. Welches ist das?

Die rekursiven Aufrufe müssen irgendwann aufhören, denn sonst entsteht eine „Endlosschleife“. Daher muss, wie bei einer rekursiven Definition in der Mathematik auch, der rekursive Aufruf in eine Fallunterscheidung verpackt sein. ... diese Fallunterscheidung wird oft vergessen ...

## Teil II: Quiz

---

### Regeln für die heutige Quizrunde

Diesmal geht es in den Quizfragen um das Verstehen rekursiv definierter Funktionen. Dies fällt, wie schon mehrmals gesagt, den meisten Programmieranfängern recht schwer. Eine gute Möglichkeit hier einen Schritt voranzukommen, besteht darin, dass man rekursive Funktionen mit ein paar Freunden und/oder Kommilitonen (bei einem Bier) spielt. Die Regeln sind wie folgt:

1. Jeder Mitspieler bekommt ein Blatt Papier, auf dem er seinen eigenen Stack-Frame verwaltet. Was alles steht doch gleich nochmal im Stack-Frame?
2. Einer der Mitspieler nimmt sich die Aufgabe und füllt seinen Stack-Frame aus, soweit die Informationen vorliegen.
3. Nach dem Ausfüllen des Stack-Frames beginnt der Spieler mit dem Abarbeiten des Algorithmus. Sollte er dabei auf einen rekursiven Funktionsaufruf stoßen, unterbricht er seine Arbeit und lässt diesen von einem Mitspieler bearbeiten.
4. Der neue Mitspieler erledigt alle oberen Punkte, bis er endgültig mit seiner Arbeit fertig ist. Zum Schluss übermittelt er das berechnete Ergebnis an seinen Auftraggeber, der anschließend die Kontrolle erhält.

### Aufgabe 1: Berechnung der Fakultät

**Funktion:** Die Funktion `fakultaet()` sei wie folgt definiert:

```
1 int fakultaet( int n )
2 {
3     if ( n < 2 )
4         return 1;
5     else return n * fakultaet(n - 1);
6 }
```

**Aufgabe:** Welche Resultate werden für gegebene  $n$  produziert?

$n$	1	2	3	4	5
$n!$	1	$2 \cdot (1) = 2$	$3 \cdot (2 \cdot (1)) = 6$	$4 \cdot (3 \cdot (2 \cdot (1))) = 24$	$5 \cdot (4 \cdot (3 \cdot (2 \cdot (1)))) = 120$

**Funktionalität:** Was macht `fakultaet()`? Sie berechnet die Fakultät  $n!$ .

## Aufgabe 2: Ausgabe I

**Funktion:** Die Funktion `prtb2()` sei wie folgt definiert:

```
1 void prtb2( int i )
2     {
3         if ( i >= 2 )
4             prtb2( i / 2 );
5         printf( "%d", i % 2 );
6     }
```

**Aufgabe:** Welche Ausgaben werden für gegebene  $n$  produziert?

$n$	1	2	3	4	5	6
<code>prtb2( n )</code>	1	10	11	100	101	110

**Funktionalität:** Was macht `prtb2( n )`? Ausgabe der Zahl  $n$  im Binärformat (Basis 2)

## Aufgabe 3: Ausgabe II

**Funktion:** Die Funktion `display()` sei wie folgt definiert:

```
1 void display( int i )
2     {
3         if ( i != 0 )
4         {
5             printf( "%d", i % 2 );
6             display( -(i/2) );
7         }
8     }
```

**Aufgabe:** Welche Ausgaben werden für gegebene  $n$  produziert?

$n$	15	63	255
<code>display( n )</code>	1-11-1	1-11-11-1	1-11-11-11-1

**Funktionalität:** Was macht `display( n )`? Ausgabe im gespiegelten Binärformat

## Aufgabe 4: Ausgabe III

**Funktion:** Die Funktion `triline()` sei wie folgt definiert:

```
1 void triline( int i, int nr, int max )
2     {
3         if ( i == 0 && nr < max - 1 )
4             triline( 0, nr + 1, max );
5         if ( i < max - nr - 1 || i > max + nr - 1 )
6             printf( " " );
7         else printf( "%d", nr );
8         if ( i == 2*(max - 1) )
9             printf( "\n" );
10        else triline( i + 1, nr, max );
11    }
```

**Aufgaben:**

Aufruf	<code>triline(0,0,1)</code>	<code>triline(0,0,2)</code>	<code>triline(0,0,3)</code>
Ausgabe	0	111	22222
		0	111
			0

**Funktionalität:** Was macht `triline()`? Sie druckt Dreiecke, die auf der Spitze stehen



## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler bei der Summenberechnung

Eine alte mathematische Aufgabe ist die Berechnung der Summe  $s = \sum_{i=1}^n$ , für die der kleine Gauß eine schöne Formel gefunden hat. Starmathematiker DR. G. AUSS schlägt folgende Lösung vor:

```
1 int summe( int n )
2     {
3         return n + summe(n - 1);
4     }
```

Doch was ist hier falsch? Beschreibe den Fehler und korrigiere das Programm entsprechend.

Die Idee ist ja eigentlich richtig. Nur fehlt eine geeignete Fallunterscheidung, damit die Rekursion (der rekursive Abstieg) abbricht. Daher lautet eine geeignete Lösung wie folgt:

```
1 int summe( int n )
2     {
3         if ( n <= 0 )
4             return 0;
5         else return n + summe(n - 1);
6     }
```

Dies ist neben den „normalen“ Programmierfehlern der einzige substantielle, der im Zusammenhang mit der Rekursion gemacht werden kann.

Ein zweiter, zumindest beliebter Fehler ist, dass man die eigenen Aktionen und den rekursiven Abstieg nicht in der richtigen Reihenfolge vornimmt.

Aber da das schon alles ist, machen wir lieber mit den Anwendungen weiter.

## Teil IV: Anwendungen

---

In jeder der folgenden Aufgaben ist eine Funktion zu implementieren, die eine gestellte Aufgabe *rekursiv* lösen soll. Ihr solltet eure erarbeiteten Lösungen natürlich eintippen und mittels eines einfachen Hauptprogramms auch testen.

### Aufgabe 1: Fibonacci-Zahlen

Schreibe eine Funktion, die die Fibonacci-Zahlen von 0 bis 10 berechnet. Die Fibonacci-Zahlen  $F_i$  sind wie folgt definiert:  $F_0 = 0$ ,  $F_1 = 1$  und  $F_n = F_{n-1} + F_{n-2}$

```
1 int fibonacci( int n )
2     {
3         if ( n < 2 )
4             return (n < 1)? 0: 1;
5         else return fibonacci(n - 1) + fibonacci(n - 2);
6     }
```

### Aufgabe 2: Ausgabe eines int-Arrays

Gegeben sei ein `int`-Array der Größe `size`. Schreibe eine Funktion `print_array()`, die die einzelnen Elemente rekursiv ausgibt. Beispiel: Sollte ein Array der Größe 4 die Elemente 0, 1, 815 und 4711 enthalten, sollte die Ausgaben 0, 1, 815 und 4711 lauten.

```
1 int print_array( int *array, int n )
2     {
3         if ( n > 1 )
4         {
5             printf( "%d ", array[ 0 ] );
6             print_array( array + 1, n - 1 );
7         }
8         else printf( "%d\n", array[ 0 ] );
9     }
```

### Aufgabe 3: Gespiegelte Ausgabe eines int-Arrays

Schreibe eine zweite Funktion `print_array_reverse()`, die die Array-Elemente von hinten nach vorne ausgibt. Bei obigem Beispiel soll die Ausgabe 4711, 815, 1 und 0 lauten.

```

1 int print_array_reverse( int *array, int n )
2 {
3     if ( n > 1 )
4         print_array_reverse( array + 1, n - 1 );
5     printf( "%d ", array[ 0 ] );
6 }

```

## Aufgabe 4: Ab und auf

Schreibe eine Funktion, die die Zahlen hinab bis 0 und wieder zurück ausgibt. Beispiel: Das Argument 3 soll dazu führen, dass die Zahlen 3 2 1 0 1 2 3 ausgegeben werden.

```

1 int ab_und_auf( int n )
2 {
3     printf( "%d ", n );
4     if ( n > 0 )
5     {
6         ab_und_auf( n - 1 );
7         printf( "%d ", n );
8     }
9 }

```

```

1 int ab_und_auf( int n )
2 {
3     printf( "%d ", n );
4     if ( n <= 0 )
5         return;
6     ab_und_auf( n - 1 );
7     printf( "%d ", n );
8 }

```

## Aufgabe 5: Auf und ab

Schreibe eine Funktion, die die Zahlen hinauf bis zu einem gegebenen Zahlenwert und wieder zurück ausgibt. Eine beispielhafte Ausgabe lautet: 0 1 2 3 2 1 0

```

1 int auf_ab( int n, int max )
2 {
3     printf( "%d ", n );
4     if ( n < max )
5     {
6         auf_ab( n+1, max );
7         printf( "%d ", n );
8     }
9 }

```

```

1 int auf_ab( int n, int max )
2 {
3     printf( "%d ", n );
4     if ( n >= max )
5         return;
6     auf_ab( n + 1, max );
7     printf( "%d ", n );
8 }

```

Der wesentliche Trick ist hier, dass man der Funktion zwei Argumente mitgibt, damit sie „merkt“, dass das Ende des rekursiven Abstiegs erreicht ist.

## Tutorial: Binäres Suchen in Tabellen

In den vorherigen Übungspaketen haben wir das Thema „Suchen in Tabellen“ immer mal wieder aufgegriffen. Ein Standardthema in diesem Bereich ist die binäre Suche in *sortierten*

*Tabellen.* Obwohl das Prinzip der binären Suche recht einfach ist, liegen die Tücken, wie so oft, im Detail. Um alles richtig zum Laufen zu bekommen, benötigt man selbst als fortgeschrittener Programmierer Einiges an Zeit. Da wir aber noch andere, ebenso wichtige Themen in diesem Semester behandeln wollen, haben wir uns entschlossen, hier ein kleines Tutorial zu diesem Thema zu präsentieren, was jeder zumindest einmal durchgehen sollte.

## 1. Motivation und Einführung

Ein charakteristisches Merkmal der bisherigen Suche war, dass sie in unsortierten Tabellen vorgenommen wurde. Sofern das gesuchte Element vorhanden ist, benötigt man im ungünstigsten Fall  $n$ , im Durchschnitt  $n/2$  Vergleiche. Wenn die Tabelle aber bereits sortiert ist, kann man die Suche auf  $\log n$  Vergleiche reduzieren, was bei sehr großen Tabellen eine deutliche Geschwindigkeitssteigerung zur Folge haben kann.

*„Aber wieso soll die Zahl der Vergleiche nicht mehr linear mit  $n$  sondern nur noch logarithmisch mit  $\log n$  wachsen? Wir haben doch immer noch  $n$  Elemente?“* Richtig :-)! Aber, der wesentliche Punkt obiger Bemerkung ist, dass die Tabelle sortiert ist. In diesem Falle können wir in der Mitte der Tabelle nachschauen und wissen sofort, ob wir in der linken oder rechten Hälfte weiter suchen müssen.

*Aha, wir können uns sofort für eine Hälfte entscheiden. Aber dann sind wir doch nur doppelt so schnell?“* Nee ;-)! im ersten Schritt halbieren wir die Tabellengröße. Wenn die gesamte Tabelle sortiert ist, dann sind es auch beide Hälften und wir können im nächsten Schritt die Größe einer der beiden Hälften wieder halbieren. . . was dann ein Viertel macht, danach ein achtel, ein sechzehntel usw. Da auch die weiteren Schritte prinzipiell gleich sind, klingt das nach Rekursion. In welchem Übungspaket waren wir doch gerade . . . ? Genau. Im Umkehrschluss bedeutet dies: sollte die Tabelle *doppelt* so groß sein, benötigen wir nur einen einzigen Schritt mehr. Und das soll auch in *C* klappen? Schauen wir mal.

Aber bevor wir weiter machen, sollten wir uns nochmals folgendes in Erinnerung rufen: Falls unsere Funktion `binarySearch()` das gesuchte Element gefunden hat, soll sie den entsprechenden Tabellenindex zurückgeben, andernfalls eine `-1`.

## 2. Ein paar Beispiele

Um die Sache halbwegs einfach zu gestalten, schauen wir uns hier ein paar Beispiele an. Dabei geht es erst einmal nur um das Verstehen des Algorithmus; etwaige Verbesserungen besprechen wir später. Ferner verzichten wir erst einmal auf alle programmiertechnischen Details. Nehmen wir an, wir haben eine Tabelle mit den folgenden acht Elementen:

5	9	12	18	24	27	41	53
---	---	----	----	----	----	----	----

#### Suche nach 12:

5	9	12	18	24	27	41	53
5	9	12	18				
		12	18				
		12					

#### Suche nach 27:

5	9	12	18	24	27	41	53
				24	27	41	53
				24	27		
					27		

Beide Beispiele zeigen folgende Dinge: Nach dreimaligem Halbieren haben wir nur noch ein einziges Element übrig. Dieses Element können wir sehr einfach hinsichtlich der gesuchten Zahl überprüfen. Und schon sind wir fertig. Interessant ist auch noch festzuhalten, dass wir beim Halbieren am Ende immer nur ein Element übrig haben, egal wie viele Elemente wir anfangs haben.

### 3. Implementierung: ein erster Versuch

Aus dem vorangegangenen Beispiel wissen wir nun folgende Dinge:

1. Sofern wir nur noch ein Element haben, sind wir mit der Suche fertig. Wir vergleichen dieses eine Element mit dem gesuchten und geben entweder eine -1 oder den Index zurück.
2. Haben wir mehr als ein Element, sind wir mit der Suche noch nicht fertig. Hierzu müssen wir entscheiden, ob wir links oder rechts weiter suchen müssen. Für diese Entscheidung vergleichen wir das gesuchte Element mit dem letzten Element der linken Hälfte oder mit dem ersten Element der rechten Hälfte.

Ein erster algorithmischer Ansatz sieht wie folgt aus:

#### Funktion Binäre Suche

```
Parameter Integer: token, size
          Array of Integer: such_feld

wenn size = 1
dann wenn token = such_feld[ 0 ]
    dann return index
    sonst return -1
sonst wenn kleiner_rechte_Hälfte( token, such_feld )
    dann return Binäre Suche(token, linke Hälfte von such_feld)
    sonst return Binäre Suche(token, rechte Hälfte von such_feld)
```

Natürlich fehlen jetzt noch einige Dinge, insbesondere die Bestimmung der beiden Hälften und des richtigen Wertes für den Index. Insbesondere letzteres erfordert ein wenig Gehirnschmalz, da wir als eine wesentliche Entwurfsentscheidung den üblichen Funktionskopf `Funktion( token, array, size )` beibehalten wollen.

#### 4. Bestimmung der Hälften und des Index

Fangen wir mit der Bestimmung der „Hälfte“ an: `haelfte = size / 2`. Für den Fall `size == 8` hätten wir `haelfte == 4`, linke Hälfte von `0..haelfte-1` und die rechte Hälfte von `haelfte..size-1`. Also, erst einmal alles prima.

Nun müssen wir noch den *richtigen* Index berechnen. „Warum das denn? Die 12 hat doch den Index 2, wie wir ganz klar sehen.“ Ja, richtig, aber das Problem ist subtil: In der Funktion hat das Array nur noch die Größe 1 und damit würde sie immer den Index 0 zurückgeben. Aber das wollen wir ja nicht ;-). Die einfachste Lösung ist, dass wir immer den Index des ersten Elementes unseres aktuellen Arrays mit übergeben. Somit kommen wir zu folgender zweiten Version:

Funktion Binäre Suche

```
Parameter Integer: token, size, offset
          Array of Integer: such_feld

setzte haelfte = size / 2
wenn size = 1
dann wenn token = such_feld[ 0 ]
    dann return offset // index ist ja null
    sonst return -1
sonst wenn token < such_feld[ haelfte ]
    dann return Binäre Suche( token, such_feld,
                                haelfte, offset )
    sonst return Binäre Suche( token, such_feld + haelfte,
                                haelfte, offset + haelfte )
```

#### 5. Ist die Hälfte wirklich die Hälfte?

„Der Algorithmus gefällt mir, aber haben wir da nicht einen Fehler gemacht?“ Wieso? Bei der Division von `int`-Werten kann es zu Rundungsfehlern kommen und dann ist `size/2 + size/2` nicht mehr `size`, zumindest bei ungeraden Zahlen.“ Stimmt, sehr gut beobachtet! Am besten rekapitulieren wir das anhand eines Beispiels mit `size == 3`. Falls wir in der linken Hälfte suchen, erhält die Suchfunktion einen Zeiger auf das erste Element (Index 0) und eine Länge von 1, denn `size/2 = 3/2` ergibt 1. Sollten wir in der rechten Hälfte suchen, bekommen wir einen Zeiger auf das zweite Element (Index 1) aber auch nur eine Länge `haelfte = 1`. „Genau, dann geht das dritte Element verloren.“ Genau! Aufgrund des Rundungsfehlers ist die Länge der rechten Hälfte nicht 2 sondern lediglich `haelfte = size/2 = 1`. Wir müssen nun dafür sorgen, dass als Länge der rechten Hälfte die korrekte Zahl übergeben wird, was wir mittels `size - haelfte = size - size/2 = 2` einfach hinbekommen.

Nun haben wir die wichtigsten Dinge beisammen und können daraus ein C-Programm erstellen, was wir auf der nächsten Seite erledigen.

## 6. Kodierung

Unsere Suchfunktion:

```
1 int search( int token, int *tp, int size, int offset )
2 {
3     int ind, half = size / 2;
4     if ( size == 1 )
5         ind = (*tp == token)? offset: -1;
6     else if ( token < tp[ half ] )
7         ind = search( token, tp, half, offset );
8     else ind = search( token, tp + half,
9                       size - half, offset + half );
10    return ind;
11 }
```

Ein Hauptprogramm zum Testen:

Nun benötigen wir noch ein Hauptprogramm zum Testen unserer neuen Funktion. Zunächst erstellen wir eine Tabelle `table[]`, in die wir unsere Zahlen eintragen. Ferner benötigen wir noch Daten zum Testen. Dazu können wir einerseits das selbe Feld `table[]` nehmen, aus dem wir die zu suchenden Zahlen schrittweise entnehmen. Ferner bauen wir noch eine zweite Tabelle `fail[]`, in die wir Zahlen eintragen, die nicht in `table[]` zu finden sind. Aus dieser zweiten Tabelle entnehmen wir wiederum die Zahlen und suchen diese in der Tabelle `table[]`.

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6
7     int table[] = { 0, 2, 4, 6, 8, 16, 32, 64 };
8     #define TAB_SIZE (sizeof(table)/sizeof(*table))
9
10    int fail[] = { 1, 4711, 815 };
11    #define F_SIZE    (sizeof(fail)/sizeof(*fail))
12
13    for( i = 0; i < TAB_SIZE; i++ )
14        printf( "Wert=%d Index=%d\n", table[ i ],
15              search( table[i], table, TAB_SIZE, 0 ) );
16
17    for( i = 0; i < F_SIZE; i++ )
18        printf( "Wert=%d Index=%d\n", fail[ i ],
19              search( fail[i], table, TAB_SIZE, 0 ) );
20 }
```

## 7. Nachlese

Schön ist, dass die Funktion sowie das Programm funktionieren. Ein wenig unschön ist, dass die Funktion `search()` einen weiteren Parameter benötigt, damit der Index richtig berechnet wird. Diesen Index müssen wir beim ersten Aufruf (im Hauptprogramm) auch mit null initialisieren, was man zusätzlich wissen muss.

Durch ein paar kleine Modifikationen kann man sich dieses Problems entledigen. Der Grund, warum wir diese Variante erst jetzt präsentieren ist, dass wir erst einmal die Idee des binären Suchens vollständig erläutern wollten.

```
1 int search( int token, int *tp, int size )
2     {
3         int ret, half = size / 2;
4         if ( size == 1 )
5             return (*tp == token)? 0: -1;
6         if ( token < tp[ half ] )
7             return search( token, tp, half );
8         ret = search( token, tp + half, size - half );
9         return ( ret == -1 )? -1: half + ret;
10    }
```

Der Aufbau der Funktion ist wieder der gleiche. Für den Fall, dass wir nur noch ein Element haben, geben wir entweder 0 für „gefunden“ oder -1 für „nicht gefunden“ zurück. Im zweiten Teil suchen wir im linken Teil der Tabelle. Hier geben wir den Index, den wir erhalten, einfach weiter. Im dritten Teil müssen wir aber unterscheiden: Bekommen wir den Rückgabewert -1, ist die gesuchte Zahl nicht in diesem Tabellenteil und wir müssen diesen Index weiterreichen. Sollten wir aber einen „gültigen“ Index erhalten haben, müssen wir diesen noch um den Betrag `half` korrigieren, da wir uns ja in der rechten Hälfte befinden.



# Übungspaket 23

## Mehrdimensionale Arrays

---

### Übungsziele:

Deklaration und Verwendung mehrdimensionaler Arrays

### Skript:

Kapitel: 49

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Bevor wir mit komplizierteren Themen wie Zeichenketten weiter machen, müssen wir noch „eben“ eine kleine Übung zu mehrdimensionalen Arrays nachschieben. Mehrdimensionale Arrays sind eine einfache Erweiterung der eindimensionalen Arrays und eigentlich jedem aus der Mathematik als Matrizen bekannt. Insofern erwarten wir keine wesentlichen Probleme ;-)

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Wiederholung eindimensionaler Arrays

Aus welchen Komponenten besteht eine Array-Definition (ohne Initialisierung)?

- |                       |                                   |
|-----------------------|-----------------------------------|
| 1. Elementtyp         | 2. Array-Name                     |
| 3. Eckige Klammern [] | 4. Array-Größe (innerhalb der []) |

Welche Indizes sind gültig, wenn wir ein Array der Größe  $n$  (z.B. 14) haben? `0 .. n-1`

## Aufgabe 2: Aufbau mehrdimensionaler Arrays

Wie werden die einzelnen Dimensionen eines mehrdimensionalen Arrays definiert?

Bei der Definition eines mehrdimensionalen Arrays muss die Größe jeder einzelnen Dimension gesondert in eckigen Klammern angegeben werden.

Wie wird auf die einzelnen Elemente eines mehrdimensionalen Arrays zugegriffen?

Ein einzelnes Element erhält man, in dem man für jede einzelne Dimension einen konkreten Wert angibt.

Wie werden die Elemente eines mehrdimensionalen Arrays in der Programmiersprache C im Arbeitsspeicher abgelegt?

Die einzelnen Elemente werden „zeilenweise“ im Arbeitsspeicher abgelegt. Beispiel:  
`a[0][0], a[0][1], ..., a[0][m-1], a[1][0], ..., a[n-1][0], ..., a[n-1][m-1]`

## Aufgabe 3: Speicherbelegung am Beispiel

Nehmen wir an, wir hätten folgende Matrize:  $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$

Wie kann eine derartige Matrize in C umgesetzt werden? `int A[ 2 ][ 3 ]`

Welche Indizes sind gültig? `A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2]`

In welcher Reihenfolge werden die einzelnen Elemente im Arbeitsspeicher abgelegt?

`A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2]`

Wie werden die Parameter  $a_{ij}$  im Speicher angeordnet? `a11, a12, a13, a21, a22, a23`

## Teil II: Quiz

---

### Aufgabe 1: Definition mehrdimensionaler Arrays

In der folgenden Tabelle sind einige Definitionen vorgegeben. Das erste Beispiel ist bereits vorgegeben; vervollständige die freien Spalten:

Definition	Dimensionen	Größe in Bytes	gültige Indizes
char c[2][4][2]	.....3	.....16	$(0..1) \times (0..3) \times (0..1)$
char c[3][3]	.....2	.....9	$(0..2) \times (0..2)$
char c[2][2][2][2]	.....4	.....16	$(0..1) \times (0..1) \times (0..1) \times (0..1)$
char c[1][2][1]	.....3	.....2	$(0..0) \times (0..1) \times (0..0)$
char c['z'-'a'][2]	.....2	.....50	$(0..24) \times (0..1)$
char c['z'-'a'][1]	.....2	.....25	$(0..24) \times (0..0)$
char c[3][7][2][3]	.....4	.....126	$(0..2) \times (0..6) \times (0..1) \times (0..2)$

### Aufgabe 2: Implizit definierte Arrays

Die Größe eines Arrays kann man auch „implizit“ durch eine zusätzliche Initialisierung definieren. Gegeben seien die folgenden drei Definitionen:

1. `int a[] [] = {{0, 1, 2}, {3, 4, 5}};`
2. `int b[][ 2 ] = {{0, 1}, {2, 3}, {4, 5}};`
3. `int c[][ 2 ] = {0, 1, 2, 3, 4, 5};`

Vervollständige die folgende Tabelle:

Array	„explizite“ Definition
a	<code>int a[ 2 ][ 3 ]</code>
b	<code>int b[ 3 ][ 2 ]</code>
c	<code>int c[ 3 ][ 2 ]</code>

## Teil III: Fehlersuche

---

### Aufgabe 1: Arbeiten mit mehreren Zahlen

Die folgenden Programmzeilen bewirken nichts sinnvolles, sondern dienen lediglich dem Einüben mehrdimensionaler Arrays. Doch auch hier hat Freizeitwindsurfer DR. SURF ein paar kleine Fehler gemacht. Finde, beschreibe und korrigiere diese.

```
1 int i;
2 int a[ 2, 3, 4 ];
3 int b[ 3 ][4 ; 5 ];
4 int c[ 2 ][ 2 ];
5 i = a[ 1, 1, 1 ];
6 i = b[ 3 ][ 4 ][ 5 ];
7 i = c[ 1 ][ 1 ][ 1 ];
```

Zeile	Fehler	Erläuterung	Korrektur
2	, statt []	Jede einzelne Dimension muss gesondert, in eckigen Klammern stehend, definiert werden.	a[2][3][4]
3	; statt []	Jede einzelne Dimension muss gesondert, in eckigen Klammern stehend, definiert werden.	b[3][4][5]
5	, statt []	Auch bei Zugriff auf die einzelnen Elemente muss jede einzelne Dimension in eckigen Klammern notiert werden.	a[1][1][1]
6	falsche Indizes	Die angegebenen Indizes sind ausserhalb der erlaubten Grenzen. Eine beispielhafte Korrektur verwendet jeweils die maximalen Werte.	b[2][3][4]
7	falsche Dimensionen	Das Array c hat nur zwei und nicht drei Dimensionen.	c[1][1]

#### Programm mit Korrekturen:

```
1 int i;
2 int a[ 2 ][ 3 ][ 4 ];
3 int b[ 3 ][ 4 ][ 5 ];
4 int c[ 2 ][ 2 ];
5 i = a[ 1 ][ 1 ][ 1 ];
6 i = b[ 2 ][ 3 ][ 4 ];
7 i = c[ 1 ][ 1 ];
```

## Teil IV: Anwendungen

---

In den ersten drei Aufgaben üben wir das Definieren und Initialisieren mehrdimensionaler Arrays. Anschließend diskutieren wir, wie mehrdimensionale Matrizen als Parameter an Funktionen übergeben werden können. Den Abschluss bildet eine kleine Anwendung.

### Aufgabe 1: Deklaration mehrdimensionaler Arrays

Vervollständige in der folgenden Tabelle die fehlenden Definitionen:

Elementtyp	Größe	Deklaration
int	$3 \times 2 \times 4$	int a[ 3 ][ 2 ][ 4 ]
char	$2 \times 10$	char b[ 2 ][ 10 ]
double	$4 \times 1 \times 5 \times 2$	double d[ 4 ][ 1 ][ 5 ][ 2 ]

### Aufgabe 2: Implizit definierte Arrays

1. Definiere ein zweidimensionales Array mit zwei Zeilen à drei Spalten, in denen die Zahlen 1, 2, 3, 4, 5 und 6 stehen:

```
int a[ ][ 3 ] = {{1, 2, 3 }, {4, 5, 6 }};
```

2. Definiere ein zweidimensionales Array mit drei Zeilen à zwei Spalten, in denen der Reihe nach die Buchstaben a bis f stehen:

```
char c[ ][ 2 ] = {{ 'a', 'b' }, { 'c', 'd' }, { 'e', 'f' }};
```

3. Definiere ein Array mit drei Zeilen à drei Spalten, in denen überall die Zahl 1.0 steht:

```
double d[ ][ 3 ] = {{1.0, 1.0, 1.0 }, {1.0, 1.0, 1.0 }, {1.0, 1.0, 1.0 }};
```

### Aufgabe 3: Zugriff auf mehrdimensionale Arrays

1. Entwickle ein kleines Programm, das eine  $3 \times 3$ -Matrix definiert und wie folgt initialisiert: Die Diagonalelemente sollen als Wert die Zeilennummer erhalten, alle anderen den Wert Null. Zur Eigenkontrolle soll das Programm diese Matrix ausgeben.

Beispiel: 
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

```

1  #include <stdio.h>
2
3  #define N      3
4
5  int main( int argc, char **argv )
6  {
7      int i, j, a[ N ][ N ];
8      for( i = 0; i < N; i++ )
9          for( j = 0; j < N; j++ )
10             a[ i ][ j ] = (i == j)? i + 1: 0;
11      for( i = 0; i < N; i++ )
12      {
13          for( j = 0; j < N; j++ )
14              printf( "%d ", a[ i ][ j ] );
15          printf( "\n" );
16      }
17  }

```

2. Schreibe ein Programm, das eine  $4 \times 1 \times 2 \times 1$ -Matrix definiert, in der alle Elemente den Wert 1.0 haben. Zur Eigenkontrolle soll das Programm diese Matrix ausgeben.

```

1  #include <stdio.h>
2
3  #define N      4
4  #define M      1
5  #define K      2
6  #define L      1
7
8  int main( int argc, char **argv )
9  {
10     double d[ N ][ M ][ K ][ L ];
11     int i, j, k, l;
12     for( i = 0; i < N; i++ )
13         for( j = 0; j < M; j++ )
14             for( k = 0; k < K; k++ )
15                 for( l = 0; l < L; l++ )
16                     d[ i ][ j ][ k ][ l ] = 1.0;
17     for( i = 0; i < N; i++ )
18         for( j = 0; j < M; j++ )
19             for( k = 0; k < K; k++ )
20                 for( l = 0; l < L; l++ )
21                     printf( "%3.1f ", d[i][j][k][l] );
22     printf( "\n" );
23 }

```

## Aufgabe 4: Mehrdimensionale Arrays als Parameter

Mehrdimensionale Arrays können in ihrer generischen Form leider nicht so einfach als Parameter an Funktionen übergeben werden, wie wir dies bei eindimensionalen Parametern kennengelernt haben. Bevor du weiter liest, überlege dir zunächst wie dies bei eindimensionalen Arrays war und warum es bei mehrdimensionalen Arrays nicht so direkt geht.

Bei eindimensionalen Arrays haben wir immer den Array-Namen, der die Adresse des ersten Elementes repräsentiert, sowie die Array-Größe übergeben. Und schon war alles gut, da wir die einzelnen Elemente nacheinander im Speicher vorgefunden haben.

Mehrdimensionale Arrays werden auch elementweise im Speicher abgelegt, doch müssen wir mehrere Dimensionen berücksichtigen. Zur Erinnerung: eine  $2 \times 3$ -Matrix hat folgendes Speicherabbild:

m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]
---------	---------	---------	---------	---------	---------

Gegeben sei die Definition: `char c[ 3 ][ 3 ]`. Vervollständige folgende Tabelle:

Ausdruck	Resultat	Ausdruck	Resultat
<code>sizeof(c[ 1 ][ 0 ])</code>	1	<code>sizeof(c[ 0 ])</code>	3
<code>sizeof(c[ 0 ][ 0 ])</code>	1	<code>sizeof(c[ 1 ])</code>	3
<code>sizeof(c[ 4 ][ 5 ])</code>	1	<code>sizeof(c)</code>	9

Gegeben sei nun folgendes Programm:

```
1  #include <stdio.h>
2
3  char f( char m[][], int size )
4      {
5      }
6
7  int main( int argc, char **argv )
8      {
9          char c[ 3 ][ 3 ];
10         f( c, 9 );
11     }
```

Hierzu haben wir nun folgende Fragen:

Ist die Syntax dieses Programms korrekt?

Nein.

Falls nein, welche Zeilen sind problematisch?

Die Zeilen 3 und 10.

Falls nein, worin liegt das Problem?

Die Parameter `m` und `c` sind inkompatibel.

Welchen Typ hat der Parameter `c`?

`char (*)[ 3 ]`  $\Rightarrow$  Zeiger auf drei `char`

Was wird der Compiler anmeckern und was bzw. wie können wir das Problem beheben?

Die beiden Matrizen (Parameter) `c` und `m` sind inkompatibel bzw. der Parameter `m` ist nicht vollständig spezifiziert. Da es sich um ein zweidimensionales Array handelt, muss der Compiler die Zeilenlänge (also die Zahl der Spalten) wissen, denn sonst weiß er nicht, wo `c[1]` im Arbeitsspeicher anfängt.

Wir müssen also angeben, wie lang eine Zeile ist. Dies erreichen wir mittels der Definition `char f( char m[][ 3 ], int size )`, in der der Parameter `size` die Zahl der Spalten spezifiziert. Wir haben zwar nun die Möglichkeit, die Funktion `f()` mit Matrizen unterschiedlicher Zeilenzahl aufzurufen, doch müssen sie alle eine Zeilenlänge von *genau* drei Zeichen besitzen.

Eine interessante Frage bleibt nun, ob wir nicht dennoch eine Funktion schreiben können, die beliebige *quadratische* Matrizen initialisieren kann? Die Antwort ist recht einfach: Wir wissen, wie auch ein mehrdimensionales Array im Arbeitsspeicher abgelegt wird und wir wissen, wie wir das erste Element eines beliebigen Arrays erhalten. Um beides zusammenzubringen, können wir innerhalb der Funktion ein eindimensionales Array betrachten, das wir durch eigene Berechnungen rekonstruieren. Lange Rede kurzer Sinn, hier kommt ein kleines Beispiel zur Initialisierung von  $n \times n$ -Diagonalmatrizen:

```
1  #include <stdio.h>
2
3  void dig_init( int *p, int n )
4  {
5      int i, j;
6      for( i = 0; i < n; i++ )
7          for( j = 0; j < n; j++ )
8              p[ i + j * n ] = (i == j)? 1: 0;
9  }
10
11 int main( int argc, char **argv )
12 {
13     #define SIZE 3
14     int a[ SIZE ][ SIZE ];
15     int i, j;
16     dig_init( & a[ 0 ][ 0 ], SIZE );
17     for( i = 0; i < SIZE; i++ )
18     {
19         for( j = 0; j < SIZE; j++ )
20             printf( "%d ", a[ i ][ j ] );
21         printf( "\n" );
22     }
23 }
```



## Aufgabe 5: Matrixmultiplikation

Als kleine Zusatz- bzw. Abschlussaufgabe beschäftigen wir uns mit der Multiplikation einer Matrix mit einem Vektor.

### 1. Vorbetrachtungen

In der Mathematik ist das Resultat  $R$  der Multiplikation einer Matrix  $A$  mit einem Vektor  $V$  wie folgt definiert:

$$A = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}, \quad V = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix}, \quad R = \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

$$R = A \cdot V \quad \text{mit} \quad r_i = \sum_{k=1}^m a_{ik} v_k$$

### 2. Aufgabenstellung 1

Entwickle eine Funktion `mat_mul()`, die eine  $n \times m$  Matrix  $A$  mit einem Vektor  $V$  der Länge  $m$  multipliziert und das Ergebnis in einem Vektor  $R$  der Länge  $n$  ablegt. In der ersten Variante soll die Funktion `mat_mul()` „wissen“, dass die Zeilenlänge der Matrix  $A$  genau  $m$  ist. Als Elementtyp soll `double` verwendet werden.

### 3. Pflichtenheft

Aufgabe	: Entwicklung einer Funktion zur Multiplikation einer $n \times m$ Matrix mit einem Vektor
Eingabe	: keine, da ein Testfall vorgegeben wird
Ausgabe	: das Ergebnis der Multiplikation
Sonderfälle	: keine
Funktionsköpfe	: <code>void mat_mul(double a[][ M ], double v[], double r[])</code>

### 4. Testdaten

Zum Test unseres Programm können wir das folgende Beispiel verwenden:

$$R = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 6.0 \end{bmatrix}$$

### 5. Implementierung

Da wir die einzelnen Definitionen und Algorithmen bereits besprochen haben, können wir gleich mit der Kodierung fortfahren.

## 6. Kodierung

Unsere beispielhafte Kodierung sieht wie folgt aus:

```
1  #include <stdio.h>
2
3  #define N      3
4  #define M      3
5
6  void mat_mul( double a[][ M ], double v[], double r[] )
7  {
8      int i, j;
9      for( i = 0; i < N; i++ )
10         for( r[ i ] = 0.0, j = 0; j < M; j++ )
11             r[ i ] += a[ i ][ j ] * v[ j ];
12 }
13
14 void prt_vec( double v[], int size )
15 {
16     int i;
17     for( i = 0; i < size; i++ )
18         printf( "%4.1f ", v[ i ] );
19     printf( "\n" );
20 }
21
22 void prt_mat( double v[][M], int size )
23 {
24     int i;
25     for( i = 0; i < size; i++ )
26         prt_vec( v[ i ], M );
27 }
28
29 int main( int argc, char **argv )
30 {
31     double a[][M] = {{ 1.0, 0.0, 0.0 }, { 1.0,
32                                     1.0, 0.0 }, { 1.0, 1.0, 1.0 }};
33     double v[] = { 1.0, 2.0, 3.0 }, r[ N ];
34     mat_mul( a, v, r );
35     printf( "Matrix A:\n" ); prt_mat( a, M );
36     printf( "Vektor V: " );  prt_vec( v, M );
37     printf( "Vektor R: " );  prt_vec( r, N );
38 }
```

**Anmerkungen:** Die Funktion `prt_vec()` dient nur der Ausgabe eines Vektors. Da in unserem Fall die Matrix A aus N Vektoren der Länge M besteht, können wir diese Funktion auch in `prt_mat()` verwenden.

## 7. Aufgabenstellung 2

Das Ziel dieser letzten Teilaufgabe ist es, die Funktionen `mat_mul()` und `prt_mat()` so umzuschreiben, dass sie für beliebige zweidimensionale Matrizen funktionieren. Dabei können wir auf die Ergebnisse der Aufgabe 4 zurückgreifen, die bereits diskutiert hat, wie mittels der Größenparameter die zweidimensionale Struktur einer Matrice rekonstruiert werden kann.

## 8. Pflichtenheft

Aufgabe : Verallgemeinerung der beiden bereits entwickelten Funktionen `mat_mul()` und `prt_mat()`

Funktionsköpfe: `void mat_mul( double *a, double *v, double *r,`  
`int n, int m )`  
`void prt_mat( double *v, int n, int m )`

## 9. Testdaten

Zum Test unseres Programm können wir wieder das folgende Beispiel verwenden:

$$R = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 6.0 \end{bmatrix}$$

## 10. Implementierung

Da wir die einzelnen Definitionen und Algorithmen bereits besprochen haben, können wir gleich mit der Kodierung fortfahren. Die bestehende Implementierung müssen wir nur um die korrekte Adressberechnung erweitern.

## 11. Kodierung

Unsere beispielhafte Kodierung sieht wie folgt aus:

```
1  #include <stdio.h>
2
3  #define N      3
4  #define M      3
5
6  void mat_mul( double *a, double *v, double *r,
7               int n, int m )
8  {
9      int i, j;
10     for( i = 0; i < n; i++ )
11         for( r[ i ] = 0.0, j = 0; j < m; j++ )
12             r[ i ] += a[ i * m + j ] * v[ j ];
13 }
14
15 void prt_vec( double *v, int size )
16 {
17     int i;
18     for( i = 0; i < size; i++ )
19         printf( "%4.1f ", v[ i ] );
20     printf( "\n" );
21 }
22
23 void prt_mat( double *v, int n, int m )
24 {
25     int i;
26     for( i = 0; i < n; i++ )
27         prt_vec( v + i * m, m );
28 }
29
30 int main( int argc, char **argv )
31 {
32     double a[][M] = {{ 1.0, 0.0, 0.0 }, { 1.0,
33                                     1.0, 0.0 }, { 1.0, 1.0, 1.0 }};
34     double v[] = { 1.0, 2.0, 3.0 }, r[ N ];
35     mat_mul( & a[ 0 ][ 0 ], v, r, N, M );
36     printf( "Matrix A:\n" ); prt_mat(& a[0][0], N, M);
37     printf( "Vektor V: " );  prt_vec( v, M );
38     printf( "Vektor R: " );  prt_vec( r, N );
39 }
```

# Übungspaket 24

## Zeichenketten

---

### Übungsziele:

1. Verständnis über die interne Repräsentation von Zeichenketten
2. Arbeiten mit Zeichenketten
3. Definition konstanter Zeichenketten
4. Verwendung diverser Sonderzeichen in Zeichenketten

### Skript:

Kapitel: 50

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Schon bei unseren ersten Programmerversuchen hatten wir mit konstanten Zeichenketten zu tun. Damals waren es die einfachen Ausgabertexte sowie Eingabeformate. In diesem Übungspaket beschäftigen wir uns eingehender mit der internen Repräsentation, der Bearbeitung von Zeichenketten und dem Verwalten von Zeichenketten in Tabellen, um diese zu sortieren oder in ihnen zu suchen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Interne Repräsentation *einzelner* Zeichen

Zunächst wiederholen wir nochmals den Datentyp `char`, denn sicher ist sicher ;-) Auf welche drei Arten kann innerhalb eines C-Programms ein Zeichen dargestellt werden? Erkläre dies am Beispiel des Großbuchstabens A:

1. `'A'`
2. `65` (nur ASCII)
3. `'\101'`

Wie viel Speicherplatz benötigt ein Zeichen (Datentyp `char`) `Genau ein Byte bzw. 8 Bit`

Was bedeutet die Abkürzung ASCII?

ASCII bedeutet „American Standard Code for Information Interchange“. Dieser ASCII-Code definierte früher 128 unterschiedliche Zeichen, von denen einige zum Darstellen gedacht waren, andere hingegen für das Realisieren von Datenübertragungsprotokollen.

Was ergibt `sizeof(char)` im C-Programm? `1 (Byte) (das kleinste eigenständige Datum)`

## Aufgabe 2: Aufbau von Zeichenketten

Erkläre mit eigenen Worten, was aus Nutzersicht eine Zeichenkette ist.

Eine Zeichenkette ist eine Aneinanderreihung einzelner Zeichen, die man auch Text oder im Englischen String nennt. Zeichenketten (Texte) machen ein Programm spannender.

Unter *konstanten Zeichenketten* verstehen wir diejenigen, die wir als Programmierer selbst direkt hinschreiben. Wie werden diese im C-Programm dargestellt? Anfang: `"` Ende: `"`

Welchen Typ verwendet man in C für Zeichenketten? `char *`

Wie werden Zeichenketten im Arbeitsspeicher abgelegt? `Zeichenweise nacheinander.`

Wie erkennt der Compiler den Anfang? `Er weiß die Anfangsadresse jeder Zeichenkette.`

Wie erkennt er ihr Ende? `Er hängt ein Null-Byte (Bits: 0000 0000) an.`

In welchem Segment werden Zeichenketten abgelegt? `Im Konstantensegment`

Können konstante Zeichenketten verändert werden? `Nein, keinesfalls!`

Falls nein, wie kann man sich seine Zeichenketten selbst zusammenbauen?

Man kann sich ein Array vom Typ `char` nehmen, und darin die einzelnen Stellen mit den gewünschten Zeichen belegen. Wichtig ist, dass es so groß ist, dass an das Ende noch ein Null-Byte (`'\0'`) passt.

Wie können die folgenden vier Sonderzeichen in eine Zeichenkette eingefügt werden?

1. Tabulator `\t`   2. Zeilenwechsel `\n`   3. Gänsefüßchen `\"`   4. \ (Backslash) `\\`

Müssen Zeichenketten in einer Zeile anfangen und abgeschlossen sein? `Ja, unbedingt!`

Was kann man machen, wenn die Zeichenkette länger als eine Zeile ist? `'\'` ans Ende

**Beispiel:** In folgendem Quelltext hat das Label `NAME` den Wert `"Herr Anneliese"`.

```
1 #define NAME                "Herr \  
2 Anneliese"
```

**Erklärung:** Der Präprozessor entfernt bereits jedes Auftreten der Zeichenkombination `'<Backslash><Zeilenwechsel>'`, sodass der eigentliche C-Compiler sie nicht mehr sieht.

## Aufgabe 3: Eingabe

Funktionen zum Einlesen von Zeichenketten wie beispielsweise `scanf( "%s", buf )` verhindern keinen *Buffer Overflow* (sie sind nicht *fail save*), was zu fatalen Fehlern führen kann. Entsprechend sichere Alternativen besprechen wir am Ende dieses Übungspaketes.

## Aufgabe 4: Ausgabe

Mit welcher Formatangabe können Zeichenketten ausgegeben werden? `%s`

Gib ein Beispiel: `printf( "Hallo %s, wie geht es ihnen?\n", "Frau Peter" );`

## Aufgabe 5: Operationen mit Zeichenketten

Es gibt einige Operationen, die auf Zeichenketten arbeiten. Diese gehören aber nicht zum C-Sprachumfang, sondern sind Teil der C-Standardbibliothek (`#include <string.h>`). Erläutere kurz, was die folgenden Funktionen machen:

`strcmp()`: Vergleicht zwei Zeichenketten auf ihre lexikalische Ordnung. Das Ergebnis ist kleiner als null, null oder größer als null, je nach dem, welche der beiden Zeichenketten früher im Alphabet kommt.

`strcpy()`: Kopiert eine Zeichenkette in eine andere. Aufruf: `strcpy( ziel, quelle )`

`strlen()`: Bestimmt die Zahl der darzustellenden Zeichen einer Zeichenkette.  
Beispiel: `strlen( "vier" ) ⇒ 4`

`sizeof()`: Zahl der Bytes im *Arbeitsspeicher*. `sizeof( "vier" ) ⇒ 5` (4 plus `'\0'`)

## Teil II: Quiz

## Aufgabe 1: Zeichenketten: C-Kodierung und Ausgabe

In dieser Übungsaufgabe gehen wir davon aus, dass die angegebenen Zeichenketten mittels der Funktion `printf()` ausgegeben werden. In der folgenden Tabelle sind links verschiedene Zeichenketten angegeben, die alle gültiger C-Code sind, und rechts, wie sie auf dem Bildschirm erscheinen würden. In jeder Zeile fehlt eine der beiden Seiten, die ihr vervollständigen sollt. Leerzeichen und Zeilenumbrüche symbolisieren wir mit den Zeichen `_` und `↵`. Ferner gehen wir davon aus, dass sich alle acht (Ausgabe-) Positionen ein Tabulator-Stop befindet.

[illegible]



## Teil III: Fehlersuche

---

### Aufgabe 1: Ausgeben von Zeichenketten

Wie sollte es anders sein, wieder einmal hat einer unserer Starprogrammierer, diesmal DR. STRING, etwas für uns programmiert. Aber irgendwie will es nicht laufen. Finde die Fehler und korrigiere sie (direkt im Quelltext):

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     printf( "es geht ganz einfach los\n" );
6     printf( "\netwas \t''schwieriger''\n" );
7     printf( "noch "schwieriger"\n" );
8     printf( "ganz \'schwieriger\'\n" );
9     printf( "a\"usserst \""schwierig\""n" );
10    printf( \"ultra a\\\"usserst \"schwierig\"n" );
11    printf( "so, nun reicht \
12 es mir aber. \"ich\" gehe jetzt \
13 windsurfen\\ egal, \nwer das wissen will \":-)\n" );
14 }
```

Zeile	Fehler	Erläuterung	Korrektur
5	N statt n	Die Escape-Sequenz für den Zeilenwechsel schreibt sich mit kleinem N.	\n
6		Hier ist alles richtig.	
7	" statt \"	Ein Gänsefüßchen beendet eine Zeichenkette und kann somit nicht Bestandteil von ihr sein, es sei denn, man stellt ihm ein Backslash voran.	\"
8	\' statt ´	In C gibt es keine Escape-Sequenz \'. Um ein Apostroph auszugeben, schreibt man ihn einfach hin.	´
9	\"" statt \\\"	Die ersten beiden Backslashes bilden eine Einheit. Somit ist das Gänsefüßchen ohne ein Backslash und beendet die Zeichenkette vorzeitig.	\\\"
10	\\" statt "	Da wir am Anfang noch nicht innerhalb einer Zeichenkette sind, kann der Compiler mit der Escape-Sequenz \" noch nichts anfangen.	"
11-13		Hier ist wieder alles richtig.	

### Programm mit Korrekturen:

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     printf( "es geht ganz einfach los\n" );
6     printf( "\netwas \t''schwieriger''\n" );
7     printf( "noch \"schwieriger\"\n" );
8     printf( "ganz 'schwieriger'\n" );
9     printf( "a\"usserst \"schwierig\"\n" );
10    printf( "ultra a\\\"usserst \"schwierig\"\n" );
11    printf( "so, nun reicht \
12 es mir aber. \"ich\" gehe jetzt \
13 windsurfen\\ egal, \nwer das wissen will \":-)\n" );
14 }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Länge einer Zeichenkette ermitteln

#### 1. Aufgabenstellung

Wie wir schon im ersten Teil dieses Übungspaketes kurz angerissen haben, gibt es eine Funktion `strlen( char * str )`, die die Länge einer Zeichenkette `str` ermittelt. Ziel dieser Übungsaufgabe ist es nun, diese Funktion selbst zu entwickeln. Wir nennen sie einfach `int my_strlen( char * str )`. Überlege im Spezifikationsteil (nächster Punkt), was das Kriterium für die zu verwendende Schleife ist und was die Funktion zurückgeben soll, wenn ihr ein Null-Zeiger übergeben wird.

#### 2. Pflichtenheft

Aufgabe	: Funktion zur Ermittlung der Länge einer Zeichenkette
Parameter	: Eingabeparameter <code>str</code> vom Typ <code>char *</code> .
Rückgabewert	: Länge der Zeichenkette, <code>-1</code> falls <code>str</code> ein Nullzeiger ist.
Schleifenbedingung	: Die Schleife muss beendet werden, wenn das Null-Byte erreicht ist. Die Schleifenbedingung lautet <code>*str != Null-Byte</code> .

#### 3. Testdaten

Zeichenkette	"vier"	"C-Kurs"	"hoch\""	"\"\\\""	"Mr. _\130"	" "	0
Ergebnis	4	6	5	3	5	0	-1

#### 4. Implementierung

Wenn wir einen Null-Zeiger haben, soll das Ergebnis `-1` sein. In allen anderen Fällen müssen wir innerhalb der Zeichenkette solange nach rechts gehen, bis wir ihr Ende – das Null-Byte – erreicht haben und dabei jedesmal den Längenzähler um eins erhöhen. Abstrakt formuliert erhalten wir folgende algorithmische Beschreibung:

Länge einer Zeichenkette ermitteln

Parameter: Zeiger auf Zeichen: `str`

Variablen: Integer: `len`

wenn `str != 0`

dann Setze `len = 0`

solange `*str != Null-Byte`

wiederhole Setze `str = str + 1` // eins weiter schalten

Setze `len = len + 1`

sonst Setze `len = -1`

return `len`

## 5. Kodierung

```
1  #include <stdio.h>
2
3  #define FMT      "str= '%s' len= %d\n"
4
5  int my_strlen( char * str )
6  {
7      int len;
8      if ( str != 0 ) // if (str) would be fine as well
9          for( len = 0; *str; str++ )
10             len++;
11      else len = -1;
12      return len;
13  }
14
15  int main( int argc, char **argv )
16  {
17      printf( FMT, "vier", my_strlen( "vier" ) );
18      printf( FMT, "C-Kurs", my_strlen( "C-Kurs" ) );
19      printf( FMT, "", my_strlen( "" ) );
20      printf( FMT, "null-zeiger", my_strlen( 0 ) );
21  }
```

**Anmerkung:** Obiges Beispiel zeigt zusätzlich, dass man die Formatierungsanweisung für sich wiederholende Ausgabeanweisungen auch mittels eines `#define` „zentral“ definieren kann.

## Aufgabe 2: Vokale in einer Zeichenkette zählen

### 1. Aufgabenstellung

Ziel dieser Übungsaufgabe ist es, eine Funktion zu entwickeln, die die Zahl der Vokale einer Zeichenkette zählt. Wir nennen diese Funktion `int vokale( char * str )`. Überlege im Spezifikationsteil, was das Kriterium für die zu verwendende Schleife ist und was die Funktion zurückgeben soll, wenn ihr ein Null-Zeiger übergeben wird.

### 2. Pflichtenheft

Aufgabe	: Funktion zum Zählen der Vokale einer Zeichenkette
Parameter	: Eingabeparameter <code>str</code> vom Typ <code>char *</code> .
Rückgabewert	: Zahl der Vokale, <code>-1</code> falls <code>str</code> ein Nullzeiger ist.
Schleifenbedingung	: Die Schleife muss beendet werden, wenn das Null-Byte erreicht ist. Die Schleifenbedingung lautet <code>*str != Null-Byte</code> .

### 3. Testdaten

Zeichenkette	"hallo"	"aAeEiIoOuU"	"\141\145\151\157\165\n"	0
Ergebnis	3	10	5	-1

#### 4. Implementierung

Strukturell gesehen, müssen wir genau das Gleiche machen wie in der vorherigen Aufgabe. Nur müssen wir den Längenzähler jetzt nicht mehr jedes mal sondern nur bei Vokalen um eins hochzählen. Das ergibt folgende Implementierung:

Zählen von Vokalen in einer Zeichenkette

Parameter: Zeiger auf Zeichen: str

Variablen: Integer: cnt

```
wenn str != 0
dann Setze cnt = 0
    solange *str != Null-Byte
    wiederhole wenn ist_vokal( *str )
        dann Setze cnt = cnt + 1
        Setze str = str + 1 // eins weiter schalten
sonst Setze cnt = -1
return cnt
```

#### 5. Kodierung

Die Funktion vokale() kann in C wie folgt kodiert werden:

```
1 int vokale( char * str )
2 {
3     int cnt;
4     if ( str != 0 ) // if (str) would be fine as well
5     {
6         for( cnt = 0; *str; str++ )
7             if ( *str == 'a' || *str == 'A' ||
8                 *str == 'e' || *str == 'E' ||
9                 *str == 'i' || *str == 'I' ||
10                *str == 'o' || *str == 'O' ||
11                *str == 'u' || *str == 'U' )
12                cnt++;
13     }
14     else cnt = -1;
15     return cnt;
16 }
```

## Aufgabe 3: Zählen von Ziffern

### 1. Aufgabenstellung

In dieser Aufgabe soll nicht die Zahl der Vokale sondern die Zahl der Ziffern in einer Zeichenkette bestimmt werden. Da sich diese Aufgabe in nur einem kleinen Detail von der vorherigen unterscheidet, übernehmen wir fast alles und fahren gleich mit der Kodierung fort.

**Hinweis:** Statt jedes Zeichen einzeln auf die Ziffern zu überprüfen können wir das Makro `isdigit()` aus der Standardbibliothek `<ctype.h>` verwenden.

### 2. Kodierung

```
1  #include <string.h>
2
3  int digits( char * str )
4  {
5      int i, cnt = -1;
6      if ( str != 0 )           // or simply: if ( str )
7          for( i = cnt = 0; str[ i ]; i++ )
8              if ( isdigit( str[ i ] ) )
9                  cnt++;
10     return cnt;
11 }
```

Etwas Aufmerksamkeit erfordert vielleicht noch Zeile 8: Statt hier eine `if`-Abfrage zu nehmen, addieren wir gleich das Ergebnis des Makros `isdigit()`. Dieses liefert nämlich eine 1 (logisch wahr), wenn es sich um eine Ziffer handelt, sonst den Wert 0. Dieser „Trick“ liefert folgende Kurzfassung:

```
1  #include <string.h>
2
3  int digits( char * str )
4  {
5      int cnt = -1;
6      if ( str )
7          for( cnt = 0; *str; str++ )
8              cnt += isdigit( *str );
9      return cnt;
10 }
```

Für welche der Varianten ihr euch entscheidet, ist erst einmal Geschmackssache. Nehmt diejenige, die für euch klarer ist; die meisten Programmieranfänger werden sich für die erste Variante entscheiden, aber dazulernen schadet auch nichts ;-)

# Aufgabe 4: Vergleich zweier Zeichenketten

## 1. Aufgabenstellung

Auch wenn es bereits eine Funktion `strcmp()` in der Standardbibliothek gibt, so ist es dennoch eine gute Übung, diese einmal selbst zu programmieren. Mit dem bisher Erlernten sollte dies auch kein Problem sein. Entsprechend ist die Aufgabe, eine Funktion zu entwickeln, die zwei Zeichenketten miteinander vergleicht und ein Resultat zurückgibt. Anhand des Resultats soll man erkennen, ob die erste Zeichenkette kleiner (lexikalisch vor), gleich (lexikalisch identisch) oder größer (lexikalisch nach) als die zweite Zeichenkette ist. Die Funktion kann davon ausgehen, dass keines der beiden Argumente (Operanden) ein Null-Zeiger ist.

**Beispiele:** `my_strcmp( "abc", "abe")`  $\Rightarrow$  -2  
`my_strcmp( "xx", "xx")`  $\Rightarrow$  0

## 2. Pflichtenheft

Aufgabe : Funktion zum lexikalischen Vergleichen zweier Zeichenketten  
Parameter : Eingabeparameter `s1` und `s2` vom Typ `char *`.  
Rückgabewert: <0 falls `s1` lexikalisch vor `s2` kommt, 0 falls `s1` und `s2` identisch sind und >0 falls `s1` lexikalisch nach `s2` kommt.

## 3. Testdaten

Als Testdaten nehmen wir einfach die in der Aufgabenstellung erwähnten Beispiele.

## 4. Vorüberlegungen

Hier geben wir ein paar kleine Hinweise, wie man diese Aufgabe am besten löst. Sportlich orientierte lesen lieber nicht, sondern versuchen es erst einmal alleine.

Was müssen wir machen? Nehmen wir an, wir haben zwei Zeichenketten "`a11`" und "`a22`". Wir müssen jetzt in beiden Zeichenketten solange „synchron“ nach rechts gehen, bis wir einen Unterschied gefunden haben. In unserem Beispiel ist dies an der zweiten Stelle der Fall, 1 ist anders als 2. Wenn wir diese Stelle gefunden haben, reicht es aus, die beiden Werte voneinander zu subtrahieren.

Jetzt könnte es aber eine Komplikation geben: Sollten beide Zeichenketten identisch sein, gibt es keinen Unterschied und eine derartige Schleife würde über das Ende der Zeichenketten hinaus gehen. Von da her müssen wir mit der Schleife auch dann aufhören, wenn wir das Ende einer der beiden Zeichenketten erreicht haben. Auch jetzt funktioniert die Idee mit der Subtraktion: sollten beide Zeichenketten identisch sein, subtrahiert der Algorithmus die beiden Null-Bytes voneinander, was zum Resultat 0 führt und somit die Identität beider anzeigt.

## 5. Implementierung

Die Ideen der Vorüberlegung lassen sich wie folgt direkt in eine algorithmische Beschreibung umwandeln.

Funktion zum Vergleich zweier Zeichenketten

Parameter: Zeiger auf Zeichen: s1, s2  
Variablen: keine  
solange \*s1 = \*s2 und \*s1 != Null-Byte  
wiederhole Setze s1 = s1 + 1  
                  Setze s2 = s2 + 1  
return \*s1 - \*s2

## 6. Kodierung

```
1  #include <stdio.h>
2
3  int my_strcmp( char * s1, char * s2 )
4  {
5      while( *s1 && *s1 == *s2 )
6      {
7          s1++; s2++;
8      }
9      return *s1 - *s2;
10 }
11
12 int main( int argc, char **argv )
13 {
14     printf( "%d\n", my_strcmp( "abc", "abe" ) );
15     printf( "%d\n", my_strcmp( "xx", "xx" ) );
16 }
```

## 7. Alternative Kodierung

Einige von euch sind im Umgang mit Zeigern noch nicht ganz sicher. Daher haben wir hier die Funktion my\_strcmp() nochmals in „Array-Form“ abgedruckt.

```
1  int my_strcmp( char s1[], char s2[] )
2  {
3      int i;
4      for( i = 0; s1[ i ] && s1[ i ] == s2[ i ]; )
5          i++;
6      return s1[ i ] - s2[ i ];
7  }
```



# Aufgabe 5: Suchen von Zeichenketten

## 1. Aufgabenstellung

Mit der Funktion aus der letzten Aufgabe (oder der entsprechenden Funktion aus der Standardbibliothek) können wir nun auch Zeichenketten in Tabellen suchen und hoffentlich auch finden. Derartige Funktionen benötigt man beispielsweise, wenn man einen Namen in einer Tabelle suchen muss.

Aufgabe: Schreibe eine Funktion `search_str()`, die eine gegebene Zeichenkette (token) in einer Tabelle (Array) von Zeichenketten sucht und ggf. findet.

## 2. Entwurf

Welche Indizes sind in einer Tabelle mit `size` Einträgen gültig? `0...size-1`

Welchen Index nehmen wir üblicherweise bei „nicht gefunden“? `-1`

Vervollständige folgende Funktionsdeklaration:

`search_str()` `int search_str(char *token, char **table, int size);`

## 3. Pflichtenheft

Aufgabe	: Suchen einer Zeichenkette in einer Tabelle
Parameter	: Eingabeparameter <code>token</code> vom Typ Zeiger auf Zeichen, <code>table</code> vom Typ Array of Zeiger auf Zeichen und <code>size</code> vom Typ <code>int</code>
Rückgabewert	: Index von „ <code>token</code> “ in der Tabelle <code>table</code> oder <code>-1</code>

## 4. Implementierung

Programmiertechnisch gesehen ist es egal, ob wir Zahlen oder Zeichenketten suchen. Wir müssen nur die Datentypen und den Vergleichsoperator anpassen. Daher können wir das meiste aus Aufgabe 2, Übungspaket 15 übernehmen. Ferner wissen wir, dass sich zwei Zeichenketten mittels der Funktion `strcmp()` vergleichen lassen.

Funktion Suchen einer Zeichenkette in einer Tabelle

Parameter: Zeiger auf Zeichen: `token`  
          Zeiger auf Zeiger auf Zeichen: `table`  
          Ganzzahl: `size`

Variable: Ganzzahl: `i`

für `i = 0` bis `size-1` Schrittweite `1`  
wiederhole wenn `token` gleich `table[ i ]`  
          dann return `i`  
  
return `-1`

## 5. Kodierung

Die Funktion `search_str()`:

```
1  #include <string.h>                                // for strcmp()
2
3  int search_str( char *token, char **table, int size )
4  {
5      int i;
6      for( i = 0; i < size; i++ )
7          if ( ! strcmp( token, table[ i ] ) )
8              return i;
9      return -1;
10 }
```

Das Hauptprogramm zum Testen:

```
1  #include <stdio.h>                                // for printf()
2
3  int main( int argc, char **argv )
4  {
5      // the table of strings
6      char *days[] = { "Mo", "Di", "Mi",
7                          "Do", "Fr", "Sa", "So" };
8      #define DAYS      (sizeof(days)/sizeof(days[0]))
9
10     // here we have another table with test strings
11     char *test[] = { "Mo", "So", "mi", "heute" };
12     #define TEST       (sizeof(test)/sizeof(test[0]))
13
14     // here comes the test
15     int i, index;
16     for( i = 0; i < TEST; i++ )
17     {
18         index = search_str( test[ i ], days, DAYS );
19         printf( "%s: %d\n", test[ i ], index );
20     }
21 }
```

## Aufgabe 6: Sicheres Einlesen von Zeichenketten

Am Anfang dieses Übungspakets haben wir darauf hingewiesen, dass die Verwendung von Standardfunktionen zum Einlesen von Zeichenketten zu fatalen Fehlern führen kann. In dieser Aufgabe schauen wir uns zunächst das eigentliche Problem an, um dann selbst eine sichere Alternative zum Einlesen von Zeichenketten zu entwickeln.

### 1. Vorbetrachtungen und Motivation

Die wohl am meisten verbreitete Methode zum Einlesen von Zeichenketten besteht in folgendem Funktionsaufruf: `scanf( "%s", buf )`, wobei `buf` ein Array vom Typ `char` ist. Die Funktionalität ist, wie man es sich vorstellt: alle Zeichen bis zum nächsten Zeilenumbruch `\n` werden eingelesen, im Array `buf` abgelegt und mit einem Null-Byte `\0` abgeschlossen. Häufig wird auch der Funktionsaufruf `gets( buf )` verwendet, der zum selben Resultat führt.

Von beiden Vorgehensweisen raten wir dringend ab! Don't do that! Never ever! Don't even think about it! „Hoppla, warum seid ihr denn so hysterisch? Beide Funktionsaufrufe sehen doch prima aus?“ Ganz einfach: Beide Funktionsaufrufe sind nicht *fail safe* und können im besten Falle zum Programmabsturz, im ungünstigsten Fall zu ganz merkwürdigen Effekten führen. „Ja, ja, erzählt mal! Das ist doch Standard-C. Wo soll denn da ein Fehler auftreten können?“ Das Problem liegt darin, dass die Größe des Puffers `buf` nicht überprüft wird. Nehmen wir folgende Funktion:

```
1 void f( int i )
2     {
3         char buf[ 4 ];
4         scanf( "%s", buf );
5         printf( "i=%d buf='%s'\n", i, buf );
6     }
```

In Zeile 4 wird eine Zeichenkette eingelesen und in Zeile 5 wird diese nebst des Parameters `i` wieder ausgegeben. Um die folgenden Ausführungen zu verstehen, benötigen wir erst einmal einen Stack-Frame, den ihr für den Funktionsaufruf `f( 1 )` und die Eingabe `hi\n` vervollständigen sollt:

Adresse	Variable	Wert
0xFFEE1008	int i:	1
0xFFEE1004	CPU PC:	„Zeile x“
0xFFEE1000	char buf[ 4 ]:	'h' 'i' '\0'

Nach dem Einlesen befinden sich drei Zeichen im Array `buf`, wobei der Zeilenwechsel `\n` durch ein Null-Byte `\0` ersetzt wurde. So weit, so gut. „Ja, was soll der Unsinn? Ist doch klar, wo ist das Problem?“ Das Problem bekommen wir, wenn die vom Nutzer eingegebenen Zeichenketten länger werden. Vervollständige einen Stack-Frame für den Funktionsaufruf `f( 1 )` und die eingegebene Zeichenkette `01234567890\n`:

Adresse	Variable	Wert
0xFFEE1008	int i:	'8' '9' '0' '\0'
0xFFEE1004	CPU PC:	'4' '5' '6' '7'
0xFFEE1000	char buf[ 4 ]:	'0' '1' '2' '3'

Bei richtiger Komplettierung sehen wir, dass nicht nur die Variable **buf** einen Wert erhalten hat sondern dass sowohl der alte PC (die Rücksprungadresse) als auch die Variable **i** mit neuen Werten überschrieben wurden. Dies hat weitreichende Konsequenzen: Nach Beendigung der Funktion **f()** springt die CPU an die falsche Stelle zurück. Ferner können beliebig viele Variablen mit neuen Werten überschrieben worden sein. Im obigen Beispiel könnte auf einigen Systemen die Variable **i** den Wert 3160376 erhalten; auf anderen Systemen ganz andere Werte. Aus diesem Grund sollten wir von eingangs erwähnten Funktionsaufrufen strikt Abstand nehmen und unsere eigene Einlesefunktion realisieren.

## 2. Aufgabenstellung

Schreibe eine Funktion **myReadString()** die eine Zeichenkette von der Tastatur einliest und in einem übergebenen Puffer der Länge **size** ablegt. Das Einlesen wird beendet, wenn entweder der Zeilenumbruch **\n** gelesen wurde oder der Puffer voll ist (wobei immer noch Platz für das Null-Byte vorhanden sein muss).

## 3. Entwurf

Vervollständige zunächst den Funktionskopf der zu realisierenden Funktion:

**myReadString():** `int myReadString( char *buf, int size )`

## 4. Implementierung

```
Funktion myReadString
Parameter: Pointer to Zeichen: buf; Integer: size
Variable: Zeichen: c
           Integer: i
setze i = 0; lese c
solange c ≠ \n and i < size - 1
wiederhole setze buf[ i ] = c
           setze i = i + 1
setze buf[ i ] = \0
return i
```

## 5. Kodierung

Unsere Kodierung sieht wie folgt aus:

**Die Lesefunktion:**

```
1  #include <stdio.h>
2
3  int myReadString( char *buf, int size )
4  {
5      char c;
6      int i = 0;
7      c = getc( stdin );
8      while( c != '\n' && i < size - 1 )
9      {
10         buf[ i++ ] = c;
11         c = getc( stdin );
12     }
13     buf[ i ] = '\0';
14     return i;
15 }
```

**Das Hauptprogramm:**

```
17 int main()
18 {
19     char buf[ 10 ];
20     int i;
21     printf( "bitte eine Zeichenkette eingeben: " );
22     i = myReadString( buf, sizeof( buf ) );
23     printf( "%d Zeichen gelesen: '%s'\n", i, buf );
24 }
```

# Übungspaket 25

## Kommandozeilenargumente

---

### Übungsziele:

1. Umgang mit `argc/argv`
2. `argc/argv` als Schnittstelle von Programm und Betriebssystem
3. Vereinfachtes Testen mit `argc/argv`

### Skript:

Kapitel: 51 und Übungspaket 24

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Eine immer wieder gute Frage ist, woher der `gcc`-Compiler weiß, welche Datei er eigentlich übersetzen und wo er das Ergebnis hinschreiben soll. Ist doch klar, werden einige sagen, steht doch in der Kommandozeile `gcc looser.c`. Ja, aber woher soll nun das Kommando `gcc` wissen, was da so alles an Optionen und Argumenten steht? Genau das bewerkstelligt das Betriebssystem in Zusammenarbeit mit dem `argc/argv`-Mechanismus.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Kommandos, Argumente und Optionen

Erkläre mit eigenen Worten, was unter den Begriffen Kommando, Argumente und Optionen zu verstehen ist.

Das Kommando ist immer das erste, was in einer Eingabezeile steht. Daran anschließen können sich weitere Argumente und Optionen. Unter Optionen versteht man in der Regel Angaben, die mit einem Minuszeichen beginnen oder in der Mitte ein Gleichheitszeichen haben. Alles weitere bezeichnet man *üblicherweise* als Argumente (oder Parameter). Zur Illustration diene folgendes Beispiel:

```
greetings -now status='very good' good-friends.txt very-good-friends.txt
```

Kommando: `greetings`

Optionen : `-now, status='very good'`

Argumente : `good-friends.txt, very-good-friends.txt`

**Anmerkung:** Die Apostrophs im Ausdruck `status='very good'` dienen dazu, die beiden Wörter `very` und `good` zu einer lexikalischen Einheit zusammenzufügen; andernfalls wären es zwei getrennte Argumente.

Erkläre mit eigenen Worten, wie die Kommandozeile zerlegt und mittels `argc/argv` an das Programm übermittelt wird.

Für das Zerlegen der Kommandozeile in seine einzelnen Bestandteile ist die Shell (das Programm der Konsoleneingabe) zuständig. Aus den einzelnen Bestandteilen (auch Tokens genannt) baut es ein Array aus *Zeichenketten* auf. Die einzelnen Elemente werden der Reihe nach belegt. Das Programm bekommt dann einen Zeiger auf dieses Array und die Zahl der Einträge übermittelt (hierfür ist das Betriebssystem zuständig). Diese beiden Parameter werden *üblicherweise* `argc/argv` genannt. Da diese beiden Parameter aber formale Parameter der Funktion `main()` sind, könnten sie jeden beliebigen Namen bekommen. Da der Aufruf aber vom Betriebssystem organisiert wird, müssen Typ und Reihenfolge stimmen, sonst geht's in die Hose. Obiges Beispiel würde zu folgender Belegung führen:

```
argc      :    5
argv[ 0 ] : greetings
argv[ 1 ] : -now
argv[ 2 ] : status=very_good
argv[ 3 ] : good-friends.txt
argv[ 4 ] : very-good-friends.txt
```

Kleiner Hinweis: durch den ersten Parameter (Index 0) erfährt das Programm auch, unter welchem Namen es aufgerufen wurde.

## Aufgabe 2: Detailfragen zu argc/argv

Von welchem Datentyp ist der Parameter <code>argc</code> ?	<code>int</code>
Von welchem Datentyp ist der Parameter <code>argv[ 0 ]</code> ?	<code>char *</code>
Von welchem Datentyp ist der Parameter <code>argv</code> ?	<code>char **</code> bzw. <code>char *argv[]</code>
Wie viele Einträge hat das Array <code>argv</code> ?	<code>argc</code> viele
Welche Indizes sind zulässig?	<code>0 .. argc-1</code>
Müssen diese beiden Parameter <code>argc/argv</code> heißen?	Nein!
Gibt es ein Element <code>argv[ argc ]</code> ?	Nach aktuellem Standard ja
Falls ja, was steht dort?	Ein Nullzeiger
In welchem Segment liegt <code>argc</code> ?	Stack ( <code>argv[]</code> ist ein formaler Parameter)
In welchem Segment liegt <code>argv</code> ?	Stack ( <code>argv[]</code> ist ein formaler Parameter)
In welchem Segment liegen die <code>argv[ i ]</code> ?	Stack (durch das Betriebssystem)
Kann man den Typ von <code>argv[ i ]</code> ändern?	Nein! Das Betriebssystem liefert <code>argv[]</code>
Nutzt ein <code>int argv[]</code> etwas?	Nein! Das Betriebssystem liefert <code>argv[]</code>
Gibt es irgendeine andere Möglichkeit?	Nein! Das Betriebssystem liefert <code>argv[]</code>
Merkt der Compiler eine Typänderung?	Nein! <code>main()</code> ist eine gewöhnliche Funktion
Muss ich <code>argc/argv</code> akzeptieren wie es ist?	Ja! Und zwar <i>ohne</i> wenn und aber!

## Aufgabe 3: Verwendung des argc/argv Mechanismus

Viele Programmieranfänger und damit auch unsere Studenten haben immer wieder den Eindruck, dass der `argc/argv` Mechanismus völlig veraltet ist und etwas mit alten Lochkarten oder ähnlichem zu tun hat. Dabei ist dieser Mechanismus allgegenwärtig und wird bei *jedem* Programmaufruf verwendet. Schaut euch dazu einmal eure Desktop-Verknüpfungen an. Nennt mindestens zwei Beispiele, wie dabei `argc/argv` zur Anwendung kommt:

Zwei beliebig herausgesuchte Beispiele sehen wie folgt aus:

1. Beim Doppelclick auf ein Word-Dokument, das auf dem Desktop liegt, wird der Name der Datei an Word übergeben. Heißt das Dokument beispielsweise `hot-guys.doc` so wird mittels der Desktop-Verknüpfung folgender Aufruf erzeugt:

```
word hot-guys.doc
```

2. Wenn auf dem Desktop ein `html`-Dokument mit dem Namen `wind.html` liegt, so wird beispielsweise unter Linux folgender Aufruf generiert:

```
iceweasel file:///home/rs/Desktop/wind.html
```

Nur so wissen die Programme (`word/iceweasel`), was sie mit wem machen sollen.



## Aufgabe 4: Kommandoargumente im Arbeitsspeicher

Nehmen wir an, der Nutzer tippt folgendes Kommando ein:

```
echo Montag      Dienstag und Mittwoch
```

Zur Erläuterung: Das Kommando `echo` gibt es wirklich. Es macht nichts anderes, als das auf der Konsole auszugeben, was ihm als Argumente mitgeliefert wird. In unserem Beispiel würde es also die Wochentage `Montag` `Dienstag` und `Mittwoch` ausgeben.

Zeichne nun im Folgenden den Stack-Frame nebst der Inhalte der einzelnen `argv[ i ]`. Dabei interessieren wir uns nur für die Variablen `argc`, `argv` und deren Inhalte, Zeichenketten etc. Andere Informationen wie die Rücksprungadresse und den Rückgabewert können in dieser Übung getrost weggelassen werden. Das erfolgreiche Bearbeiten dieser Übungsaufgabe hilft sehr beim Bearbeiten von Aufgabe 2 des nächsten Teils. Für den Stack-Frame haben wir die nächste Seite vorgesehen, da er etwas größer ist.

Adresse	Variable	Wert	Kommentar
0xFFEE1080	.....	.....	
0xFFEE3020		Mittwoch\0	Daten für argv[4]
0xFFEE301C		und\0	Daten für argv[3]
0xFFEE3010		Dienstag\0	Daten für argv[2]
0xFFEE3008		Montag\0	Daten für argv[1]
0xFFEE3000		echo\0	Daten für argv[0]
.....	.....	.....	
0xFFEE2010		0xFFEE3020	argv[ 4 ]
0xFFEE200C		0xFFEE301C	argv[ 3 ]
0xFFEE2008		0xFFEE3010	argv[ 2 ]
0xFFEE2004		0xFFEE3008	argv[ 1 ]
0xFFEE2000		0xFFEE3000	argv[ 0 ]
.....	.....	.....	
0xFFEE1064	char **argv:	0xFFEE2000	Variable argv von main()
0xFFEE1064	int argc:	5	Variable argc von main()

Anmerkung: Alle Adressen sind natürlich frei erfunden und entbehren jeglicher wahren Grundlage. *Aber* unter der Annahme, dass ein `int` und eine Adresse jeweils 4 Bytes belegen, stimmen in den einzelnen Blöcken die Relationen untereinander.

## Teil II: Quiz

---

### Aufgabe 1: Kommandozeile vs. Argumente

In dieser Übung schauen wir uns die Zuordnungen zwischen den Elementen der Kommandozeile und den Parametern `argc/argv` etwas genauer an. Zur Erinnerung: ein Token ist etwas, das durch Leerzeichen, Tabulatoren oder einem Zeilenwechsel abgegrenzt wird. Durch Verwenden von `'` oder `"` verlieren die „eingeklammerten“ Trenner ihre Bedeutung.

Kommandozeile	Argumente						
	0	1	2	3	4	5	6
<code>echo_1_2_3_4_5_6</code>	<code>echo</code>	<code>1</code>	<code>2</code>	<code>3</code>	<code>4</code>	<code>5</code>	<code>6</code>
<code>echo_12__34____56</code>	<code>echo</code>	<code>12</code>	<code>34</code>	<code>56</code>			
<code>echo_1-2_3-4_5-6_7-8</code>	<code>echo</code>	<code>1-2</code>	<code>3-4</code>	<code>5-6</code>	<code>7-8</code>		
<code>echo_'0_1'____2_3_4</code>	<code>echo</code>	<code>0_1</code>	<code>2</code>	<code>3</code>	<code>4</code>		
<code>echo_"0____1"____2_3_4</code>	<code>echo</code>	<code>0____1</code>	<code>2</code>	<code>3</code>	<code>4</code>		
<code>echo_'1_"_2'__"3_'_4"</code>	<code>echo</code>	<code>1_"_2</code>	<code>3_'_4</code>				

### Aufgabe 2: Argumente im Detail

In der folgenden Tabelle steht jeweils ein Ausdruck. Ergänze für jeden Ausdruck den resultierenden Datentyp und den jeweiligen Inhalt. Sollte es sich beim Inhalt um einen Zeiger auf eine Zeichenkette handeln, sollte dort diese Zeichenkette notiert werden.

**Kommando:** `echo wieder einmal 'so ein' Unfug heute !`

Ausdruck	Datentyp	Inhalt
<code>argc</code>	<code>int</code>	<code>7</code>
<code>argv[ 0 ]</code>	<code>char *</code>	<code>echo</code>
<code>argv</code>	<code>char **</code>	Zeiger auf das Array mit den sechs Zeigern; dort steht der Zeiger für <code>argv[ 0 ]</code> .
<code>*(argv + 2)</code>	<code>char *</code>	<code>einmal</code>
<code>**argv + 2)</code>	<code>char</code>	<code>e</code>
<code>argv[3][3]</code>	<code>char</code>	<code>e</code>
<code>argv[3] + 3</code>	<code>char *</code>	<code>_ein</code>
<code>strlen(argv[4] + 2)</code>	<code>int</code>	<code>3</code>
<code>&amp; argv[5][2]</code>	<code>char *</code>	<code>ute</code>

# Teil III: Fehlersuche

---

## Aufgabe 1: Fehler in der Argumentverarbeitung

Diesmal hat sich unser Experte DR. ARGUMENT versucht. Doch auch er hätte vielleicht öfter die Vorlesung besuchen sollen ;-) Ziel seines Programms ist es, alle Argumente in umgekehrter Reihenfolge auszudrucken. Dabei sollen aber nur diejenigen Argumente berücksichtigt werden, die mindestens vier Zeichen lang sind oder mit einer Ziffer anfangen. Beispiel: `reverse bla 12 blubber`  $\Rightarrow$  `blubber 12 reverse` Finde und korrigiere die Fehler.

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4
5 int main( char **argv, int argc )
6 {
7     int i;
8     for( i = argc; i >= 0; i-- )
9         if ( strlen( *argv ) > 3
10             || isdigit( argv[ i ][ 1 ] ) )
11             printf( "%s ", *argv[ i ] );
12     printf( "\n" );
13     return 0;
14 }
```

Zeile	Fehler	Erläuterung	Korrektur
5	<code>argc ↔ argv</code>	Die Reihenfolge der Argumente ist vertauscht. Aufgrund des Aufrufes durch die <code>init()</code> -Funktion ist die Reihenfolge <code>argc</code> , <code>argv</code> zwingend vorgegeben.	<code>int argc,</code> <code>char **argv</code>
8	<code>i = argc</code>	Da sich nach dem aktuellen Standard an der Stelle <code>argv[ argc ]</code> ein Null-Zeiger befindet, führt dies spätestens in Zeile 9 zu einem Programmabsturz.	<code>i = argc-1</code>
9	<code>*argv</code>	Gemäß Aufgabenstellung muss jedes Argument <code>argv[ i ]</code> auf seine Länge ( <code>strlen()</code> ) getestet werden, nicht nur das erste <code>*argv ↔ argv[ 0 ]</code> .	<code>argv[ i ]</code>
10	<code>[ 1 ]</code>	Da alle Argumente ausgegeben werden sollen, die mit einer Ziffer anfangen, muss jeweils das <i>erste</i> Zeichen jedes Arguments getestet werden.	<code>argv[i][0]</code> oder <code>*(argv[i])</code>

Zeile	Fehler	Erläuterung	Korrektur
11	*	<code>argv[ i ]</code> sind bereits die einzelnen Argumente. Die Kombination <code>*argv[ i ]</code> greift auf dessen erstes Zeichen zu. Also einmal zu viel dereferenziert.	<code>argv[ i ]</code>

#### Programm mit Korrekturen:

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4
5  int main( int argc, char **argv )
6  {
7      int i;
8      for( i = argc - 1; i >= 0; i-- )
9          if ( strlen( argv[ i ] ) > 3
10             || isdigit( argv[ i ][ 0 ] ) )
11             printf( "%s ", argv[ i ] );
12     printf( "\n" );
13     return 0;
14 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Das echo-Kommando

### 1. Aufgabenstellung

Wie schon mehrfach erwähnt, dient das `echo`-Kommando in Linux dazu, anfallende Resultate und Testanweisungen mittels des `argc/argv`-Mechanismus auszugeben. Ziel dieser Aufgabe ist es, diese einfache Funktionalität nachzubauen, was keine Schwierigkeiten mehr bereiten sollte. Die Standardfassung des `echo`-Kommandos hat aber für Übende eine kleine Schwäche: Es ist manchmal schwer ersichtlich, welches Argument zu welchem Parameter `argv[ i ]` gehört. Beispielsweise führen die beiden Kommandos `echo 1 2 3` und `echo '1 2' 3` zu identischen Ausgaben, obwohl sich die Parameterkonstellation deutlich unterscheidet. Vervollständige folgende Tabelle zur Selbstkontrolle:

<code>echo 1 2 3:</code>	<code>argc:</code>	<input type="text" value="4"/>	<code>argv[ 1 ]:</code>	<input type="text" value="1"/>	<code>argv[ 2 ]:</code>	<input type="text" value="2"/>
<code>echo '1 2' 3:</code>	<code>argc:</code>	<input type="text" value="3"/>	<code>argv[ 1 ]:</code>	<input type="text" value="1 2"/>	<code>argv[ 2 ]:</code>	<input type="text" value="3"/>

Aus diesem Grund wollen wir zwei Änderungen: Erstens soll jeder Parameter gesondert auf einer neuen Zeile erscheinen, und zweitens soll jeder Parameter mit einem geeigneten „Marker“ versehen werden.

Beispiel:	Eingabe	Ausgabe
	<code>my-echo 'bald ist' wochenende</code>	<code>my-echo:</code>
		<code>--&gt;bald ist&lt;--</code>
		<code>--&gt;wochenende&lt;--</code>

### 2. Implementierung

Die Aufgabe ist sehr einfach: Ausgabe des ersten Argumentes (dem Programmnamen) und dann für jedes weitere Argument eine neue Zeile mit dem Argument und zwei geeigneten Markern.

`my-echo`: Ausgabe der mitgegebenen Parameter

Parameter: Integer: `argc`

Array of Zeichenketten: `argv[ 0 .. argc - 1 ]`

Variablen: Integer: `i`

Ausgabe: `argv[ 0 ]`

für `i = 1` bis `argc - 1` Schrittweite 1

wiederhole Ausgabe Text: `-->` Wert `argv[ i ]` Text `<--`

### 3. Kodierung

```
1 #include <stdio.h>
2
3 int main( int argc, char **argv )
4 {
5     int i;
6     printf( "%s:\n", argv[ 0 ] );
7     for( i = 1; i < argc; i++ )
8         printf( "  -->%s<--\n", argv[ i ] );
9 }
```

### 4. Explorative Tests

Das erstellte Programm sollte mit den Testdaten aus Übungsaufgabe 1 aus dem Quizteil getestet werden. Um noch mehr über die Shell zu erfahren, sollte jeder sein Programm noch mit folgenden Eingaben testen:

Kommando	Argumente			
	argv[1]	argv[2]	argv[3]	argv[4]
my-echo "0 1" 2 3 4	-->0 1<--	-->2<--	-->3<--	-->4<--
my-echo 0" 1 2 3 4	-->0 1<--	-->2<--	-->3<--	-->4<--
my-echo "0 '1" 4	-->0 '1<--	-->4<--	--><--	--><--
my-echo "0 '1 2 ' 3" 4	-->0 '1 2 ' 3<--	-->4<--	--><--	--><--
my-echo 1\ 2 3 4	-->1 2<--	-->3<--	-->4<--	--><--
my-echo "1\ 2\" 3 " 4	-->1 2" 3 <--	-->4<--	--><--	--><--

## Aufgabe 2: argc/argv zu interaktiven Testzwecken

### 1. Aufgabenstellung

Der `argc/argv`-Mechanismus dient nicht nur dazu, den Studenten ein paar Extraaufgaben zu geben. Nein, er eignet sich insbesondere für das effiziente Testen der eigenen Programme. Gehen wir beispielsweise nochmals zurück zur Aufgabe, in der wir die Zahl der Vokale in einer Zeichenkette gezählt haben. Für jeden neuen Test sah der Arbeitszyklus meist wie folgt aus: Laden des Programms in den Editor, ändern der betroffenen Testzeile, abspeichern, übersetzen, Programm aufrufen. Das ist auf Dauer recht mühsam und unerquicklich. Wir können nun in einfacher Weise diese beiden Programme zusammenbringen. Statt im Programm `my-echo` nur die Argumente auszugeben, könnten wir auch die Funktion `vokale( argv[ i ] )` entsprechend aufrufen.

**Beispiel:** Für den Programmaufruf `vokale hallo peter ingo llnc 4711` könnte

folgende Ausgabe erscheinen:

```
vokale:
  hallo: 2 Vokale
  peter: 2 Vokale
  ingo: 2 Vokale
  llnc: 0 Vokale
  4711: 0 Vokale
```

Die Aufgabe besteht nun darin, das Programm `my-echo.c` entsprechend zu ergänzen. Da die Änderungen so einfach sind, können wir gleich mit der Implementierung fortfahren.

## 2. Kodierung

```
1  #include <stdio.h>
2
3  int vokale( char * str )
4  {
5      int cnt;
6      if ( str != 0 )  // if (str) would be fine as well
7      {
8          for( cnt = 0; *str; str++ )
9              if ( *str == 'a' || *str == 'A' ||
10                 *str == 'e' || *str == 'E' ||
11                 *str == 'i' || *str == 'I' ||
12                 *str == 'o' || *str == 'O' ||
13                 *str == 'u' || *str == 'U' )
14                  cnt++;
15      }
16      else cnt = -1;
17      return cnt;
18  }
19
20 int main( int argc, char **argv )
21 {
22     int i;
23     printf( "%s:\n", argv[ 0 ] );
24     for( i = 1; i < argc; i++ )
25         printf( "    %s: %d Vokale\n", argv[ i ], vokale
26                ( argv[ i ] ) );
27 }
```

# Übungspaket 26

## Der Datentyp struct

---

### Übungsziele:

1. Organisation von `structs` im Arbeitsspeicher
2. Problemangepasste Verwendung von `structs`.

### Skript:

Kapitel: 53

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Strukturen (`structs`) sind nach den Arrays die zweite Form komplexer Datenstrukturen. Ein wesentliches Element von Strukturen ist, dass sie *unterschiedliche* Variablen zu einem neuen (komplexen) Datentyp zusammenfügen können. Dieser neue Datentyp kann anschließend wie jeder andere verwendet werden. Zu einem sind `structs` ein hervorragendes Strukturierungsmittel. Zum anderen sind sie aufgrund ihrer Eigenschaft des Zusammenfügens *unterschiedlicher* Datentypen eine wesentliche Voraussetzung dynamischer Datenstrukturen (Übungspakete 31 bis 33). Insofern ist dieses Übungspaket von großer Wichtigkeit.



# Teil I: Stoffwiederholung

---

## Aufgabe 1: Strukturen vs. Arrays

Strukturen und Arrays gehören zu den komplexen Datentypen. Für den Programmieranfänger ist es anfangs oft schwer, diese beiden Strukturierungsmethoden auseinander zu halten, obwohl dies für die weitere Arbeit von besonderer Bedeutung ist. Erkläre in eigenen Worten, was **structs** (Strukturen) sind und was der Sinn dahinter ist.

Eine Struktur (**struct**) ist in erster Linie ein Strukturierungselement, dass die Datenorganisation problemadäquat und damit lesbarer machen soll. In der Regel sorgt eine Struktur dafür, dass die Daten im Arbeitsspeicher so angelegt werden, dass zusammengehörende Elemente dicht beieinander liegen. Das typische Beispiel sind alle Angaben zu einer Person. Ein wesentliches Merkmal von Strukturen ist, dass sie Datenelemente *unterschiedlichen* Typs zusammenfassen können.

Strukturen und Arrays unterscheiden sich vor allem in zweierlei Hinsicht. Stelle diese in der folgenden Tabelle einander gegenüber.

Aspekt	Arrays	Strukturen
Datentypen der Elemente bzw. der Komponenten	In einem Array haben <i>immer</i> alle Elemente den selben Typ; es kann keine Unterschiede geben.	Ein <b>struct</b> kann Elemente verschiedener Datentypen zu einem neuen Typ zusammenfassen.
Zugriff auf die einzelnen Elemente bzw. Komponenten	Auf die einzelnen Elemente eines Arrays greift man über seinen Namen und einem Index zu, der in eckigen Klammern steht.  Beispiel: <code>a[ i ]</code>	Auf die einzelnen Komponenten eines <b>structs</b> kann man zugreifen, in dem man beide Namen mittels eines Punktes verbindet.  Beispiel: <code>struktur.komponente</code>

Lassen sich Arrays und **structs** kombinieren

Ja! Sogar beliebig verschachtelt.

## Aufgabe 2: Beispiele

Zeige anhand dreier Beispiele, wie **structs** definiert werden:

```
1 struct cpx_nr { double re, im; } n1, n2; // 2 complex numbers
2
3 struct ic { char c; int i; };           // just a char and an int
4
5 struct cstr {int len; char str[ 20 ] }; // a complex string
```

Zeige anhand einiger Beispiele, wie man auf die einzelnen Komponenten eines `structs` zugreift. Berücksichtige dabei auch mindestens ein Array:

```
1 struct cpx_nr { double re, im; };           // data type complex
2 struct cpx_nr n1, coefs[ 10 ];             // 1 simple, 1 array var
3
4 n1.re = 1.0; n1.im = -1.0;                 // setting the components
5 coefs[ 1 ].im = sin( -0.5 );                // access array elements
6 coefs[ 3 ].re = n1.re * coefs[ 1 ].im;      // more complex
7 coefs[ 2 ] = n1;                           // copying all at once
8 coefs[ 0 ] = coefs[ 2 ];                   // copying all at once
```

Natürlich müssen wir auch Zeiger auf `structs` können. Nehmen wir an, wir haben einen Zeiger `p` auf einen `struct s`, in dem sich eine Komponente `i` vom Typ `int` befindet. Auf welche beiden Arten kann man auf die Komponente `i` desjenigen `structs s` zugreifen, auf das `p` zeigt?

1. `(*p).i = 4711`

2. `p->i = 4711`

Zeichne für folgende `struct`-Definition ein Speicherbild:

```
1 struct two_strings { char *first, *second; };
```

Struktur	Komponente	Wert
two_strings	char *second:	
	char *first :	

Wie viele Bytes belegt ein derartiges `struct`, wenn ein Zeiger vier Bytes belegt? `8`

Nun nehmen wir folgende Variablendefinition an:

```
1 struct two_strings example = { "Enrico", "Johanna" };
```

Wie viele Bytes belegt eine derartige Variable? `8 bzw. 23 Bytes.`

**Anmerkung:** Die Antwort hängt davon ab, ob man die Konstanten mitzählt. Korrekt ist eigentlich 8, da die Variable aus 2 Zeigern besteht; die Namen gehören nicht dazu.

Zeichne hierfür ein Speicherbildchen:

Adresse	Var.	Komponente	Wert	Adresse	Wert
0xFE24	example	char *second:	0xF838	0xF83C	'n' 'n' 'a' '\0'
0xFE20		char *first :	0xF830	0xF838	'J' 'o' 'h' 'a'
				0xF834	'c' 'o' '\0'
				0xF830	'E' 'n' 'r' 'i'

## Aufgabe 3: Rekursive struct-Definitionen

Zur Erinnerung: *Rekursion* bedeutet, sich selbst wieder aufzurufen. Das haben wir bereits bei Funktionen kennengelernt und dort auch deren Vorzüge gesehen. Ein wichtiger Aspekt bei rekursiven Funktionsaufrufen ist, dass man sie irgendwann terminieren muss.

Nun zu den **structs**: Nehmen wir an, wir hätten folgende Definition:

```
1 struct rekursion {
2     int i;
3     struct rekursion noch_was;
4 }
```

Erkläre mit eigenen Worten, weshalb eine derartige rekursive **struct**-Definition in C *nicht* möglich ist:

In oben gegebenem Beispiel beinhaltet das **struct rekursion** sich in sich selbst. Dies würde aber zu einer Endlosrekursion führen, wodurch der Speicherbedarf für eine derartige Struktur unendlich wäre. Die Programmiersprache C bietet keine Möglichkeit, die Verschachtelungstiefe irgendwie zu begrenzen. Dies entspräche den russischen Matroschka Puppen, wenn immer wieder eine neue Puppe um die bestehenden herum platziert würde.

Was wäre aber, wenn wir folgende Definition hätten?

```
1 struct rekursion {
2     int i;
3     struct rekursion *noch_mehr;
4 }
```

Von welchem Datentyp ist die Komponente **i**?

int

Von welchem Datentyp ist die Komponente **noch\_mehr**?

struct rekursion \*  
also ein *Zeiger* auf sich selbst

Dies ist in der Tat in C erlaubt. Wie viele Bytes belegt eine derartige Struktur, wenn ein **int** und ein Zeiger jeweils vier Bytes brauchen (**sizeof(struct rekursion)**)? 8 Bytes

Diskutiere mit den Kommilitonen bei einem Kaffee, Bier oder sonstwas, was man damit machen könnte.

Eine mögliche Antwort wäre: Man könnte mehrere von diesen Strukturen „aneinanderhängen“, wenn man nur mehrere davon von der CPU bekäme. Dazu werden wir in Kapitel 69 und Übungspaket 29 kommen.

## Teil II: Quiz

---

So ein „übliches“ Quiz ist uns zu diesem Thema nicht eingefallen, da die Sachverhalte schlicht zu einfach sind. Daher beschäftigen wir uns diesmal mit Typen und Werten im Rahmen von `structs`. Dies ist insbesondere eine sehr gute Vorbereitung für die dynamischen Datenstrukturen, die wir in den Übungspaketen **31** bis **33** behandeln werden.

### Aufgabe 1: structs, Typen, Zeiger, Werte

Nehmen wir an, wir haben folgendes C-Programm:

```
1 struct combo {
2     char c;
3     int  a[ 2 ];
4 };
5
6 int main( int argc, char **argv )
7 {
8     struct combo test;
9     struct combo a[ 2 ];
10    struct combo *ptr;
11    test.a[ 0 ] = 4711;
12    test.a[ 1 ] = 815;
13    ptr = & test; ptr->c = 'x';
14 }
```

Dann können wir am Ende von Programmzeile 10 folgendes Speicherbildchen erstellen, in dem wie immer alle Adressen frei erfunden sind und wir davon ausgehen, dass Variablen vom Typ `int` sowie Zeiger immer vier Bytes belegen.

Adresse	Var.	Komponente	Wert	Adresse	Var.	Komponente	Wert
0xFE2C		int a[1]	815	0xFE44		int a[1]	24
0xFE28		int a[0]	4711	0xFE40		int a[0]	-273
0xFE24	test	char c	'x'	0xFE3C	a[1]	char c	'a'
0xFE20	ptr		: 0xFE24	0xFE38		int a[1]	24
				0xFE34		int a[0]	-273
				0xFE30	a[0]	char c	'a'

Ergänze zunächst die Effekte der Programmzeilen 11 bis 13 im obigen Speicherbildchen.

Vervollständige nun die folgende Tabelle. Die Ausdrücke werden im Verlaufe des Quiz immer schwieriger. Im Einzelfalle lohnt es sich, entweder ein kurzes Testprogramm zu schreiben und/oder mit den Betreuern zu diskutieren.

Ausdruck	Type	Wert	Anmerkung
<code>test</code>	<code>struct combo</code>	----	Die Struktur ab 0xFE24
<code>sizeof( test )</code>	<code>int</code>	12	
<code>&amp; test</code>	<code>struct combo *</code>	0xFE24	
<code>ptr</code>	<code>struct combo *</code>	0xFE24	
<code>sizeof( ptr )</code>	<code>int</code>	4	
<code>*ptr</code>	<code>struct combo</code>	----	Dies ist die Struktur <code>test</code>
<code>sizeof( *ptr )</code>	<code>int</code>	12	Da <code>*ptr</code> die gesamte Struktur ist
<code>test.c</code>	<code>char</code>	'x'	
<code>ptr-&gt;a[0]</code>	<code>int</code>	4711	
<code>ptr-&gt;c</code>	<code>char</code>	'x'	
<code>&amp; ptr</code>	<code>struct combo **</code>	0xFE20	
<code>test.a[0]</code>	<code>int</code>	4711	
<code>&amp;(test.a[0])</code>	<code>int *</code>	0xFE28	
<code>sizeof( a )</code>	<code>int</code>	24	
<code>&amp;(a[ 1 ].a[ 0 ])</code>	<code>int *</code>	0xFE40	
<code>*(a + 1)</code>	<code>struct combo</code>	----	Identisch mit <code>a[ 1 ]</code> (ab 0xFE3C)

Die folgenden Quiz-Fragen sind sehr schwer!

Ausdruck	Type	Wert	Anmerkung
<code>sizeof( test.a )</code>	<code>int</code>	8	Hier handelt es sich tatsächlich um das ganze Array <code>a</code>
<code>test.a</code>	<code>int []</code>	0xFE28	Dies ist wieder „nur“ die Adresse des ersten Elementes, also <code>&amp; a[ 0 ]</code>
<code>a</code>	<code>struct combo []</code>	0xFE30	Das ist das Array bestehend aus zwei Elementen vom Typ <code>struct combo</code>

Betrachte noch die folgenden Anweisungen. Trage die Auswirkungen der einzelnen Anweisung in das Speicherbildchen der vorherigen Seite ein.

```

1 a[ 0 ].c = 'a';
2 (*(a + 0)).a[ 0 ] = -273;
3 (a + 0)->a[ 1 ] = 24;
4 a[ 1 ] = a[ 0 ];

```

## Teil III: Fehlersuche

---

### Aufgabe 1: Definition und Verwendung von structs

Nach der Vorlesung hat DR. STRUCKI versucht, ein paar `structs` zu programmieren. Offensichtlich benötigt er eure Hilfe, da ihm nicht alles klar geworden ist.

```
1 struct cpx double re, im;                                // a complex number
2 struct ivc { double len; cpx cx_nr; };                  // plus len
3
4 int main( int argc, char **argv )
5 {
6     struct cpx cx, *xp1, *xp2;
7     struct ivc vc, *v_p;
8     struct ivc i_tab[ 2 ];
9
10    cx.re = 3.0; im = -4.0;
11    vc.cx_nr = cx; vc.len = 5.0;
12    vc.re = 3.0; vc.cx_nr.im = 4.0;
13
14    xp1 = & cx; xp2 = vc.cx_nr;
15    xp1->re = 6.0; xp1.im = -8.0;
16    *xp2 = *xp1; vc.len = 10.0;
17
18    cx *= 2;
19    vc.cx_nr += cx;
20
21    (*(i_tab + 0)) = vc;
22    (i_tab + 1)->cx_nr = cx;
23 }
```

Zeile	Fehler	Erläuterung	Korrektur
1	{ } fehlen	Bei einem <code>struct</code> <i>müssen</i> die Komponenten innerhalb { } stehen.	<code>{double ... };</code>
2	„struct“ fehlt	Bei der Verwendung von Strukturen muss das Wort <code>struct</code> dastehen.	<code>struct cpx</code>
10	<code>cx.</code> fehlt	Beim <i>Zugriff</i> auf die Komponenten muss immer auch die eigentliche Struktur angegeben werden.	<code>cx.im</code>
11		Hier ist alles richtig ;-)	

Zeile	Fehler	Erläuterung	Korrektur
12	<code>cx_nr.</code> fehlt	Hier fehlt ein Teil der kompletten Namensgebung: Beim Zugriff auf eine Komponente müssen immer alle Teile angegeben werden.	<code>vc.cx_nr.re</code>
14	<code>&amp;</code> fehlt	<code>xp2</code> ist ein Zeiger. Entsprechend muss rechts auch die Adresse gebildet werden.	<code>&amp;vc.</code>
15	<code>.</code> falsch	<code>xp1</code> ist ein Zeiger. Entsprechend greift man auf die Komponenten entweder mittels <code>-&gt;</code> oder <code>*(...).</code> zu.	<code>xp1-&gt;</code>
18	<code>cx *= 2</code>	Schön gedacht, aber die einzelnen Komponenten muss man schon einzeln verändern.	<code>cx.re *= 2 ...</code>
19	<code>+=</code>	dito.	
21		Sieht komisch aus, aber hier ist alles richtig.	
22		Sieht komisch aus, aber hier ist alles richtig.	

### Programm mit Korrekturen:

```

1 struct cpx { double re, im; };           // a complex number
2 struct ivc { double len; struct cpx cx_nr; };      // plus len
3
4 int main( int argc, char **argv )
5 {
6     struct cpx cx, *xp1, *xp2;
7     struct ivc vc, *v_p;
8     struct ivc i_tab[ 2 ];
9
10    cx.re = 3.0; cx.im = -4.0;
11    vc.cx_nr = cx; vc.len = 5.0;
12    vc.cx_nr.re = 3.0; vc.cx_nr.im = 4.0;
13
14    xp1 = & cx; xp2 = & vc.cx_nr;
15    xp1->re = 6.0; xp1->im = -8.0;
16    *xp2 = *xp1; vc.len = 10.0;
17
18    cx.re *= 2; cx.im *= 2;
19    vc.cx_nr.re += cx.re; vc.cx_nr.im += cx.im;
20
21    (*(i_tab + 0)) = vc;
22    (i_tab + 1)->cx_nr = cx;
23 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Eine einfache Personenverwaltung

### 1. Aufgabenstellung

Gegenstand dieser Übungsaufgabe ist die Entwicklung eines kleinen Programms, das für uns eine Namensliste „verwaltet“. Damit ihr euch auf das Einüben von **structs** konzentrieren könnt, haben wir diese eigentlich doch recht komplexe Aufgabe für euch wie folgt stark vereinfacht:

1. Die Angaben zu einer Person bestehen lediglich aus Name und Alter. Desweiteren legen wir fest, dass ein Name nur aus *genau einem* Zeichen besteht.
2. Alle Personen sollen in einer Tabelle verwaltet werden. Die Zahl der Personen sowie die Daten stehen von Anfang an fest, können also direkt in das Programm integriert werden.
3. Im Rahmen unserer hypothetischen Anwendung muss unser Programm drei Dinge können:
  - (a) Sortieren der Personentabelle aufsteigend nach dem Namen.
  - (b) Sortieren der Personentabelle aufsteigend nach dem Alter.
  - (c) Drucken der vorliegenden Tabelle.
4. Ferner benötigen wir ein Hauptprogramm, dass uns die Funktionalität gemäß obiger Beschreibung bestätigt.

### 2. Vorüberlegungen und Hinweise

Um euch ein wenig zu helfen, haben die Doktoranden mal ein bisschen laut nachgedacht und das Gesagte protokolliert. Dabei kamen folgende Ideen auf:

1. Wir sollen also Personen verwalten, die einen Namen vom Typ **Zeichen** sowie ein Alter vom Typ **Ganzzahl** haben. Damit die Daten beim späteren Sortieren nicht durcheinander geraten, sollten wir hierfür am besten eine Struktur definieren, in der alle Angaben Platz haben.
2. Wir brauchen eine Funktion, die uns die Tabelle auf dem Bildschirm ausgibt. Hier reicht eine Funktion, da es ihr ja egal ist, ob die Tabelle sortiert ist oder nicht.
3. Da wir die Personentabelle mal nach dem Namen und mal nach dem Alter sortieren sollen, wäre hier jeweils eine entsprechende Funktion sinnvoll, die die



Sortierung nach dem jeweiligen Kriterium vornimmt. Macht also zwei unterschiedliche Sortierfunktionen.

4. Für die Sortierfunktion können wir uns am **Bubble-Sort**-Algorithmus orientieren, den wir bereits in Übungspaket 15 hatten. Nur müssen wir dann noch einige Variablen und Tauschoperationen anpassen. Aber das haben wir ja schon im ersten Teil dieses Übungspaketes wiederholt.
5. Schließlich benötigen wir noch ein Hauptprogramm. Am Anfang basteln wir uns einfach alles zusammen, dann geben wir die Daten zur Kontrolle einmal aus, sortieren sie nach den Namen, geben sie zur Kontrolle aus, sortieren sie nach dem Alter und geben sie noch ein letztes Mal aus.
6. Jetzt sollte eigentlich alles klar sein, sodass wir mit der Arbeit anfangen können.

### 3. Pflichtenheft

Aufgabe	: Programm zur Verwaltung von Personen, die einen Namen und ein Alter haben, Sortiermöglichkeiten nach jeweils einem der beiden Angaben.
Eingabe	: keine Eingaben, da direkte Kodierung.
Ausgabe	: Tabelle in unsortierter sowie in nach Namen bzw. Alter sortierten Form.
Sonderfälle	: keine.
Funktionsköpfe:	<code>pri_tab( FILE *fp, struct person *tab, int size )</code> <code>sort_name( struct person *tab, int size )</code> <code>sort_age( struct person *tab, int size )</code>

### 4. Implementierung

Da wir bereits alle benötigten Algorithmen in früheren Übungspaketen eingehend behandelt haben, können wir direkt mit der Kodierung anfangen. Falls dennoch Fragen sein sollten, einfach die Betreuer konsultieren.

### 5. Kodierung

**Definition des structs:**

```
1 #include <stdio.h>
2
3 struct person          // struct for a person
4 {
5     int    age;
6     char  name;
7 };
```

### Eine Funktion zum Drucken der Tabelle:

```
8 int prt_tab( struct person *tab, int size )
9 {
10     int i;
11     if ( tab )
12         for( i = 0; i < size; i++ )
13             printf("Person %3d: Name: %c Alter: %d\n",
14                 i, tab[ i ].name, tab[ i ].age );
15     else printf( "prt_tab: Tabelle vergessen\n" );
16     return tab != 0;
17 }
```

### Eine Funktion zum Vertauschen zweier Elemente:

```
18 // this function makes life easier, we need it twice
19 void p_swap( struct person *p1, struct person *p2 )
20 {
21     struct person tmp;
22     tmp = *p1; *p1 = *p2; *p2 = tmp;
23 }
```

### Eine Funktion zum Sortieren nach den Namen:

```
24 int sort_name( struct person *tab, int size )
25 {
26     int i, j;
27     if ( tab )
28         for( i = 0; i < size - 1; i++ )
29             for( j = 0; j < size - i - 1; j++ )
30             {
31                 if ( tab[ j ].name > tab[ j+1 ].name )
32                     p_swap( & tab[ j ], & tab[ j + 1 ] );
33                 // this time array notation...
34             }
35     else printf( "sort_name: Tabelle vergessen\n" );
36     return tab != 0;
37 }
```

### Eine Funktion zum Sortieren nach dem Alter:

```
38 int sort_age( struct person *tab, int size )
39 {
40     int i, j;
41     if ( tab )
42         for( i = 0; i < size - 1; i++ )
43             for( j = 0; j < size - i - 1; j++ )
44                 {
45                     if ( tab[ j ].age > tab[ j + 1 ].age )
46                         p_swap( tab + j, tab + j + 1 );
47                     // and now pointer arithmetic
48                 }
49     else printf( "sort_age: Tabelle vergessen\n" );
50     return tab != 0;
51 }
```

### Schließlich das Hauptprogramm:

```
52 int main(int argc, char** argv)
53 {
54     struct person ptab[] = {
55         {4, 'g'}, {12, 'm'}, {9, '9'}, {45, 'k'},
56         {1, 'c'}, {1234647675, 'b'}, {-9, 'q'},
57         {31, 'd'}, {31, 'l'}, {22, 'o'}};
58     #define PTAB_SIZE    (sizeof(ptab)/sizeof(ptab[0]))
59
60     prt_tab( ptab, PTAB_SIZE );
61     sort_name( ptab, PTAB_SIZE );
62     printf("-----\n");
63     prt_tab( ptab, PTAB_SIZE );
64     sort_age( ptab, PTAB_SIZE );
65     printf("-----\n");
66     prt_tab( ptab, PTAB_SIZE );
67
68     return 0;
69 }
```

# Aufgabe 2: Einfache Verwaltung von Personennamen

## 1. Aufgabenstellung

Diese Aufgabe ist so ähnlich wie die Vorherige. Nur wollen wir diesmal, dass die Tabelle Personen verwaltet, die sowohl einen richtigen Vor- als auch einen richtigen Nachnamen haben. Beispiel: "Herr Andrea Neutrum". Die Tabelle soll diesmal aber nicht sortiert werden. Vielmehr wollen wir nach Einträgen bezüglich eines Vor- oder Nachnamens suchen können und ggf. den ganzen Personeneintrag ausgeben. Wie schon in der vorherigen Aufgabe gelten wieder folgende Randbedingungen:

1. Wir benötigen eine Tabelle, in der alle Personen enthalten sind. Sowohl die Tabellengröße als auch die Namen der zu verwaltenden Personen stehen von Anfang an fest, sodass diese statisch in die Tabelle eingetragen werden können.
2. Wir benötigen eine Funktion zum Drucken, zwei Suchfunktionen (Suchen nach Vor- bzw. Nachname) und ein Hauptprogramm zur Demonstration der Funktionsfähigkeit.

## 2. Vorüberlegungen und Hinweise

Hier wieder ein paar Hilfestellungen unserer betreuenden Doktoranden:

1. Nach der vorherigen Übung ist diese ja eigentlich schon zu einfach. Wir benötigen lediglich eine Struktur, in der zwei Namen vom Typ `char *` Platz haben.
2. Die vier benötigten Funktionen stehen schon in der Aufgabenstellung. Auch das Finden von Tabelleneinträgen haben wir bereits in Übungspaket 15 eingeübt. Das Hauptprogramm zum Funktionstest können wir wieder genau so wie in der vorherigen Übung aufbauen: Daten eintragen, Suchen, Drucken, etc.
3. Das Suchen erledigen wir am besten in getrennten Funktionen und geben einen entsprechenden Index zurück. Dieser nimmt einen speziellen Wert an, wenn der Eintrag nicht gefunden wurde.

## 3. Pflichtenheft

Aufgabe	: Programm zur Verwaltung von Personen, die einen richtigen Vor- und Nachnamen haben. Suchmöglichkeiten nach jeweils einer der beiden Angaben.
Eingabe	: keine Eingaben, da direkte Kodierung.
Ausgabe	: Tabelle in unsortierter Form, gefundene Personen.
Sonderfälle	: keine.
Funktionsköpfe:	<pre>pri_tab( FILE *fp, struct person *tab, int size ) find_first(char * name, struct person *tab, int size ) find_last( char * name, struct person *tab, int size )</pre>

## 4. Implementierung

Aufgrund der vielen Vorarbeiten können wir direkt mit der Kodierung beginnen. Falls dennoch Fragen sein sollten, einfach direkt die Betreuer konsultieren.

## 5. Kodierung

**Definition des structs:**

```
1  #include <stdio.h>
2
3  struct person          // struct for a person
4  {
5      char *first, *last;
6  };
```

**Zwei Funktionen zum Drucken:**

```
7  void prt_person( FILE *fp, struct person ps )
8  {
9      fprintf( fp, "Person: %s %s\n", ps.first, ps.last );
10 }
11
12 void prt_notfound( FILE *fp, char *name )
13 {
14     fprintf( fp, "%s: nicht vorhanden\n", name );
15 }
```

**Zwei Such-Funktionen:**

```
16 int find_first( char * name, struct person *tab, int size )
17 {
18     int i;
19     for( i = 0; i < size; i++ )
20         if ( ! strcmp( name, tab[ i ].first ) )
21             return i;
22     return -1;
23 }
24
25 int find_last( char * name, struct person *tab, int size )
26 {
27     int i;
28     for( i = 0; i < size; i++ )
29         if ( ! strcmp( name, tab[ i ].last ) )
30             return i;
31     return -1;
32 }
```

Schließlich das Hauptprogramm:

```
33 int main(int argc, char** argv)
34 {
35     int i;
36     struct person ptab[] = {
37         { "Enrico", "Heinrich" }, { "Ralf", "Joost" },
38         { "Matthias", "Hinkfoth" }, { "Rene", "Romann" },
39         { "Ralf", "Warmuth" }, { "Ralf", "Salomon" } };
40     #define PTAB_SIZE    (sizeof(ptab)/sizeof(ptab[0]))
41
42     // test output
43     for( i = 0; i < PTAB_SIZE; i++ )
44         prt_person( stdout, ptab[ i ] );
45     printf( "-----\n" );
46
47     i = find_first( "Matthias", ptab, PTAB_SIZE );
48     if ( i != -1 )
49         prt_person( stdout, ptab[ i ] );
50     else prt_notfound( stdout, "Matthias" );
51
52     i = find_last( "Heinrich", ptab, PTAB_SIZE );
53     if ( i != -1 )
54         prt_person( stdout, ptab[ i ] );
55     else prt_notfound( stdout, "Heinrich" );
56
57     i = find_first( "Superman", ptab, PTAB_SIZE );
58     if ( i != -1 )
59         prt_person( stdout, ptab[ i ] );
60     else prt_notfound( stdout, "Superman" );
61
62     return 0;
63 }
```

# Übungspaket 27

## Definition eigener Datentypen

---

### Übungsziele:

1. Umgang mit der `typedef`-Anweisung

### Skript:

Kapitel: 54

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Für viele Programmieranfänger und leicht Fortgeschrittene erscheint die `typedef`-Anweisung einigermaßen überflüssig. Als Lernender mag man sogar annehmen, dass es diese Anweisung nur zu ihrer Verwirrung gibt. Aber auch diese Anweisung hat, wie so viele andere Sprachelemente, in der Programmiersprache C ihren Sinn und ihre Berechtigung. In diesem Übungspaket werden wir die Anwendung dieser Anweisung von verschiedenen Seiten beleuchten und hoffen, dass bei euch darüber auch weitere Einsichten von ganz alleine kommen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Motivation und Datentypen

Erkläre kurz in eigenen Worten, was ein Datentyp ist.

Ein Datentyp hat einen Namen und legt fest, wie viele Bits er hat und wie diese Bits zu interpretieren sind. Ein typisches Beispiel ist der Datentyp `char`: Er umfasst meist acht Bits und erlaubt die Speicherung normaler ASCII Zeichen. Ein etwas komplizierteres Beispiel ist `struct cplx { double re, im; }`, der aus zwei `double`-Werten besteht und gemeinhin als „komplexe Zahl“ verstanden wird.

Bei welchen Datenstrukturen fängt die Sache für dich an, unübersichtlich zu werden?

Zeiger auf Zeiger, Zeiger auf Zeiger auf Zeiger, Zeiger auf Strukturen

Ein weit verbreiteter Problembereich betrifft die Definition von Zeigern. Einen Zeiger auf einen `int`-Wert könnten wir beispielsweise auf eine der beiden folgenden Arten definieren: `int *p` und `int* p`. Der immer wiederkehrende Streit betrifft die Frage, wo denn nun der Stern am besten stehen sollte. Eine Antwort lautet: dem Compiler ist es so was von egal... Er mag genauso gerne die Variante `int*p`.

Die Anhänger der Variante `int* p` argumentieren, dass der Stern zum Typ gehört und deshalb beim Wort „`int`“ stehen sollte. Nehmen wir an, wir hätten folgende Definition: `int* p, q`. Die Variablen `p` und `q` haben die folgenden Typen: `p: int *` `q: int`. Dies ist oft nicht das, was man eigentlich haben möchte.

Nehmen wir an, wir hätten folgendes Programm:

```
1 #define INT_PTR int *
2
3 INT_PTR      p, q;
```

von welchem Typ sind jetzt die beiden Variablen? `p: int *` `q: int`

Schlussfolgerung: Auch diese `#define`-Variante bringt uns kein Stück weiter.

Was wir bräuchten, bildlich gesprochen, ist ein Mechanismus, der uns das Sternchen vor *jede* Variable schreibt. Und genau das macht `typedef`.

Wie lautet die Syntax der `typedef`-Anweisung? `typedef <alter Typ> <neuer Typ>`

Bei der `typedef`-Anweisung sind zwei Dinge wichtig:

1. Alle neuen Typen werden immer auf die Basistypen zurückgeführt.
2. Die neuen, selbstdefinierten Typen kann man so verwenden, als gäbe es sie schon immer.



## Teil II: Quiz

---

### Aufgabe 1: typedef I: einfache Beispiele

Nehmen wir an, wir hätten ein Programm, in dem die folgenden Typ- und Variablendefinitionen gegeben sind:

```
1 typedef int myInt;
2 myInt mi_index;
3
4 typedef char Zeichen, charLabel;
5 Zeichen lab_1;
6 charLabel lab_2;
7
8 typedef int personalID, *PID_PTR;
9 personalID class_mate, *mate_p;
10 PID_PTR cm_index, *cm_pp;
11
12 typedef personalID *LOOPI, **LOOPII;
```

Trage nun in folgende Tabelle jeweils den neu definierten Datentyp bzw. den Datentyp der angegebenen Variablen ein. Zur Illustration sind die ersten beiden Beispiele bereits vorgegeben.

Zeile	Name	resultierender C-Typ	Beschreibung
1	myInt	int	einfacher Datentyp int
2	mi_index	int	einfacher Datentyp int
4	Zeichen	char	einfacher Datentyp char
	charLabel	char	einfacher Datentyp char
5	lab_1	char	einfacher Datentyp char
6	lab_2	char	einfacher Datentyp char
8	personalID	int	einfacher Datentyp int
	PID_PTR	int *	Zeiger auf int
9	class_mate	int	einfacher Datentyp int
	mate_p	int *	Zeiger auf int
10	cm_index	int *	Zeiger auf int
	cm_pp	int **	Zeiger auf Zeiger auf int
12	LOOPI	int *	Zeiger auf personalID und damit Zeiger auf int
	LOOPII	int **	Zeiger auf Zeiger auf personalID und damit Zeiger auf Zeiger auf int

## Aufgabe 2: typedef II: Arrays und Zeiger

Nun schauen wir ein wenig auf Arrays und Zeiger:

```
1 typedef char *STRING;
2 typedef STRING *STR_PTR;
3
4 typedef char LABEL[ 5 ];
5 LABEL label, *l_ptr;
6
7 typedef int VEC[ 3 ], *XP;
8 VEC vec, *vp1;
9 XP ip;
10
11 typedef VEC *V_PTR;
12 V_PTR vp2, *vp3;
```

Trage in folgende Tabelle jeweils den neu definierten Datentyp bzw. den Datentyp der angegebenen Variablen ein. Zur Illustration sind die ersten beiden Beispiele bereits vorgegeben. Die übliche Annahme ist, dass Zeiger und `int`-Werte jeweils vier Bytes benötigen.

Zeile	Name	resultierender C-Typ	Beschreibung
1	STRING	char *	Zeiger auf char
	STR_PTR	char **	Zeiger auf STRING ergibt Zeiger auf Zeiger auf char
4	LABEL	char [ 5 ]	char-Array mit fünf Elementen. Beispiel: LABEL l und char l[ 5 ] sind identisch
5	label	char [5]	Array mit fünf Elementen vom Typ char Equivalent: char label[ 5 ]
	l_ptr	char (*) [5]	Zeiger auf ein Array mit fünf Elementen: sizeof(* l_ptr) == 5
7	VEC	int [3]	Array mit drei int-Werten
	XP	int *	Zeiger auf <i>einen</i> int-Wert
8	vec	int vec[3]	Array mit drei int-Werten
	vp1	int (*) [3]	Zeiger auf VEC ergibt Zeiger auf int [3] sizeof(*vp1) == 12 und sizeof(**vp1) == 4
9	ip	int*	Zeiger auf int; nahezu langweilig
11	V_PTR	int (*) [3]	Zeiger auf ein int-Array mit drei Elementen
12	vp2	int (*) [3]	<i>absolut</i> identisch mit vp1
	vp3	int (**) [3]	Zeiger auf V_PTR ist Zeiger auf Zeiger auf VEC ist Zeiger auf Zeiger auf int [3] sizeof(*vp3) == 4, sizeof(**vp3) == 12 und sizeof(**vp3) == 4

## Aufgabe 3: typedef III: Zeiger Strukturen

Zum Schluss noch ein paar Beispiele zu Strukturen:

```
1 typedef struct bsp { char c1; char c2; } STRUCT1;
2
3 typedef struct { char c1, c2; } STRUCT2, *S2P;
4
5 typedef struct xyz { int *ip; int i; } XYZ;
6
7 typedef struct _element {
8     struct _element *next;
9     int value;
10 } ELEMENT, *E_PTR;
```

Trage in folgende Tabelle jeweils den neu definierten Datentyp bzw. den Datentyp der angegebenen Variablen ein.

Zeile	Name	„ausgeschriebener“ Typ
1	STRUCT1	<code>struct</code> bestehend aus zwei Zeichen. Diese Struktur ist auch unter dem Namen <code>struct bsp</code> bekannt.
3	STRUCT2	identisch mit <code>STRUCT1</code> , aber keine Alternative verfügbar, da nach <code>struct</code> kein Name steht.
	S2P	Zeiger auf eine Struktur, die aus zwei Zeichen besteht
5	XYZ	Dies ist eine Struktur mit einem Zeiger auf ein <code>int</code> sowie einem <code>int</code> -Wert.
10	ELEMENT	<code>ELEMENT</code> ist eine Struktur, die einen <code>int</code> -Wert sowie einen <i>Zeiger</i> auf sich selbst enthält.
	E_PTR	Zeiger auf die Struktur <code>ELEMENT</code>

## Teil III: Fehlersuche

---

### Aufgabe 1: Variablendefinitionen mittels typedef

Offensichtlich benötigt auch DR. TYPE-DE-FOOL etwas Unterstützung.

```
1 #define SPECIAL_CHAR    char *
2
3 SPECIAL_CHAR    char_ptr, one_char;
4 SPECIAL_CHAR    char_ptr1, char_ptr2;
5
6 typedef int myInt alsoMyInt;
7
8 myInt *one_int;
9 alsoMyInt i_ptr1, i_ptr2
```

Zeile	Fehler	Erläuterung	Korrektur
3		Unglaublich, aber hier ist alles richtig.	
4	* fehlt	Die zweite Variable ( <code>char_ptr2</code> ) ist <i>nur</i> ein Zeichen und kein Zeiger, da es sich bei <code>SPECIAL_CHAR</code> um ein <code>#define</code> und nicht ein <code>typedef</code> handelt.	* <code>char_ptr2</code>
6	Tippfehler	Die Anweisung <code>typedef</code> ist falsch geschrieben	<code>typedef</code>
6	Komma fehlt	Das Komma zwischen den beiden Typen <code>myInt</code> und <code>alsoMyInt</code> fehlt	<code>myInt,</code>
8	* zu viel	Da hier eine einfache <code>int</code> -Variable gewünscht wird, ist das Sternchen zu viel.	<code>myInt one_int</code>
9	* fehlen	Der <code>alsoMyInt</code> ein normaler <code>int</code> -Typ ist, müssen für die zwei Zeiger Sternchen verwendet werden.	<code>*i_ptr1,</code> <code>*i_ptr2</code>

Programm mit Korrekturen:

```
1 #define SPECIAL_CHAR    char *
2
3 SPECIAL_CHAR    char_ptr, one_char;
4 SPECIAL_CHAR    char_ptr1, *char_ptr2;
5
6 typedef int myInt, alsoMyInt;
7
8 myInt one_int;
9 alsoMyInt *i_ptr1, *i_ptr2
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Eigenständige Definition neuer Datentypen

In dieser Aufgabe sollt ihr gemäß Spezifikation eigene Datentypen mittels `typedef` bilden.

1. Definiere einen eigenen Typ `myDP`, der ein Zeiger auf ein `double` ist.

```
typedef double *myDP;
```

2. Definiere einen eigenen Typ `myCP` sowie `myCPP` der ein Zeiger auf ein `char` bzw. ein Zeiger auf ein Zeiger auf ein `char` ist.

```
typedef char *myCP, **myCPP;
```

3. Definiere einen Typ `myTag`, der genau zehn Zeichen aufnehmen kann.

```
typedef char myTag[ 10 ];
```

4. Definiere einen Typ `myTagPtr`, der ein Zeiger auf den Typ `myTag` ist.

```
typedef myTag *myTagPtr;
```

5. Definiere einen Typ `C_ARRAY`, der vier Zeiger auf ein Zeichen besitzt. Führe die Definition einmal mittels des Typs `myCP` aus der zweiten Aufgabe und einmal direkt durch.

```
typedef myCP C_ARRAY[ 4 ];
```

```
typedef char *(C_ARRAY[ 4 ]);
```

6. Definiere eine Struktur `Auto`, die die Zahl der Räder sowie die Motorleistung aufnehmen kann.

```
typedef struct { int raeder; int PS; } Auto;
```

7. Definiere einen Zeigertyp `AutoPtr`, der auf ein `Auto` zeigt.

```
typedef Auto *AutoPtr;
```

# Übungspaket 28

## Module und getrenntes Übersetzen

---

### Übungsziele:

1. Verteilen von Programmteilen auf mehrere Dateien
2. Richtige Verwendung der Header-Dateien
3. Richtiger Umgang mit dem C-Compiler

### Skript:

Kapitel: 55 und 39 sowie insbesondere auch Übungspaket 17

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Im Gegensatz zu vielen klassischen Programmiersprachen bietet C die Möglichkeit, das Programm auf mehrere Dateien zu verteilen. Diese Verteilung gestattet es, nur diejenigen Dateien zu übersetzen, die sich auch wirklich geändert haben; die anderen .o-Dateien können bleiben, wie sie sind. Diese Option führt zu übersichtlichen Quelltext-Dateien und kann die notwendige Übersetzungszeit bei großen Programmen deutlich reduzieren. Allerdings tritt beim Aufteilen des Quelltextes auf mehrere Dateien das Problem der Konsistenzhaltung auf: es kann vorkommen, dass eine Funktion in einer Datei anders definiert ist, als sie in einer anderen Datei aufgerufen wird. Hier hilft der richtige Umgang mit den Header-Dateien und den `#include`-Direktiven, was wir im Rahmen dieses Übungspaketes probieren wollen.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Fragen zum Compiler

Wie heißen die vier Teilprogramme, die der Compiler nacheinander aufruft?

- |  |   |
|--|---|
| 1. <input type="text" value="Präprozessor"/> | 2. <input type="text" value="(eigentlicher) Compiler"/> |
| 3. <input type="text" value="Assembler"/>    | 4. <input type="text" value="Linker/Binder"/>           |

Welcher Teil des Compilers bearbeitet die `#include`-Direktiven?

Welche Phasen beinhaltet der eigentliche Übersetzungsvorgang?

Mittels welcher Option hört der Compiler *vor* dem Binden auf?

Gib hierfür ein Beispiel.

In welcher Datei landet das Ergebnis der eigentlichen Übersetzung?

Was „weiß“ der Compiler nach dem Übersetzen von `datei.c`?

Nehmen wir nun an, wir würden zwei Dateien übersetzen. Was „weiß“ der Compiler von der ersten Datei, beim Übersetzen der zweiten, wenn wir folgende Kommandos verwenden?

`gcc -c datei-1.c` und `gcc -c datei-2.c`

`gcc -c datei-1.c datei-2.c`

Erkläre nochmals kurz in eigenen Worten, was die `#include`-Direktive macht.

Vervollständige die beiden folgenden Beispiele:

#include-Direktive	Effekt
<code>#include &lt;math.h&gt;</code>	Fügt die Datei <code>math.h</code> ein.
<code>#include "vowels.h"</code>	Fügt die Datei <code>vowels.h</code> ein; diese wird zunächst im aktuellen Verzeichnis gesucht.

Erkläre mit eigenen Worten, wofür man üblicherweise sogenannte `.h`-Dateien verwendet:

Die Endung `.h` ist eine Konvention und steht für Header-Datei. In diese Dateien schreibt man üblicherweise Definitionen (`#define`-Direktiven) und Funktionsdeklarationen hinein, damit der Compiler beim Übersetzen die definierten Werte sowie die Signaturen der Funktionen kennt. Die Datei `stdio.h` ist eine der besten Beispiele hierfür.

## Teil II: Quiz

---

### Aufgabe 1: Übersetzen und Binden (Linken)

Im Folgenden gehen wir davon aus, dass wir eine Reihe von Dateien haben und diese mittels des C-Compilers übersetzen und/oder binden wollen. In der ersten Spalte steht jeweils das auszuführende Kommando. In der zweiten soll jeweils stehen, welche Dateien übersetzt werden. In der dritten Spalte soll stehen, ob ein lauffähiges Programm zusammengebunden wird und wie ggf. dessen Name lautet. Vervollständige die fehlenden Spalten.

Kommando	übersetzte Dateien	gebundenes Programm
<code>gcc datei-1.c</code>	<code>datei-1.c</code>	<code>a.out</code>
<code>gcc datei-1.c -o badman</code>	<code>datei-1.c</code>	<code>badman</code>
<code>gcc -c a.c b.c</code>	<code>a.c, b.c</code>	----
<code>gcc a.o b.o</code>	----	<code>a.out</code>
<code>gcc -o temperatur x.c y.o</code>	<code>x.c</code> ( <code>y.o</code> gibt es schon)	<code>temperatur</code>
<code>gcc -c inge.c peter.c</code>	<code>inge.c, peter.c</code>	----
<code>gcc -o supi supi.c</code>	<code>supi.c</code>	<code>supi</code>

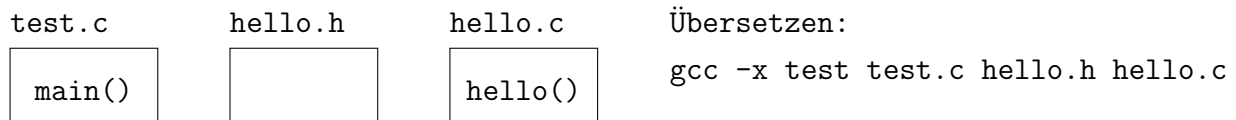


## Teil III: Fehlersuche

---

### Aufgabe 1: Eine einfache verteilte Anwendung

Chefprogrammierer DR. MODELL-HAMMER hat versucht, eine erste kleine verteilte Anwendung zu schreiben. Dabei hat er folgende Struktur im Sinn gehabt:



In dieser Struktur befindet sich in der Datei `test.c` das Hauptprogramm, das seinerseits die Funktion `hello()` aufruft, die einen netten Begrüßungstext ausgibt. Diese Funktion gibt den Geburtstag, der als Struktur definiert ist, an die aufrufende Stelle zurück. Leider hat auch DR. MODELL-HAMMER ein paar kleine Fehler gemacht ... er ist desperate ...

**Das Hauptprogramm: test.c**

```
1 #include <stdio.h>
2 #include "hello.c"
3
4 int main( int argc, char **argv )
5 {
6     struct birthdate birth = hello( "frida" );
7     printf( "my birthday is: %d. %d. %d\n",
8           birth.day, birth.month, birth.year );
9 }
```

**Die Header-Datei: hello.h**

```
1 struct birthdate hello( char *name );
```

**Das Modul: hello.c**

```
1 #include <stdio.h>
2 #include "hello.h"
3
4 struct birthdate { int day; int month; int year; };
5
6 struct birthdate hello( char *name )
7 {
8     struct birthdate birth;
9     printf( "hello %s, my darling!\n", name );
10    birth.day = 30; birth.month = 2; birth.year = 1991;
11    return birth;
12 }
```

Zeile	Fehler	Erläuterung	Korrektur
hello.c: 4	struct ...	Das geht leider schief. Der Compiler kann die Datei <b>hello.c</b> einwandfrei übersetzen. Da aber im Regelfalle alle anderen Dateien nicht eine <b>.c</b> sondern eine <b>.h</b> -Datei einbinden, bekommt der Compiler dort nicht mit, was <b>struct birthdate</b> sein soll. Mit anderen Worten: Diese Definition gehört in die <b>.h</b> -Datei, damit der Compiler auch bei den anderen <b>.c</b> -Dateien weiß, was mit <b>struct birthdate</b> gemeint ist.	Verschieben nach <b>hello.h</b>
hello.h: 1	Fehlende Definition	Wie eben erläutert, muss hier die Definition für die Struktur <b>birthdate</b> hin.	dito.
test.c: 2	hello.c	Das Einbinden einer anderen <b>.c</b> -Datei wollen wir ja gerade nicht. Dann hätten wir hier gleich alle Zeilen der Datei <b>hello.c</b> hinschreiben können. Wir wollen ja gerade, dass die C-Anweisungen in getrennten Dateien bleiben. Daher bindet man hier üblicherweise eine Header-Datei <b>.h</b> ein. Diese dient dann dem Compiler, damit er weiß, was los ist und somit leichter Fehler entdecken kann, was wiederum uns bei der Programmentwicklung <i>sehr</i> hilft.	dito.
Kommando	hello.h zu viel	Beim Aufruf des Compilers müssen wir die <b>.c</b> -Dateien angeben; die <b>.h</b> -Dateien werden durch die <b>#include</b> -Direktiven vom C-Compiler automatisch eingebunden.	<b>hello.h</b> streichen

Aufgrund seiner Länge haben wir das komplette Programm auf der nächsten Seite abgedruckt.

### Das Hauptprogramm: test.c

```
1 #include <stdio.h>
2 #include "hello.h"
3
4 int main( int argc, char **argv )
5 {
6     struct birthdate birth = hello( "frida" );
7     printf( "my birthday is: %d. %d. %d\n",
8           birth.day, birth.month, birth.year );
9 }
```

### Die Header-Datei: hello.h

```
1 struct birthdate { int day; int month; int year; };
2
3 struct birthdate hello( char *name );
```

### Das Modul: hello.c

```
1 #include <stdio.h>
2 #include "hello.h"
3
4 struct birthdate hello( char *name )
5 {
6     struct birthdate birth;
7     printf( "hello %s, my darling!\n", name );
8     birth.day = 30; birth.month = 2; birth.year = 1991;
9     return birth;
10 }
```

**Kommando zum Übersetzen:** gcc -o test test.c hello.c

## Teil IV: Anwendungen

---

Ziel dieses Übungspaketes ist es, ein kleines Programm zu entwickeln, das auf zwei Dateien verteilt ist und dennoch zu einem lauffähigen Programm zusammengebunden wird. Um keine weiteren Probleme zu erzeugen, verwenden wir das Programm zum Zählen von Vokalen aus Übungspaket 24.

### Vorbemerkung

Bevor wir mit der Arbeit loslegen, wiederholen wir hier als Lehrkörper nochmals drei wesentliche Punkte aus dem Skript:

1. Dem Compiler ist es es völlig egal, wo sich welche Funktionen befinden. Er muss aber wissen, in welchen Dateien sie kodiert sind, damit er sie übersetzen und am Ende zu einem Programm zusammenbinden kann.
2. Wenn der Compiler mehrere Dateien übersetzt, egal ob mittels eines oder mehrerer Kommandos, dann übersetzt er die angegebenen Dateien *nacheinander*. Nach dem Übersetzen jeder einzelnen Datei schreibt er das Ergebnis in eine Ergebnis-Datei (meist eine *.o*-Datei) und vergisst sofort alles! Das bedeutet folgendes: Der Compiler merkt sich *nicht* irgendwelche Definitionen und/oder Funktionsköpfe; wir müssen selbst dafür sorgen, dass er alles zur richtigen Zeit mitbekommt.
3. Die Verwendung von *.h*-Dateien ist nicht notwendig aber hilft uns und dem Compiler. In einer *.h*-Datei stehen üblicherweise Konstantendefinitionen (beispielsweise `#define SIZE 10`) und Funktions*deklarationen* (beispielsweise `int my_ia_prt(FILE *fp, int *a, int size );`), die auch Prototypen genannt werden. Durch diese Deklarationen kann der Compiler Inkonsistenzen feststellen und uns diese mitteilen, wodurch die Programmentwicklung für uns stark vereinfacht wird.

## Aufgabe 1: Getrenntes Übersetzen am Beispiel

### 1. Aufgabenstellung

Wir wollen wieder die Anzahl Vokale zählen, die sich in einer Zeichenkette befinden. Damit wir nicht von vorne anfangen müssen, greifen wir auf die Programme aus den Übungspaketen 24 (Anwendungsteil, Aufgaben 2) und 25 zurück. Aber diesmal packen wir die damals entwickelte Funktion `int vokale( char * str )` in eine *gesonderte* Datei. Beide Dateien wollen wir getrennt übersetzen und zu einem Programm zusammenbinden. Das Testen erledigen wir mittels des `argc/argv`-Mechanismus.

### 2. Pflichtenheft

Aufgabe	: Trennen von Funktion (Funktionalität) und Hauptprogramm in mehrere Dateien
Eingabe	: keine; die Testdaten werden über <code>argc/argv</code> übergeben
Ausgabe	: Zahl der Vokale je Zeichenkette
Sonderfälle	: keine

### 3. Entwurf

Da wir diesmal die „Programmzeilen“ trennen wollen, benötigen wir ein paar Überlegungen zum Thema Entwurf, wobei wir ein wenig helfen wollen ;-)

Wir wollen also das Hauptprogramm (`main`) vom Zählen der Vokale (`vokale()`) trennen. Ferner wollen wir sicherstellen, dass der Compiler überall die selbe Vorstellung bezüglich Funktionstyp und Parameter (Signatur) von der Funktion `vokale()` hat.

Wie viele Dateien benötigen wir?

Drei Dateien

Definiere die benötigten Dateinamen:

`main.c`, `vokale.c` und `vokale.h`

### 4. Implementierung

Hier brauchen wir uns keine weiteren Gedanken machen, da wir bereits alle notwendigen Programmzeilen in den Übungspaketen 24 und 25 entwickelt haben.

### 5. Kodierung

Zuerst die Schnittstelle `vokale.h`:

```

1  /*
2  *      datei: vokale.h
3  *      das hier ist die schnittstelle fuer vokale.c
4  */
5
6  int vokale( char * str );    // that's it! simple as this
```

Da wir diese Datei jetzt überall mittels `#include "vokale.h"` einbinden, kann der Compiler die Konsistenz über die Dateigrenzen hinweg sicherstellen.

Nun die Implementierung der Funktion vokale():

```
1  /*
2  *      datei: vokale.c
3  *      das ist die implementierung der funktion vokale()
4  */
5
6  #include "vokale.h"
7
8  int vokale( char * str )
9  {
10     int cnt;
11     if ( str != 0 ) // if (str) would be fine as well
12     {
13         for( cnt = 0; *str; str++ )
14             if ( *str == 'a' || *str == 'A' ||
15                  *str == 'e' || *str == 'E' ||
16                  *str == 'i' || *str == 'I' ||
17                  *str == 'o' || *str == 'O' ||
18                  *str == 'u' || *str == 'U' )
19                 cnt++;
20     }
21     else cnt = -1;
22     return cnt;
23 }
```

Und schließlich das Hauptprogramm main():

```
1  /*
2  *      datei: main.c
3  *      hier befindet sich das hauptprogramm
4  */
5
6  #include <stdio.h>
7  #include "vokale.h" // nicht vokale.c!!!
8
9  int main( int argc, char **argv )
10 {
11     int i;
12     for( i = 1; i < argc; i++ )
13         printf( "str='%s' vowels=%d\n",
14                 argv[ i ], vokale( argv[ i ] ) );
15 }
```

Das war's auch schon :-)

# Übungspaket 29

## Dynamische Speicherverwaltung: malloc() und free()

---

### Übungsziele:

In diesem Übungspaket üben wir das dynamische Allokieren

1. und Freigeben von Speicherbereichen
2. von Zeichenketten
3. beliebiger Arrays

### Skript:

Kapitel: 69

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Endlich wird es spannend :-) Einige von euch haben schon bemerkt, dass man bei den Arrays immer schon vorher wissen muss, wie viele Elemente man benötigt. Manchmal weiß man das aber nicht. In diesem Übungspaket üben wir, Speicher dynamisch zu allozieren und auch wieder freizugeben. Wenn man sich daran erst einmal gewöhnt hat, ist es ganz praktisch. Aber das ist ja immer so ...

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Motivation

Auf den ersten Blick mag die dynamische Speicherverwaltung überflüssig wie ein Kropf zu sein, denn wir können ja alle Variablen, insbesondere Arrays, schon zur Übersetzungszeit festlegen. Erste Antwort: sie ist natürlich nicht überflüssig, würden wir sie sonst behandeln ;-)? Überlege und beschreibe in eigenen Worten, in welchen Fällen eine dynamische Speicherverwaltung notwendig bzw. nützlich ist.

Die Antwort steht eigentlich schon in der Fragestellung: „denn wir können ja alle Variablen insbesondere Arrays schon zur Übersetzungszeit festlegen.“ Das können wir in vielen Fällen nämlich genau nicht. Beispielsweise wissen wir nicht, wie viele Zahlen wir im Rahmen des Programms verarbeiten müssen. Oder wir wissen nicht, wie lang die zu verarbeitenden Zeichenketten sind. Das heißt eben auch, dass wir die Größe eines Arrays eben *nicht* immer von vornherein wissen und unsere Arrays beim Schreiben des Programms definieren können. Entsprechend müssen wir uns damit abfinden, dass die dynamische Speicherverwaltung insbesondere bei Arrays auch bei uns zur Anwendung kommt.

## Aufgabe 2: Fragen zu den Grundlagen

Für die dynamische Speicherverwaltung verwenden wir im Wesentlichen zwei Funktionen. Wie heißen diese Funktionen und welche Signaturen haben sie?

Dynamische Speicherallokation: `void * malloc( int size )`

Dynamische Speicherfreigabe: `void free( void * ptr )`

Welche Datei muss man für diese Funktionen einbinden? `#include <stdlib.h>`

In welchem Segment liegt der neue Speicher? `Auf dem Heap`

Wie/in welcher Form wird der Speicher zurückgegeben? `Als Zeiger auf das erste Byte`

Welchen Wert hat der Zeiger im Fehlerfalle? `Es ist ein Null-Zeiger`

## Aufgabe 3: Ein einfaches Beispiel

Im Folgenden betrachten wir eine kleine, nutzlose Beispielanwendung. Diese Anwendung fragt den Benutzer nach der Zahl der Parameter, die er benötigt. Anschließend alloziert das Programm ein entsprechendes Array und, sofern dies erfolgreich war, initialisiert es die Elemente mit den Werten 0, 2, 4, ...

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



```

3
4 int main( int argc, char **argv )
5     {
6         int i, size, *p;
7         printf( "wie viele int's benoetigst du? " );
8         scanf( "%d", & size );
9         if ( size > 0 )
10        {
11            p = malloc( size * sizeof( int ) );
12            for( i = 0; i < size; i++ )
13                p[ i ] = i * 2;
14            printf( "Kontrollausgabe:" );
15            for( i = 0; i < size; i++ )
16                printf( " %d", p[ i ] );
17            printf( "\n" );
18        }
19        else printf( "sorry, da kann ich nicht helfen\n" );
20    }

```

Wir wollen nun von euch, dass ihr ein Speicherbildchen malt, in denen die Variablen `i`, `size` und `p` enthalten sind. Ebenso wollen wir den neu allozierten Speicherbereich sowie die in den Zeilen 12 und 13 generierten Initialwerte sehen. Für das Beispiel nehmen wir an, dass der Nutzer die Zahl 3 eingegeben hat. Mit anderen Worten: Wie sieht der Arbeitsspeicher unmittelbar vor Ausführung der Programmzeile 14 aus?

Segment: Stack

Adresse	Variable	Wert
0xFE28	int * p : 0xF040	
0xFE24	int size :	3
0xFE20	int i :	3

Segment: Heap

Adresse	Variable	Wert
0xF048	int [ 2 ] :	4
0xF044	int [ 1 ] :	2
0xF040	int [ 0 ] :	0

## Teil II: Quiz

---

### Aufgabe 1: Speicherallokation

In der folgenden Tabelle stehen in den drei Spalten der Reihe nach das auszuführende Kommando, die angelegte Datenstruktur und die Zahl der Bytes, die diese im Arbeitsspeicher belegt. Dabei gehen wir davon aus, dass ein `char` 1 Byte, ein `int` sowie ein Zeiger jeweils 4 Bytes und ein `double` 8 Bytes im Arbeitsspeicher belegt. Vervollständige die fehlenden Spalten. Im Sinne eines etwas erhöhten Anspruchs haben wir noch die folgende Struktur:

```
1 struct bla { int bla_bla; double blub_blub; };
```

malloc()-Aufruf	Datenstruktur	Größe
<code>malloc(3 * sizeof( int ))</code>	Array mit 3 ints	12
<code>malloc(5 * sizeof( double ))</code>	Array mit 5 double	40
<code>malloc( 12 )</code>	Vermutlich ein Array mit 12 char	12
<code>malloc(2 * sizeof(struct bla))</code>	Array mit 2 struct bla Elementen	24
<code>malloc(2 * sizeof(struct bla *))</code>	Array mit 2 Zeigern auf auf Elemente vom Typ struct bla	8
<code>malloc(5 * sizeof(char *))</code>	Array mit 5 char-Zeigern	20
<code>malloc(13)</code>	ein char-Array	13
<code>malloc(6 * sizeof(double **))</code>	ein Array mit 6 Zeigern auf double-Zeiger	24

### Aufgabe 2: Speicherfreigabe

Erkläre mit eigenen Worten, was folgende Funktion macht:

```
1 void tst_and_free( void *p )
2     {
3         if ( p != 0 )
4             free( p );
5     }
```

Diese Funktion gibt Speicher wieder frei, da sie offensichtlich die Funktion `free()` aufruft. Allerdings testet sie vorab, ob es sich um einen „regulären“ Zeigerwert handelt. Mit anderen Worten: Diese Funktion vermeidet es, einen Null-Zeiger versehentlich freizugeben, da dies unter Umständen innerhalb von `free()` zu einem Programmabsturz führen könnte.

## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler bei malloc() und free()

Der berühmt-berüchtigte Manager DR. M. AL LOC fand die Idee mit den dynamischen Datenstrukturen so spannend, dass er es gleich einmal ausprobierte. Er wollte eine Funktion haben, die ihm ein Array mit `size doubles` erzeugt, wobei die einzelnen Elemente mit den Werten 0.0, 1.0, ... initialisiert werden sollen. Am Ende soll der Speicher wieder freigegeben werden. Finde und korrigiere die Fehler:

```
1  #include <stdio.h>                                // for printf() und scanf()
2  #include <stdlib.h>                                // for malloc() and free()
3
4  double *mk_darray( int n )
5  {
6      int i; double d_a;                            // compact due to space
7      if ((d_a = malloc( n )) != 0 )
8          for( i = 0; i < n; i++ )
9              d_a[ i ] = i;
10     return d_a;
11 }
12
13 int main( int argc, char **argv )
14 {
15     int i, size; double *dp;                        // compact due to space
16     printf("wie viele parameter? "); scanf("%d", & size);
17     dp = mk_darray( size );
18     printf( "hier das ergebnis:" );
19     for( i = 0; i < size; i++ )
20         printf( " %5.2f", dp[ i ] );
21     printf( "\n" );
22     free( dp );                                     // free the memory
23     free( dp );                                     // just to be absolutely sure
24     return 0;
25 }
```

---

Zeile	Fehler	Erläuterung	Korrektur
6	* fehlt	Die Variable <code>d_a</code> muss natürlich als Zeiger definiert werden, da <code>malloc()</code> einen Zeiger auf einen neuen Speicherbereich zurückgibt.	<code>*d_a</code>

---

Zeile	Fehler	Erläuterung	Korrektur
7	<code>sizeof()</code> fehlt	Ein ganz typischer Fehler: In diesem Falle würde <code>malloc()</code> nur <code>n</code> Bytes bereitstellen. Wir benötigen aber <code>n * sizeof(double)</code> Bytes, was wir beim Aufruf von <code>malloc()</code> <i>unbedingt</i> berücksichtigen müssen.	<code>sizeof()</code>
18 ff.	Test auf Null-Zeiger fehlt	In Zeile 17 wird ein neuer Speicherplatz dynamisch alloziiert. Aber dies kann schief gehen, was durch einen Null-Zeiger angezeigt wird. Da man Null-Zeiger nicht dereferenzieren (auf die Inhalte zugreifen) darf, muss der nachfolgende Programmtext geklammert werden.	<code>if ( ) ...</code>
22/23	ein <code>free()</code> zu viel	Einen zuvor alloziierten Speicherplatz darf man <i>nur einmal</i> freigeben. Das Ergebnis durch den zweiten Aufruf ist unbestimmt und <i>kann</i> hier oder später zu einem Programmabsturz führen.	ein <code>free()</code> streichen

#### Programm mit Korrekturen:

```

1  #include <stdio.h>                                // for printf() und scanf()
2  #include <stdlib.h>                                // for malloc() and free()
3
4  double *mk_darray( int n )
5  {
6      int i; double *d_a;                            // compact due to space
7      if ((d_a = malloc(n * sizeof( double ))) != 0 )
8          for( i = 0; i < n; i++ )
9              d_a[ i ] = i;
10     return d_a;
11 }
12
13 int main( int argc, char **argv )
14 {
15     int i, size; double *dp;                        // compact due to space
16     printf("wie viele parameter? "); scanf("%d", & size);
17     if (dp = mk_darray( size ))
18     {
19         printf( "hier das ergebnis:" );
20         for( i = 0; i < size; i++ )
21             printf( " %5.2f", dp[ i ] );
22         printf( "\n" );
23         free( dp );
24     }
25     return 0;
26 }
```

# Teil IV: Anwendungen

---

## Aufgabe 1: Aneinanderhängen zweier Zeichenketten

### 1. Aufgabenstellung

Entwickle eine Funktion, die als Eingabeparameter zwei Zeiger auf je eine Zeichenkette erhält, diese beiden Zeichenketten zu einer *neuen* Zeichenkette zusammenfügt und das Ergebnis als Zeiger auf diese zurückgibt.

**Beispiel:** Aufruf: `my_strcat( "Prof.", "Weiss-Nicht" )`  
Ergebnis: "Prof. Weiss-Nicht"

Diese Funktion kann wieder einfach mittels des `argc/argv`-Mechanismus getestet werden.

### 2. Vorüberlegungen

Was können wir aus der Aufgabenstellung direkt ablesen?

1. Wir müssen zwei Zeichenketten zusammenfügen, deren Längen wir zur Übersetzungszeit noch nicht wissen.
2. Daraus folgt, dass wir das Ergebnis nicht auf dem Stack ablegen können, sondern uns den benötigten Speicherplatz dynamisch vom Heap besorgen müssen.
3. Die neue Zeichenkette muss so lang sein, dass die beiden alten hinein passen, dann noch ein Leerzeichen und am Ende ein Null-Byte.

### 3. Pflichtenheft

Aufgabe	: Funktion zum Verknüpfen zweier Zeichenketten. Das Ergebnis wird auf dem Heap abgelegt.
Parameter	: Zwei Eingabeparameter <code>p1</code> und <code>p2</code> vom Typ <code>char *</code>
Rückgabewert	: Einen Zeiger auf eine neue Zeichenkette vom Typ <code>char *</code>

### 4. Testdaten

Ähnlich wie in der Aufgabenstellung beschrieben. Bei Verwendung des `argc/argv`-Mechanismus nehmen wir `argv[ 1 ]` als ersten Namen und alle weiteren Parameter als zweiten Namen.

**Beispiel:** Eingabe: `./my-strcat Dr. Peter Maria`  
Ausgabe: `Dr. Peter`  
`Dr. Maria`

## 5. Implementierung

Da wir den `argc/argv`-Mechanismus schon eingehend geübt haben, können wir uns ganz auf die neu zu entwickelnde Funktion konzentrieren. Die wesentlichen Punkte standen schon in den „Vorüberlegungen“. Wir erhalten zwei Zeichenketten. Die Länge der neuen Zeichenkette muss die Summe der beiden eingehenden Zeichenketten sein, plus eins für das Leerzeichen und eins für das abschließende Null-Byte. Das lässt sich wie folgt ausdrücken:

Funktion zum Aneinanderhängen von Zeichenketten

```
Parameter: Pointer to Char: p1, p2
Variablen: Integer: len1, len2      // zwei laengenwerte
           Pointer to char: res      // der rueckgabewert

setze len1 = strlen( p1 )
setze len2 = strlen( p2 )
setze res = malloc(len1 + len2 + 2)
wenn res ≠ 0
dann kopiere p1 an die Stelle res[ 0 ]
    kopiere ein Leerzeichen an die Stelle res[ len1 ]
    kopiere p2 an die Stelle res[len1 + 1]
    kopiere ein Null-Byte an die Stelle res[len1 + 1 + len2]

return res
```

## 6. Kodierung

Unsere Funktion `my_strcat()`:

```
1  #include <stdio.h>           // for I/O
2  #include <string.h>          // for strlen() und strcpy()
3  #include <stdlib.h>          // for malloc()
4
5  char *my_strcat( char *p1, char *p2 )
6  {
7      char *res;
8      int len1 = strlen( p1 ), len2 = strlen( p2 );
9      if (res = malloc( len1 + len2 + 2 ))
10     {
11         strcpy( res, p1 );
12         res[ len1 ] = ' ';
13         strcpy( res + len1 + 1, p2 );
14         res[len1 + len2 + 1] = '\0';
15     }
16     return res;
17 }
```

Das ist unser Hauptprogramm zum Testen:

```
18 int main( int argc, char **argv )
19     {
20         int i;
21         char *combi;
22         for( i = 2; i < argc; i++ )
23             if (combi = my_strcat( argv[ 1 ], argv[ i ] ))
24                 {
25                     printf( "my_strcat: '%s'\n", combi );
26                     free( combi );      // otherwise: memory leak!
27                 }
28     }
```

## Aufgabe 2: Verallgemeinertes `strcat()`

### 1. Aufgabenstellung

In dieser Aufgabe soll die Funktion `my_strcat()` der vorherigen Aufgabe erweitert werden. Statt nur zwei Zeichenketten zusammenzufügen, soll die Funktion diesmal in der Lage sein, „beliebig“ viele Zeichenketten zu verarbeiten. Die einzelnen Zeichenketten sollen wieder mittels eines Leerzeichens voneinander getrennt werden. Natürlich muss die gesamte Zeichenkette wieder mit einem Null-Byte abgeschlossen werden. Innerhalb dieser Übungsaufgabe nennen wir diese Funktion `my_nstrcat()`

**Beispiel:** Bei Übergabe der drei Parameter `"Frau"`, `"Peter"` und `"Otti"` soll das Ergebnis `"Frau Peter Otti"` lauten.

Zum Testen bedienen wir uns wieder des `argc/argv`-Mechanismus.

### 2. Vorüberlegungen

Auch hier ist es wieder sinnvoll, ein paar Vorüberlegungen anzustellen. Doch diesmal werden wir diese als Fragen formulieren, die ihr beantworten müsst.

Erfordert die Aufgabenstellung eine dynamische Bearbeitung mittels `malloc()`?

Ja, unbedingt

Falls ja, warum?

Anzahl und Längen der Zeichenketten sind unbekannt

Falls nein, warum nicht?

Die Aufgabe wäre in einem anderen Übungspaket ;-)

Wie übergeben wir die Zeichenketten?

In einem Array verpackt

Welchen Datentyp hat dieser Parameter?

Array of Pointer to Char: `char **`

Benötigen wir noch einen zweiten Parameter?

Ja

Falls ja, welchen Typ hat dieser Parameter?

`int size`

Welchen Typ hat die Funktion?

`char *`

Wie lautet die vollständige Funktionsdeklaration?

`char *my_nstrcat( char **str_arr, int size )`

Wie bestimmen wir die Länge einer Zeichenkette?

`len = strlen( zeichenkette )`

Wie lang ist die Ergebnis-Zeichenkette?

$\sum_{i=0}^{\text{size}-1} \text{strlen}( \text{zeichenkette}[ i ] )$



### 3. Pflichtenheft

Aufgabe	: Funktion zum Verknüpfen von $n$ Zeichenketten. Das Ergebnis wird auf dem Heap abgelegt.
Parameter	: Zwei Eingabeparameter: <code>char **str_arr</code> und <code>int size</code>
Rückgabewert	: Einen Zeiger auf eine neue Zeichenkette vom Typ <code>char *</code>

### 4. Testdaten

Wir bedienen uns wieder des `argc/argv`-Mechanismus:

**Beispiel:** Kommando: `./my-strcat Dr. Peter Maria`  
Ausgabe: `./my-strcat Dr. Peter Maria` als *eine* Zeichenkette

### 5. Implementierung

Wir können uns gleich auf die Funktion `my_nstrcat()` konzentrieren. Wir müssen der Reihe nach die folgenden Schritte durchführen: Erstens, Berechnung der Gesamtlänge der neuen Zeichenkette. Diese ist die Summe der Einzellängen plus die Anzahl der Zeichenketten ( $n-1$  Leerzeichen als Trenner sowie ein Null-Byte am Ende). Zweitens, Allokation der neuen Zeichenkette. Und drittens, Hineinkopieren der einzelnen Zeichenketten mit jeweils einem anschließenden Leerzeichen. Das letzte Leerzeichen wird durch ein Null-Byte ersetzt. Bei jedem Kopiervorgang müssen wir in der neuen Zeichenkette um so viele Positionen weiterrücken, wie die zuletzt kopierte Zeichenkette lang war. Das lässt sich wie folgt ausdrücken:

Funktion zum Aneinanderhängen von Zeichenketten

```
Parameter: Array of Pointer to Char: str_arr
           Integer: size
Variablen: Integer: i, len, pos
setze len = 0
für i = 0 bis size - 1 Schrittweite 1
wiederhole setze len = strlen( str_arr[ i ] )
setze res = malloc( len + size )
wenn res  $\neq$  0
dann setze pos = 0
    für i = 0 bis size - 1 Schrittweite 1
    wiederhole kopiere str_arr[ i ] an die Stelle res[ pos ]
        setze pos = pos + strlen( str_arr[ i ] )
        setze res[ pos ] = Leerzeichen
        setze pos = pos + 1
    setze res[ pos ] = Null-Byte
return res
```

## 6. Kodierung

```
1  #include <stdio.h>           // for I/O
2  #include <string.h>          // for strlen() und strcpy()
3  #include <stdlib.h>          // for malloc()
4
5  char *my_nstrcat( char **str_arr, int size )
6  {
7      char *res;
8      int i, len, pos;
9      for( i = 0, len = size; i < size; i++ )
10         len += strlen( str_arr[ i ] );
11     if (res = malloc( len ))
12     {
13         for( i = pos = 0; i < size; i++ )
14         {
15             strcpy( res + pos, str_arr[ i ] );
16             pos += strlen( str_arr[ i ] );
17             res[ pos ] = ' ';
18             pos++;           // advance to the next position
19         }
20         res[ pos - 1 ] = '\\0';    // replace last blank
21     }
22     return res;
23 }
24
25 int main( int argc, char **argv )
26 {
27     char *combi;
28     if (combi = my_nstrcat( argv, argc ))
29     {
30         printf( "my_nstrcat: '%s'\\n", combi );
31         free( combi );           // just to get used to it
32     }
33 }
```

**Anmerkung:** Wir hätten auch auf eine der beiden Variablen `len` oder `pos` verzichten können. Nur fanden wir es so verständlicher. Und die Einsparung einer `int`-Variablen fällt bei den Zeichenketten nicht ins Gewicht.

# Übungspaket 30

## Kopieren von Dateien

---

### Übungsziele:

1. Öffnen und Schließen von Dateien
2. Einfaches Lesen und Schreiben
3. Behandlung der „EOF-Marke“
4. Kopieren ganzer Dateien

### Skript:

Kapitel: 59 bis 67 und insbesondere Übungspaket 25

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Wie schon im Skript geschrieben, ist das Arbeiten mit Dateien mühsam und immer wieder voller Überraschungen, selbst für fortgeschrittene Programmierer. In so einer Situation hilft nur eines: intensiv Üben. In diesem Übungspaket schauen wir uns die wichtigsten Funktionen aus dem Paket `stdio.h` an, üben den richtigen Umgang mit den Ein- und Ausgabeformatierungen und entwickeln ein kleines Programm, das ähnlich dem Unix/Linux Programm `cat` Dateien kopiert.

# Teil I: Stoffwiederholung

---

Vorab nochmals eine wesentliche Bemerkung aus dem Skript: Die Dateischnittstelle ist eine *sehr* schwere Sache! Wenn man denkt, man hat's verstanden, kommt meist' doch wieder etwas neues hinzu. Während des Lernens kommt man sich oft vor, „wie ein Schwein, dass in's Uhrwerk schaut.“ Nichtsdestotrotz versuchen wir uns mittels dieses Übungspaketes der Dateischnittstelle ein wenig auf freundschaftlicher Basis zu nähern. Also: *Good Luck!*

## Aufgabe 1: Die Dateischnittstelle FILE/\*FILE

Zuerst einmal beschäftigen wir uns mit den relevanten Dateien und Datenstrukturen.

1. Welche Datei muss bzw. sollte man einbinden, wenn man etwas einliest oder ausgibt?

`#include <stdio.h>`, denn hier befinden sich alle wesentlichen Definitionen

2. Nun zur eigentlichen FILE-Datenstruktur: Diese haben wir *außerordentlich vereinfacht* in Skriptkapitel 63 erläutert. Ziel war es, einen ersten Eindruck hiervon zu vermitteln. Für das weitere Vorgehen ist es sinnvoll, sich die Datei `stdio.h` einmal genauer anzuschauen. Sie befindet sich unter Linux im Verzeichnis `/usr/include` bzw. unter Windows im include-Verzeichnis des installierten Compiler wie beispielsweise `C:\MinGW\include`. Diese Datei ist sehr lang. Trotzdem einfach mal hineinschauen, die Betreuer werden gerne weiterhelfen. Sucht diese Datei und benennt die wesentlichen Komponenten der FILE-Datenstruktur. Hinweis: Linux-Nutzer schauen sich auch die Datei `libio.h` an, die sich ebenfalls unter `/usr/include` befindet.

Auf vielen Systemen sieht die FILE-Datenstruktur wie folgt aus:

```
typedef struct _iobuf {
    char * _ptr;      int    _cnt;      char * _base;
    int    _flag;     int    _file;     int    _charbuf;
    int    _bufsiz;   char * _tmpfname;
} FILE;
```

Das sieht für Anfänger und leicht Fortgeschrittene recht kryptisch aus. Aber jeder kann ja mal raten, wofür die einzelnen Einträge zuständig sind.

Das Schöne an der Sache ist, dass wir gar nicht direkt mit den einzelnen Komponenten arbeiten; dies überlassen wir den einzelnen abstrakten Funktionen.

3. Schauen wir uns noch kurz die folgenden Fragen zur FILE-Datenstruktur an:

Arbeiten wir direkt mit der FILE-Datenstruktur?

Wer arbeitet mit der FILE-Datenstruktur?

Welche Funktionen sind dir bekannt?

## Aufgabe 2: Detailfragen zu den Dateizugriffen

An dieser Stelle sei angemerkt, dass der Datentyp `FILE *` (etwas irreführend) vielfach auch *File Pointer* genannt wird.

Wie heißt der File Pointer für die Standardeingabe?	<code>stdin</code>
Wie heißt der File Pointer für die Fehlerausgabe?	<code>stderr</code>
Wie heißt der File Pointer für die Standardausgabe?	<code>stdout</code>
Mit welcher Funktion öffnet man Dateien?	<code>FILE *fopen(char *fname, char *mode)</code>
Welchen Rückgabewert hat diese Funktion?	Ein neuer File Pointer vom Typ <code>FILE *</code>
Wie erkennt man einen Fehler beim Öffnen?	Der Rückgabewert ist ein Null-Zeiger
Welcher <code>mode</code> öffnet zum Lesen?	<code>"r"</code>
Welcher <code>mode</code> öffnet zum Schreiben?	<code>"w"</code>
Beispiel für das Öffnen zum Lesen:	<code>FILE *fp = fopen("bsp-rd.txt", "r")</code>
Beispiel für das Öffnen zum Schreiben:	<code>FILE *fp = fopen("bsp-wt.txt", "w")</code>

Wie unterscheiden sich die Funktionen `scanf()` und `fscanf()` beziehungsweise `printf()` und `fprintf()` voneinander?

Die Funktionen `scanf()` und `printf()` sollten bereits hinlänglich bekannt sein. `scanf()` liest Eingaben von der Tastatur und `printf()` gibt auf den Bildschirm aus. Oder? Nun, prinzipiell jein. Korrekt ist die Aussage: `scanf()` liest von der Standardeingabe und `printf()` gibt auf die Standardausgabe aus. Im Regelfall entsprechen aber die Standardeingabe der Tastatur und die Standardausgabe dem Bildschirm. Die Funktionen `fscanf()` und `fprintf()` ähneln den obigen Funktionen mit dem Unterschied, dass bei diesen beiden Funktionen die Quelle, bzw. das Ziel, explizit angegeben werden müssen. Als Angabe für Quelle und Ziel verwendet man einen File Pointer (eine Variable vom Typ `FILE *`). Insofern sind `printf( "Hallo Welt\n" )` und `fprintf(stdout, "Hallo Welt\n" )` identisch, ebenso `scanf( "%d", & i )` und `fscanf( stdin, "%d", & i )`.

## Teil II: Quiz

---

### Aufgabe 1: Eingabe gemischter Datentypen

Auch wenn man schon eine ganze Weile dabei ist, so sind viele doch immer wieder davon überrascht, wie in der Programmiersprache C die Eingabe verarbeitet wird. Zu diesem Thema haben wir folgendes Quiz entwickelt. Zunächst aber mal einige Anmerkungen:

- Wir haben ein C-Programm, das die Funktion `scanf()` mit verschiedenen Formaten aufruft. Jeder Aufruf von `scanf()` liest immer nur *einen* Parameter ein. Anschließend wird sofort der Rückgabewert der Funktion `scanf()` sowie der eingelesene Wert ausgegeben.
- Die Routinen für die eigentliche Ein-/Ausgabe haben wir ausgelagert und am Ende dieses Übungsteils abgedruckt. Alle Funktionen haben einen Namen, der immer mit 'do\_' anfängt und mit dem entsprechenden Datentyp aufhört. Beispiel: `do_int()` verarbeitet einen Wert vom Typ `int`.
- Das Programm liest und schreibt immer auf die Standardein- bzw. -ausgabe.
- Im Hauptprogramm werden die entsprechenden Routinen aufgerufen und mit der entsprechenden Formatangabe versorgt. Mit anderen Worten: Aus dem Funktionsnamen und der Formatangabe lässt sich die Ausgabe direkt ableiten.
- Eure Aufgabe ist es nun, sich die Programmeingabe anzuschauen und vorherzusagen, was die entsprechenden Ausgaben sein werden. Dazu ist es durchaus ratsam, sich nochmals eben die Dokumentation der Formatierungen `%d`, `%c` und `%e` anzuschauen.
- Die Eingabedatei bzw. die Eingabedaten sehen wie folgt aus:

Die Eingabedatei:

```
1 12
2
3 56
4 12345 987AB 123
5 abc
```

Die Daten zeichenweise dargestellt:

1	2	_	\n	\n	_	_	5
6	\n	1	2	3	4	5	_
9	8	7	A	B	_	1	2
3	\n	a	b	c	EOF		

- Zur besseren Veranschaulichung haben wir die Funktion `debug_char()` geschrieben, die die Ausgaben etwas aufbereitet: Sonderzeichen wie Tabulator und Zeilenwechsel werden zusätzlich als C-Escape-Sequenzen dargestellt.
- Zur Erinnerung: Die Standardein- und -ausgabe kann mittels `<` und `>` umgeleitet werden, ohne dass das Programm davon etwas bemerkt. Beispiel: `a.exe < test.txt`

## Das Programm:

```

1 int main( int argc, char **argv )
2 {
3     do_int( "%d" );
4     do_int( "%d" );
5     do_char( "%c" );
6     do_char( "%c" );
7     do_int( "%3d" );
8     do_char( "%c" );
9     do_int( "%d" );
10    do_char( "%c" );
11    do_int( "%d" );
12    do_char( "%c" );
13    do_int( "%d" );
14    do_nchar( "%4c", 4 );
15    do_int( "%d" );
16 }

```

## Die resultierenden Ausgaben:

```

1 int      : res= 1 i=12
2 int      : res= 1 i=56
3 char     : res= 1 c=\n
4 char     : res= 1 c=1
5 int      : res= 1 i=234
6 char     : res= 1 c=5
7 int      : res= 1 i=987
8 char     : res= 1 c=A
9 int      : res= 0 i=---
10 char    : res= 1 c=B
11 int     : res= 1 i=123
12 char    : res= 1 c=\nabc
13 int     : res=-1 i=---

```

Die Funktionen `do_int()`, `do_char()` und `do_nchar()` befinden sich auf der nächsten Seite.

## Die Eingabedatei:

zeichenweise dargestellt

1	2	␣	\n	\n	␣	␣	5	6	\n	1	2	3	4	5
␣	9	8	7	A	B	␣	1	2	3	\n	a	b	c	EOF

## Die auszufüllende Tabelle: Was wird wie abgearbeitet?

Zeile	Datentyp	res	Wert	Anmerkungen
3	int	1	12	ein <code>int</code> eingelesen, wie zu erwarten war
4	int	1	56	Leerzeichen, 2 Zeilenwechsel, 2 Leerzeichen überlesen, <code>int</code> eingelesen, alles normal
5	char	1	'\n'	Der Zeilenwechsel ist also das nächste Zeichen
6	char	1	'1'	Und jetzt schon die Ziffer '1'
7	int	1	234	Endlich wieder ein <code>int</code> aber nur drei Ziffern :-(
8	char	1	'5'	Hier jetzt die Ziffer '5' als Zeichen
9	int	1	987	Wieder ganz normal ein <code>int</code> gelesen
10	char	1	A	Ok, das Zeichen direkt hinter der Zahl
11	int	0	---	Nanu, keine Zahl gefunden? 'B' ist ja auch keine Zahl
12	char	1	'B'	Aha, wir stehen noch beim 'B', ist auch keine Zahl
13	int	1	123	Puhhh, wieder alles normal
14	char	1	\nabc	Interessant, mit <code>%4c</code> werden direkt vier Zeichen eingelesen und in einem Array abgelegt
15	int	-1	---	Das Dateiende bemerken wir also mit <code>res == EOF</code>

## Die einzelnen, sehr komprimierten Verarbeitungsroutinen:

```
1  #include <stdio.h>
2
3  void do_int( char * fmt )
4  {
5      int i, res = scanf( fmt, & i );
6      printf( "int      : res=%2d ", res );
7      if ( res == 1 )
8          printf( "i=%d\n", i );
9      else printf( "i=---\n" );
10 }
11
12 void debug_char( char c, int ok )          // prettyprint char
13 {
14     if ( ! ok )
15         printf( "--- " );
16     else if ( c == '\t' )
17         printf( "\\t" );
18     else if ( c == '\n' )
19         printf( "\\n" );
20     else printf( "%c", c );
21 }
22
23 void do_char( char * fmt )
24 {
25     char c;
26     int res = scanf( fmt, & c );
27     printf( "char    : res=%2d c=", res );
28     debug_char( c, res == 1 );
29     printf( "\n" );
30 }
31
32 void do_nchar( char * fmt, int len )
33 {
34     int i, res;
35     char c[ 10 ];
36     res = scanf( fmt, c ); // or & c[ 0 ]
37     printf( "char    : res=%2d c=", res );
38     for( i = 0; i < len; i++ )
39         debug_char( c[ i ], res > 0 );
40     printf( "\n" );
41 }
```



## Teil III: Fehlersuche

---

### Aufgabe 1: Zählen von Vokalen in einer Datei

Diesmal hat sich unser Programmierer DR. V. ERROR versucht. Sein Programm soll die Anzahl der Vokale in einer Datei ermitteln. Bei der Dateiverarbeitung hat er aber nicht ganz aufgepasst. Finde und korrigiere die Fehler in folgendem Programm:

```
1  #include <stdio.h>
2
3  int cntVowels( FILE fp )
4  {
5      int c, cnt;
6      for( cnt = 0; (c = getc()) != EOF; )
7          cnt += c=='a' || c=='e' || c=='i' || c=='o'
8                || c=='A' || c=='E' || c=='I' || c=='O'
9                || c=='u' || c=='U' ;
10     return cnt;
11 }
12
13 int main( int argc, char **argv )
14 {
15     int i, cnt;
16     int fp;
17     for( i = 1; i < argc; i++ )
18         if ((fp = fopen( argv[ i ], "w" )) != 0 )
19             {
20                 cnt = cntVowels( fp );
21                 printf(stdout, "'%s': %d vokale\n", argv[i], cnt);
22                 fclose( *fp );
23             }
24         else printf( stderr,
25                     "'%s': fehler beim oeffnen\n", argv[i] );
26 }
```

Zeile	Fehler	Erläuterung	Korrektur
3	* fehlt	Hier <i>müsste</i> ein Zeiger stehen, da sonst der Aufrufer mögliche Veränderungen der FILE-Datenstruktur nicht mitbekommt.	FILE *fp
6	fp fehlt	Die Funktion <code>getc()</code> erfordert einen File Pointer, damit sie weiß, von wo sie lesen soll.	<code>getc( fp )</code>

Zeile	Fehler	Erläuterung	Korrektur
16	int statt FILE *	Funktionen wie <code>getc()</code> und <code>fprintf()</code> benötigen Zeiger vom Typ <code>FILE</code> . Es gibt auch Funktionen, die hier <code>int</code> -Parameter erwarten; diese besprechen bzw. verwenden wir aber nicht.	<code>FILE *</code>
18	"w" statt "r"	Wenn man von Dateien <i>Lesen</i> will, muss man sie mit- tels "r" öffnen.	"r"
21	stdout zu viel	Die Funktion <code>printf()</code> schreibt <i>immer</i> auf die Stan- dardausgabe und benötigt keinen File Pointer.	ohne <code>stdout</code>
22	* zu viel	Fast alle Funktionen aus <code>stdio.h</code> benötigen einen Fi- le <i>Pointer</i> , ohne ihn zu dereferenzieren.	<code>fp</code>
24	Falsche Funktion	Um etwas auf die Standardfehlerausgabe zu schrei- ben, benötigt man die Funktion <code>fprintf()</code> nebst des Parameters <code>stderr</code> .	<code>fprintf()</code>

### Programm mit Korrekturen:

```

1  #include <stdio.h>
2
3  int cntVowels( FILE *fp )
4  {
5      int c, cnt;
6      for( cnt = 0; (c = getc( fp )) != EOF; )
7          cnt += c=='a' || c=='e' || c=='i' || c=='o'
8                || c=='A' || c=='E' || c=='I' || c=='O'
9                || c=='u' || c=='U' ;
10     return cnt;
11 }
12
13 int main( int argc, char **argv )
14 {
15     int i, cnt;
16     FILE *fp;
17     for( i = 1; i < argc; i++ )
18         if ((fp = fopen( argv[ i ], "r" )) != 0 )
19             {
20                 cnt = cntVowels( fp );
21                 printf( "'%s' hat %d vokale\n", argv[i], cnt );
22                 fclose( fp );
23             }
24         else fprintf( stderr,
25                     "'%s': fehler beim oeffnen\n", argv[i] );
26     }

```

## Teil IV: Anwendungen

---

Das Ziel des Anwendungsteils ist es, die Funktion des Unix/Linux-Kommandos `cat` teilweise nachzuimplementieren. Dieses Kommando macht nichts anderes als eine oder mehrere Dateien in eine andere zu kopieren. Für den Programmieranfänger ist diese Aufgabe schon nicht mehr ganz einfach, da sie einerseits einen (relativ einfachen) inhaltlichen Teil (das Kopieren einer Datei in eine andere) hat, andererseits auch eine Einbindung in die „reale“ Computerwelt beinhaltet. Um hier klar zu sehen, gehen wir in drei Schritten vor: Zuerst diskutieren wir die generelle Aufgabenstellung, dann kümmern wir uns um die eigentliche Funktionalität und letztlich beschäftigen wir uns mit der Parameterversorgung.

### Vorüberlegungen: Design des `cat`-Kommandos

Nochmals von vorne: Das Linux `cat`-Kommando kopiert eine oder mehrere Dateien in eine neue Datei. Dabei macht es sich einige Eigenschaften des Betriebssystems, insbesondere das Umlenken von Dateien zunutze. Das bedeutet: das `cat`-Kommando schreibt seine Ausgaben immer auf die Standardausgabe, also in der Regel auf den Bildschirm. Will man die Ausgaben, was meist der Fall ist, in einer anderen Datei haben, muss man die Bildschirm-  
ausgaben einfach umlenken. Ok, hier ein paar Beispiele:

<code>cat</code>	Kopiert die Tastatureingaben auf den Bildschirm.
<code>cat datei-1</code>	Kopiert die Datei <code>datei-1</code> auf den Bildschirm.
<code>cat d-1 d-2 ... d-n</code>	Kopiert die Datei <code>d-1</code> , <code>d-2</code> ... <code>d-n</code> nacheinander auf den Bildschirm.
<code>cat &gt; out</code>	Kopiert die Tastatureingaben in die Datei <code>out</code>
<code>cat in-1 &gt; out</code>	Kopiert die Datei <code>in-1</code> in die Datei <code>out</code>
<code>cat in-1 ... in-n &gt; out</code>	Kopiert die Datei <code>in-1</code> ... <code>in-n</code> nacheinander in die Datei <code>out</code>
<code>cat -</code>	Kopiert die Tastatureingabe (die Datei <code>-</code> ) auf den Bildschirm.
<code>cat d-1 - d-2</code>	Kopiert zuerst die Datei <code>d-1</code> , dann die Tastatureingabe und schließlich die Datei <code>d-2</code> auf den Bildschirm.

Soweit, so gut. Wichtig für das Verständnis ist nun, dass das Kommando gar nicht merkt, dass es die Ausgaben nicht auf den Bildschirm sondern ggf. in eine Datei schreibt. Dieses Umlenken erledigt das Betriebssystem und ist (nahezu) für das Kommando unsichtbar. Das heißt, das Kommando `cat` schreibt prinzipiell immer und alles auf die Standardausgabe.

Was können wir aus obiger Beschreibung an Informationen für uns herausziehen?

1. `cat` schreibt immer auf die Standardausgabe.

2. `cat` liest alle Dateien nacheinander. Das heißt, dass man nur eine Funktion benötigt, die von einem File Pointer liest und auf den anderen schreibt. Sollten mehrere Dateien gelesen werden, muss man diese Funktion nur immer wieder mit einem anderen Eingabeparameter aufrufen.
3. Sollte `cat` keinen Parameter bekommen, liest es von der Standardeingabe.

Was heißt das für unser Programm? Wir brauchen eine Funktion `copyFile()`, die von einem File Pointer liest und auf einen anderen schreibt. Um möglichst flexibel zu bleiben, sehen wir zwei Parameter vor, auch wenn der zweite immer mit `stdout` belegt werden wird. Um diese (recht einfache) Funktion zu testen, rufen wir sie einfach mit der Standardein- und Standardausgabe auf.

Ferner brauchen wir dann noch ein Hauptprogramm, das je nach Parameterlage die einzelnen Dateien öffnet, die Funktion `copyFile()` entsprechend aufruft und die Dateien wieder schließt. Und genau das machen wir jetzt in den nächsten beiden Aufgaben.

## Aufgabe 1: Kopieren *einer* Datei

### 1. Aufgabenstellung

Entwickle eine Funktion `copyFile`, die Dateien zeichenweise liest und wieder ausgibt. Nach den oben bereits angestellten Vorüberlegungen sollten die folgenden Fragen einfach zu beantworten sein:

Welche Parameter benötigt die Funktion?	Eine Ein- und eine Ausgabedatei
Welchen Typ haben die beiden Parameter?	<code>FILE *</code>
Welchen Typ sollte die Funktion haben?	<i>Beispielsweise</i> <code>int</code>
Was für ein Rückgabewert ist sinnvoll?	<i>Bspw.</i> die Zahl der kopierten Zeichen

**Hinweis:** Für das Weiterarbeiten empfiehlt es sich, die Dokumentation der Funktion `fgetc()` und `scanf()` hinsichtlich des Lesens von Zeichen und des Erreichens des Dateiendes anzuschauen.

### 2. Pflichtenheft

Aufgabe	: Entwickle eine Funktion zum Kopieren einer Datei.
Parameter	: Je ein Zeiger auf die Eingabe- und Ausgabedatei, die bereits geöffnet sind.
Rückgabewert	: Die Zahl der kopierten Zeichen.
Schleifenbedingung	: Lesen bis das Dateiende (EOF) erreicht ist.

### 3. Testdaten

Als Testdateien bieten sich sowohl die Standardeingabe als auch die eigens erstellten C-Dateien an.

<b>Beispiele:</b>	Kommando	Bedeutung
	<code>mycat</code>	Lesen der Standardeingabe
	<code>mycat &lt; mycat.c</code>	Lesen der Datei <code>mycat.c</code> über die Standardeingabe
	<code>mycat &gt; out.txt</code>	Lesen der Standardeingabe, Schreiben in die Datei <code>out.txt</code>

### **Hinweise für das interaktive Testen:**

#### **Linux-Nutzer:**

Linux-Nutzer können das Ende der Standardeingabe einfach durch Drücken von **STRG-D** (bzw. **CTRL-D**) (gleichzeitiges Drücken der Tasten **STRG** und **D**) erzeugen.

#### **Windows-Nutzer:**

Unter Windows hat das interaktive Testen folgenden Haken: Der Beispielquelltext funktioniert nicht, wenn man das Zeichen **EOF** nicht am Zeilenanfang eingibt. Auf der Windows-Konsole `cmd` wird **EOF** durch die Tastenkombination **STRG-Z** erzeugt. Auch ist es immer notwendig, nach der Eingabe von **STRG-Z** auch noch die **Enter**-Taste zu betätigen. Dieses Verhalten liegt aber nicht an unserem C-Programm sondern wird durch eine andere Instanz gesteuert (vermutlich der Treiber für die Tastatureingabe der Kommandozeile), die außerhalb des Fokus‘ dieser Lehrveranstaltung liegt.

Absolut garstig wird die ganze Sache, wenn man beispielsweise zuerst die Standardeingabe in eine Datei kopiert und dabei **STRG-Z** schon mal irgendwo mitten in einer Zeile hatte. Wie gerade gesagt, wird das mitten in einer Zeile stehende **STRG-Z** nicht als Ende der Eingabe erkannt und daher mit in die Datei geschrieben. Auch die nach dem **STRG-Z** kommenden Zeichen werden in die Datei kopiert. Das sieht man, wenn man sich die Datei mit einem Editor anschaut. Das Programm endet ja erst, wenn **STRG-Z** am Anfang einer Zeile eingegeben wird! Und jetzt lasst das Programm mal genau diese Datei ausgeben ...

## 4. Implementierung

Das Kopieren einer einzigen Datei sollte nach den ganzen Vorüberlegungen eigentlich kein Problem mehr sein. Die zu entwickelnde Funktion bekommt die beiden File Pointer und liest solange von der Eingabe, bis dort das Dateiende erreicht wurde. Jedes gelesene Zeichen muss auch ausgegeben werden. Parallel dazu muss für jedes Zeichen der Zeichenzähler um eins erhöht werden.

Kopieren einer Datei

```
Parameter: Pointer to FILE: fin, fout
Variablen: Integer: c, counter

setze counter = 0
lese ein zeichen von fin

solange zeichen  $\neq$  EOF
wiederhole schreibe zeichen auf fout
    lese ein zeichen von fin
    setz counter = counter + 1

return counter
```

## 5. Kodierung

```
1  #include <stdio.h>
2
3  int copyFile( FILE *fin, FILE *fout )
4  {
5      int c, cnt;
6      c = fgetc( fin );          // erstes zeichen lesen
7      for( cnt = 0; c != EOF; cnt++ )
8      {
9          fputc( c, fout );      // zeichen schreiben
10         c = fgetc( fin );      // neues zeichen lesen
11     }
12     return cnt;                // zahl der kopierten zeichen
13 }
```

In der Literatur kopiert man Dateien meist mit folgender (verkürzten) Sequenz:

```
1      while((c = fgetc( fin )) != EOF )
2          fputc( c, fout );
```

Hier wird im Schleifenkopf das nächste Zeichen gelesen `c = fgetc( fin )` und das Ergebnis der Zuweisung sogleich auf `EOF` geprüft. Diese Variante arbeitet natürlich genauso gut. Nur müssen wir jetzt noch das Zählen der kopierten Zeichen integrieren, was einfach ist:

```

1  int copyFile( FILE *fin, FILE *fout )
2      {
3          int c, cnt;
4          for( cnt = 0; (c = fgetc( fin )) != EOF; cnt++ )
5              fputc( c, fout );
6          return cnt;
7      }

```

## Aufgabe 2: Das komplette cat-Kommando

### 1. Aufgabenstellung

Ergänze die in der vorherigen Aufgabe entwickelte Funktion `copyFile()` um ein entsprechendes Hauptprogramm, das die angeforderten Dateien öffnet und schließt. Entsprechende Beispiele haben wir zu Genüge in den Vorüberlegungen besprochen. Es sei nochmals daran erinnert, dass der Dateiname – für die Tastatur steht.

### 2. Pflichtenheft

Aufgabe : Entwicklung eines Hauptprogramms, das die Funktion `copyFile()` mit den richtigen Parametern aufruft.  
 Ausgabe : Das Programm schreibt immer auf die Standardausgabe.  
 Parameter : Die einzelnen Kommandozeilenargumente  
 Sonderfälle: Falls keine Parameter angegeben werden, wird von der Tastatur gelesen. Selbiges wenn der Dateiname "-" lautet.

### 3. Implementierung

Bei der Implementierung müssen wir nur die entsprechenden Fälle berücksichtigen:

Hauptprogramm: Kopieren von Dateien

```

wenn Parameter vorhanden sind: argc > 1
dann für alle Argumente
    wiederhole wenn Dateiname ≠ "-"
        dann öffnen der Datei
            falls kein Fehler auftrat
                dann copyFile( file, stdout )
                sonst Fehlermeldung ausgeben
            sonst copyFile( stdin, stdout )
    sonst copyFile( stdin, stdout )

```

### 4. Kodierung

Unser fertiges Programm sieht wie folgt aus:

```
1  #include <stdio.h>
2
3  int copyFile( FILE *fin, FILE *fout )
4  {
5      int c, cnt;
6      for( cnt = 0; (c = fgetc( fin )) != EOF; cnt++ )
7          fputc( c, fout );
8      return cnt;
9  }
10
11 int main( int argc, char **argv )
12 {
13     int i;
14     FILE *fin;
15     if ( argc > 1 )
16         for( i = 1; i < argc; i++ )
17             if ( strcmp( argv[ i ], "-" ) )
18                 if ( fin = fopen( argv[ i ], "r" ) )
19                 {
20                     copyFile( fin, stdout );
21                     fclose( fin );
22                 }
23             else fprintf( stderr, "'%s' konnte nicht
24                          geoeffnet werden\n", argv[ i ] );
25             else copyFile( stdin, stdout );
26     return 0;
27 }
```

## 5. Testdaten

Das Komplettdprogramm kann jetzt einfach so getestet werden, wie wir es in den Vorüberlegungen diskutiert haben.



# Übungspaket 31

## Entwicklung eines einfachen Kellerspeichers (Stacks)

---

### Übungsziele:

1. Definieren einer dynamischen Datenstruktur
2. Dynamische Speicher Speicherallokation
3. Implementierung eines einfachen Stacks (LIFO)

### Skript:

Kapitel: 72 bis 74 sowie insbesondere die Übungspakete 19, 20 und 26.

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

Dynamische Datenstrukturen: Dies ist der Beginn des letzten Teils der Übungspakete. In diesem Übungspaket entwickeln wir einen einfachen Stack. Dabei werden wir im Wesentlichen die Inhalte aus Vorlesung und Skript in einfacher Weise rekapitulieren und wiederholen. Trotz seiner Einfachheit ist dieses Übungspaket eine wichtige Voraussetzung für die Übungspakete 32 und 33.

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Fragen zu Strukturen

Sind Strukturen einfache oder komplexe Datentypen?	Komplexe Datentypen
Was kann man damit also machen?	Unterschiedliche Datentypen zusammenfassen
Kann sich eine Struktur selbst enthalten?	Nein
Falls ja, warum, falls nein, warum nicht?	Der benötigte Platz wäre unendlich groß
Sind rekursive Datenstrukturen erlaubt?	Ja, <i>Zeiger</i> auf sich selbst
Warum ist dies zulässig?	Da alle <i>Zeiger</i> gleich groß sind

## Aufgabe 2: Fragen zu dynamischem Speicher

Arrays zeichnen sich durch zwei wesentliche Merkmale aus:

Die Elemente eines Arrays sind alle...	vom selben Datentyp.
Im Arbeitsspeicher liegen sie alle...	hintereinander angeordnet.

Erkläre mit eigenen Worten, weshalb man die Größe eines Arrays nicht einfach während der Laufzeit verändern kann.

Wie eingangs schon erfragt wurde, müssen vom Konzept her alle Array-Elemente nacheinander im Arbeitsspeicher liegen. Für gewöhnlich liegen davor und/oder danach weitere Daten, sodass man neue Elemente weder an- noch einfügen kann. Man könnte natürlich ein weiteres Array irgendwo anders anlegen. Doch dann würden Zeiger, die bereits in das alte Array zeigen, plötzlich ins Leere gehen. Daher ginge dies nicht automatisch, sondern müsste „per Hand“ erledigt werden.

Welche Funktion liefert neuen Speicher?	<code>malloc()</code>
Wie ist diese Funktion deklariert?	<code>void *malloc( int size )</code>
Woran sieht man, dass die Anfrage erfolgreich war?	Rückgabe eines Zeigers <code>p != 0</code>
Woran sieht man, dass sie <i>nicht</i> erfolgreich war?	Rückgabe eines Zeigers <code>p == 0</code>
Welche Funktion gibt neuen Speicher wieder zurück?	<code>free()</code>
Wie ist diese Funktion deklariert?	<code>void free( void *p )</code>
Muss man den Rückgabewert von <code>free()</code> überprüfen?	Nee, ist ja <code>void</code>

## Aufgabe 3: Allgemeine Fragen zu Listen

Für den Fall, dass man ein Array „zerpflückt“, gilt nicht mehr die alte Eigenschaft, dass sich die Elemente nacheinander im Arbeitsspeicher befinden. Was muss oder kann man einführen, um weiterhin den „Zusammenhang“ des Arrays zu behalten? Was ist der Zusammenhang (das Gemeinsame) zu Listen?

Man benötigt eine zusätzliche Komponente, die angibt, wo sich das nächste Element im Arbeitsspeicher befindet. Wenn dieses zusätzliche Element ein *Zeiger* ist, spricht man von einer linearen Liste.

Was bedeuten die folgenden Begriffe?

Einfach verkettet:	Es gibt einen Zeiger zum Nachfolger <i>oder</i> Vorgänger
Doppelt verkettet:	Je ein Zeiger zum Nachfolger <i>und</i> Vorgänger
Lineare Liste:	Alle Elemente bilden eine Kette

Typischerweise besteht jedes Element einer dynamischen Datenstruktur aus zwei Bestandteilen. Um welche handelt es sich? Erläutere dies mittels eines kleinen Beispiels:

Teil 1:	Die eigentlichen Daten: Beispiel: <code>struct data user_data</code>
Teil 2:	Die notwendigen Verwaltungsinformationen: Beispiel: <code>struct element *next</code>

## Aufgabe 4: Fragen zum Thema Stack

Was für eine Liste ist ein Stack?	Eine einfach verkettete Liste
Wie viele „Startzeiger“ benötigt ein Stack?	Genau einen
Wohin zeigt dieser Startzeiger?	An den Anfang der Liste
Wo werden neue Elemente plazierte? ( <code>push()</code> )	<i>Vor</i> das vorderste Element
Welches Element wird entfernt ( <code>pop()</code> )?	Das vorderste Element
Wie erkennt man einen leeren Stack ( <code>isEmpty()</code> )?	Der Startzeiger ist ein Null-Zeiger

Definiere eine Stack-Datenstruktur, in der jedes Element zwei `int`-Zahlen aufnehmen kann.

```
1 struct dta { int a; int b; };          // a container for two int's
2
3 struct element {
4     struct element *next; // pointer to the next element
5     struct dta data;      // here is the actual data
6 }
```

## Teil II: Quiz

---

### Aufgabe 1: Stack mittels Array

Auf der nächsten Seite befindet sich als Vorbereitung ein Miniprogramm, das einen kleinen Stack mittels eines Arrays realisiert. Aus Platzgründen haben wir auf einige Leerzeilen, Kommentare etc. verzichtet. Zur Verbesserung eurer Analysefähigkeiten haben wir alle Funktionen mit `f1()` ... `f5()` nummeriert. Finde zunächst die Funktionalität der einzelnen Funktionen heraus und vervollständige dann die neue Namenstabelle.

<code>f1()</code> :	In dieser Funktion wird die Komponente <code>level</code> auf <code>RAM_SIZE</code> gesetzt. Dies deutet darauf hin, dass <code>level</code> die Zahl der freien Einträge angibt. Diese Funktion initialisiert die Struktur <code>STACK</code> .
<code>f2()</code> :	Hier wird die Komponente <code>level</code> auf <code>RAM_SIZE</code> abgefragt. Da dieser Wert nur dann auftaucht, wenn der Stack leer ist, wäre <code>isEmpty()</code> ein guter Name.
<code>f3()</code> :	In Zeile 19 wird auf <code>! isEmpty()</code> abgefragt. Ferner wird in Zeile 20 eine Variable an der Aufrufstelle verändert. Und da die Zahl der freien Einträge ( <code>level</code> ) erhöht wird, scheint es sich hier um die Funktion <code>pop()</code> zu handeln.
<code>f4()</code> :	Hier wird abgefragt, ob die Zahl der freien Einträge ( <code>level</code> ) null ist. Entsprechend ist dies die Funktion <code>isFull()</code> .
<code>f5()</code> :	Hier wird auf <code>! isFull()</code> abgefragt, die Zahl der freien Einträge ( <code>level</code> ) verringert und etwas auf den Stack kopiert. Das ist die Funktion <code>push()</code> .

#### Neue Namenszuordnung:

`f1()`: `init()`    `f2()`: `isEmpty()`    `f3()`: `pop()`    `f4()`: `isFull()`    `f5()`: `push()`

Nun wollen wir noch gerne wissen, wie viel freie Plätze am Ende der Programmzeilen 35 bis 41 noch vorhanden sind und welche Werte auf dem Stack liegen.

Zeile	<code>level</code>	Stack				Anmerkungen
35	4	.....	.....	.....	.....	direkt nach der Initialisierung
36	3	.....	.....	.....	.....3	3 rauf, 1 rauf und wieder runter
37	0	.815	4711	...-1	.....3	drei neue Werte
38	2	.....	.....	...-1	.....3	99 wird nicht abgelegt, da Stack voll
39	3	.....	.....	.....	.....3	66 rauf und wieder runter, -1 runter
40	4	.....	.....	.....	.....	eine 0 rauf, 0 und 3 wieder runter
41	4	.....	.....	.....	.....	weniger als leer geht nicht ;-)

### Kleiner Stack mittels Array implementiert: (nur zu Übungszwecken)

```
1  #include <stdio.h>
2
3  #define RAM_SIZE      4
4  struct STACK {
5      int level;
6      int ram[ RAM_SIZE ];
7  };
8
9  void f1( struct STACK *ram )
10 {
11     ram->level = RAM_SIZE;
12 }
13 int  f2( struct STACK *ram )
14 {
15     return ram->level == RAM_SIZE;
16 }
17 void f3( struct STACK *ram, int *vp )
18 {
19     if ( ! f2( ram ) )
20         *vp = ram->ram[ (ram->level)++ ];
21 }
22 int  f4( struct STACK *ram )
23 {
24     return ram->level == 0;
25 }
26 void f5( struct STACK *ram, int val )
27 {
28     if ( ! f4( ram ) )
29         ram->ram[ --(ram->level) ] = val;
30 }
31 int main( int argc, char **argv )
32 {
33     struct STACK stack, *sp = & stack;
34     int i;
35     f1( sp );
36     f5( sp, 3 ); f5( sp, 1 ); f3( sp, & i );
37     f5( sp, -1 ); f5( sp, 4711 ); f5( sp, 815 );
38     f5( sp, 99 ); f3( sp, & i ); f3( sp, & i );
39     f5( sp, 66 ); f3( sp, & i ); f3( sp, & i );
40     f5( sp, 0 ); f3( sp, & i ); f3( sp, & i );
41     f3( sp, & i ); f3( sp, & i );
42 }
```

## Teil III: Fehlersuche

---

### Aufgabe 1: Fehler beim Aufbau einer Liste

Unser Lead-Stacker DR. WELL STACK wollte für sein Lager eine Liste programmieren. Die Liste soll aus drei Elementen bestehen, die als Inhalte die Regalnummern 1, 2 und 3 haben sollen. Offensichtlich wird wieder einmal eure Hilfe beim Finden und Korrigieren der Fehler benötigt.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct shelf {
5      int i;
6  };
7
8  struct stack {
9      struct stack next;
10     struct shelf data;
11 };
12
13 struct stack newElement( int i )
14 {
15     struct stack *p;
16     p = malloc( sizeof(struct stack) );           // get ram
17     p->data.i = i;                                // initializing data
18     return *p;                                    // return the new element
19 }
20
21 void prt_chain( struct stack *p )
22 {
23     for( ; p; p = p->next )                        // loop through the list
24         printf( "i=%d\n", p->data.i );             // print item
25 }
26
27 int main( int argc, char *argv )
28 {
29     struct stack *p, *sp;                          // init to empty
30     p = newElement( 2 ); p->next = sp; sp = p;      // first
31     p = newElement( 3 ); sp->next = p;              // 3 behind 2
32     p = newElement( 1 ); sp = p; p->next = sp;     // new start
33     prt_chain( & sp );                             // let's see what we've got
34 }
```

Zeile	Fehler	Erläuterung	Korrektur
9	* fehlt	In dieser Form würde sich die Struktur <code>struct stack</code> durch <code>next</code> selbst enthalten, was nicht erlaubt ist. Hier müsste ein <code>next-Zeiger</code> stehen.	<code>*next</code>
13/18	falscher Typ	Die Funktion <code>newElement()</code> hat den falschen Typ. Sie legt mittels <code>malloc()</code> ein neues Element an und erhält einen Zeiger auf diesen Bereich zurück, da dieser nicht über irgendwelche Variablen ansprechbar ist. Um auch außerhalb der Funktion <code>newElement()</code> auf diesen Speicherbereich zugreifen zu können, muss sie ebenfalls einen Zeiger zurückgeben. Entsprechend müssen wir den Funktionstyp auf <code>struct stack *</code> und den <code>return</code> -Wert auf <code>p</code> ändern.	<code>struct stack *</code> und <code>return p</code>
16/17	fehlende Überprüfung des Zeigers	In Zeile 16 zeigt <code>p</code> auf den von <code>malloc()</code> organisierten Speicherbereich. Ist aber kein weiterer Speicher verfügbar, liefert <code>malloc()</code> einen Null-Zeiger und das Dereferenzieren in Zeile 17 würde zu einem Programmabsturz führen. Ergo muss die Zeile 17 durch ein <code>if (p)</code> „abgesichert“ werden.	<code>if (p)</code> einfügen
29/30	fehlende Initialisierung	Der Zeiger <code>sp</code> wird in Zeile 29 definiert und in Zeile 30 verwendet, aber nirgends initialisiert. Ein Zeigerwert ungleich <code>null</code> kann in der Funktion <code>prt_chain()</code> (Zeilen 21-25) zum Programmabsturz führen, da das Abbruchkriterium (Erreichen des Null-Zeigers) wirkungslos wird.	<code>sp = 0</code>
31	falsche Verzeigerung	Die beiden Anweisungen in Zeile 31 führen leider dazu, dass die bestehende Liste völlig „vergurkt“ wird; der vorhandene Teil hinter dem ersten Element wird leider durch die zweite Anweisung abgehängt und damit <i>vergessen</i> . Zuerst muss dieser Teil an das neue Element angehängt werden.	<code>p-&gt;next = sp-&gt;next</code> vor <code>sp-&gt;next = p</code> einfügen
32	falsche Verzeigerung	Durch die erste Zeigerzuweisung <code>sp = p</code> geht die bisherige Liste verloren. Daher: Erst die aktuelle Liste durch Anhängen an das neue Element sichern, dann den Startzeiger neu setzen.	Zeigerzuweisungen vertauschen
33	& zu viel	Die Funktion <code>prt_chain()</code> erwartet einen Zeiger auf das erste Listenelement. Dies wäre bereits durch <code>sp</code> gegeben. Der Ausdruck <code>&amp; sp</code> führt zum falschen Datentyp und damit zum falschen Wert.	<code>sp</code>

### Programm mit Korrekturen:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct shelf {
5      int i;
6  };
7
8  struct stack {
9      struct stack *next;
10     struct shelf data;
11 };
12
13 struct stack *newElement( int i )
14 {
15     struct stack *p;
16     if (p = malloc( sizeof(struct stack) )) // get ram
17         p->data.i = i;                     // initializing data
18     return p;                             // return the new element
19 }
20
21 void prt_chain( struct stack *p )
22 {
23     for( ; p; p = p->next )                // loop through the list
24         printf( "i=%d\n", p->data.i );      // print item
25 }
26
27 int main( int argc, char *argv )
28 {
29     struct stack *p, *sp = 0;               // init to empty
30     p = newElement( 2 ); p->next = sp; sp = p; // first
31     p = newElement( 3 ); p->next = sp->next; sp->next = p;
32     p = newElement( 1 ); p->next = sp; sp = p; // new start
33     prt_chain( sp );                       // let's see what we've got
34 }
```



# Teil IV: Anwendungen

---

Das wesentliche Ziel dieses Übungspaketes ist die Entwicklung eines ersten, dynamisch arbeitenden Stacks. Obwohl im Prinzip einfach, bereitet diese Aufgabe so gut wie jedem Programmieranfänger zunächst einmal Schwierigkeiten. Das ist so und sollte man auch so akzeptieren. Daher haben wir diese Aufgabe in kleinere Stücke aufgeteilt. Am Ende wird Aufgabe 7 noch einen zweiten Stack zum Thema haben, der ganze Zeichenketten aufnehmen kann.

## Aufgabe 1: Definition einer geeigneten Datenstruktur

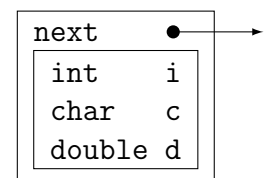
### 1. Aufgabenstellung

Unser Kellerspeicher soll so organisiert sein, dass jedes Element je einen `int`, `char` und `double`-Wert aufnehmen kann. Wie schon mehrfach besprochen, sollen dabei die eigentlichen Daten von den Verwaltungsinformationen getrennt werden.

Aufgabe: Definiere zwei entsprechende Strukturen.

### 2. Implementierung

Ok, was wissen wir? Wir sollen einen `int`, einen `char` und einen `double`-Wert ablegen können. Da es sich um unterschiedliche Datentypen handelt, benötigen wir eine Struktur. Diese nennen wir `struct user`. Ferner benötigen wir einen Zeiger auf das nächste Element. Da ein Zeiger etwas anderes als die Nutzerdaten ist, benötigen wir noch eine zweite Struktur. Diese nennen wir `struct stack`. Beiden Strukturen können wir in C so einfach formulieren, wie wir es weiter oben schon gemacht haben.



### 3. Kodierung

```
1 struct user {
2     int i; char c; double d;           // the data items
3 };
4
5 struct stack {
6     struct stack *next;                // link to next entry
7     struct user data;                  // the user data record
8 };
```

## Aufgabe 2: Allokation eines neuen Stack-Elementes

### 1. Aufgabenstellung

Wir benötigen eine Funktion `newElement()`, die uns genau ein neues Stack-Element *dynamisch* generiert. Bei derartigen Funktionen ist es meist ratsam (aber nicht zwingend), die einzelnen Komponenten sinnvoll zu initialisieren. In unserem Fall könnten wir die Werte für die Nutzerdaten direkt übergeben. Welcher Wert wäre für den Zeiger auf das nächste Element sinnvoll? Der Null-Zeiger

Aufgabe: Entwickle eine Funktion `newElement()`, die ein neues Element dynamisch alloziert und gleichzeitig alle Komponenten auf sinnvolle Werte setzt.

### 2. Entwurf

Vervollständige die folgende Funktionsdeklaration:

```
newElement(): struct stack *newElement( int i, char c, double d );
```

### 3. Implementierung

Den Funktionskopf haben wir uns eben erarbeitet. Die Funktion muss folgende Schritte abarbeiten: (1) neuen Speicher mittels `malloc()` organisieren, (2) im Erfolgsfalle die einzelnen Elemente initialisieren und (3) den neuen Zeiger zurückgeben.

Funktion `newElement()`

Parameter: Integer: `i`; Zeichen: `c`; Double: `d`

Variable: Pointer to Stack: `p`

setze `p = malloc( sizeof( struct Stack ) )`

wenn `p ≠ Null-Zeiger`

dann setze wie folgt: `p->i = i; p->c = c; p->d = d; p->next = 0`

return `p`

### 4. Kodierung

```
1 struct stack *newElement( int i, char c, double d )
2     {
3         struct stack *p;
4         if (p = malloc( sizeof( struct stack ) ))
5         {
6             p->data.i = i; p->data.c = c;
7             p->data.d = d; p->next = 0;
8         }
9         return p;
10    }
```

## Aufgabe 3: Anlegen eines neuen Stacks

### 1. Aufgabenstellung

Wir haben schon öfters darüber gesprochen, dass sich ein leerer Stack dadurch auszeichnet, dass sein Startzeiger ein Null-Zeiger ist. Daher wird der Startzeiger in der Regel einfach im Hauptprogramm auf null gesetzt. Das ist praktisch und auch üblich so. Hat nur einen Nachteil: Später könnte es wichtig sein, dass wir noch mehr machen müssen, als einen Startzeiger auf null zu setzen.

Vielleicht haben wir in einem Programm mehrere Stacks zu verwalten, vielleicht wollen wir uns merken, welcher Stack was macht. Von daher ist es günstig, sich schon jetzt anzugewöhnen, nicht direkt den Startzeiger zu „manipulieren“, sondern dafür eine Funktion zu verwenden. Ja, das ist auch dann sinnvoll, wenn die Funktion nichts anderes macht, als einen Null-Zeiger zu liefern. Sollten wir doch mehr benötigen, können wir diese Funktion einfach erweitern. Bei dieser Gelegenheit können wir direkt im Anschluss eine Funktion schreiben, die testet, ob ein Stack leer ist. Wie gesagt, im Moment erscheint dies vielleicht etwas übertrieben. Aber irgendwann wird es sich mehrfach auszahlen!

Aufgabe: Entwickle eine Funktion, die einen Stack initialisiert. Entwickle eine weitere Funktion, die einen Stack daraufhin überprüft, ob er leer ist oder nicht. Diese beiden Funktionen nennen wir `newStack()` und `isEmpty()`.

### 2. Implementierung

Gemäß Aufgabenstellung soll die Funktion `newStack()` einen leeren Stack „erzeugen“. Dafür muss sie gemäß obiger Diskussion lediglich einen Null-Zeiger zurückliefern. Der Datentyp dieser Funktion ist dann `struct stack *`.

Die zweite Funktion soll überprüfen, ob ein Stack leer ist oder nicht. Dazu benötigt sie einen Zeiger auf den Stack, der den Typ `struct stack *` hat.

Beide Funktionen sind so einfach, dass wir sie direkt kodieren können.

### 3. Kodierung

```
1 struct stack *newStack()
2     {
3         return 0;
4     }
5
6 int isEmpty( struct stack *sp)
7     {
8         return sp == 0;
9     }
```

## Aufgabe 4: push(): Daten auf den Stack packen

### 1. Aufgabenstellung

Wir benötigen eine Funktion zum Ablegen der Daten auf den Stack. Dazu müssen wir entscheiden, ob das *neue Element* bereits vorliegt, oder innerhalb der Funktion `push()` erzeugt wird. Beide Lösungen sind sinnvoll und angemessen. In größeren Anwendungen kann es sinnvoll sein, beide Varianten anzubieten, wir wählen die zweite.

Aufgabe: Entwickle eine Funktion `push()` die ein neues Element erzeugt und auf dem Stack ablegt.

### 2. Entwurf

Werden die Nutzerdaten in <code>push()</code> verändert?	Nein
Wird der Startzeiger in <code>push()</code> verändert?	Ja
Kann die Funktion <code>push()</code> fehlschlagen?	Ja, bei Speicherknappheit
Funktionsdeklaration:	<code>int push(struct stack **sp,int i,char c,double d);</code>

### 3. Implementierung

Wir müssen uns ein neues Stack-Element besorgen und die Zeiger neu setzen.

Funktion `push()`

Rückgabewert: Integer

Parameter: Pointer to Pointer to Stack: `sp`

Integer: `i`; Zeichen: `c`; Double `d`

Variablen: Pointer to Stack: `p`

setze `p = newElement( i, c, d )`

wenn `p ≠ Null-Zeiger`

dann setze `p->next = sp` dereferenziert und `sp dereferenziert = p`

return `p ≠ 0`

### 4. Kodierung

```
1 int push( struct stack **sp, int i, char c, double d )
2     {
3         struct stack *p = newElement( i, c, d );
4         if ( p )
5         {
6             p->next = *sp; *sp = p;
7         }
8         return p != 0;
9     }
```

## Aufgabe 5: pop(): Daten vom Stack entfernen

### 1. Aufgabenstellung

Jetzt fehlt noch eine Funktion, die die Daten wieder vom Stack holt. Wir entscheiden uns dafür, dass wir neben dem Entfernen des obersten Elementes auch gleich noch die Daten über entsprechende Zeiger an die aufrufende Stelle zurück liefern. Natürlich können wir ein Element nur dann entfernen, wenn auch eines vorhanden ist. Um mit obiger Entscheidung konsistent zu bleiben, führen wir diese Überprüfung innerhalb der Funktion pop() durch.

Aufgabe: Entwickle eine Funktion pop() die das oberste Element auf dem Stack entfernt und die abgelegten Daten an die aufrufende Stelle zurückgibt.

### 2. Entwurf

Wie erhält der Aufrufer die Nutzerdaten?	Über entsprechende Zeiger
Kann die Funktion fehlschlagen?	Jein, der Stack kann leer sein
Was heißt das für den Rückgabewert?	Wir können ! isEmpty() zurückgeben
Wird der Startzeiger verändert?	Ja
Wie muss er übergeben werden?	Als Adresse auf den Startzeiger
Gibt es eine Alternative?	Ja, ptr = pop( ptr, ...)
Vervollständige die folgende Funktionsdeklaration:	

pop(): `int pop( struct stack **sp, int *i, char *c, double *d );`

### 3. Implementierung

Da diese Funktion sehr ähnlich wie push() ist, beginnen wir gleich mit der Kodierung.

### 4. Kodierung

```
1  int pop(struct stack **sp, int *ip, char *cp, double *dp)
2      {
3          struct stack *t = *sp;    // pointer to top element
4          int empty = isEmpty( t );    // test and save
5          if ( ! empty )
6          {
7              *ip = t->data.i; *cp=t->data.c; *dp=t->data.d;
8              *sp = t->next;    // advance to next element
9              free( t );    // free/remove the top element
10         }
11         return ! empty;
12     }
```

# Aufgabe 6: Das Hauptprogramm

## 1. Aufgabenstellung

Natürlich benötigen wir auch ein kleines Hauptprogramm zum Testen unseres Stacks. Dazu bauen wir uns noch eine kleine Funktion zum Ausgeben der Daten. Anschließend rufen wir einfach die `push()` und `pop()`-Funktionen auf und schauen, ob alles richtig arbeitet.

## 2. Kodierung

```
1 void prt_dat( int i, char c, double d )
2     {
3         printf( "i=%d c='%c' d=%e\n", i, c, d );
4     }
5
6 int main( int argc, char **argv )
7     {
8         struct stack *sp;
9         int i; char c; double d;
10
11         sp = newStack();
12
13         push( & sp, 1, 'a', 1.0 );
14         push( & sp, 2, 'b', 2.0 );
15         push( & sp, 3, 'c', 3.0 );
16
17         pop( & sp, & i, & c, & d ); prt_dat( i, c, d );
18         pop( & sp, & i, & c, & d ); prt_dat( i, c, d );
19
20         push( & sp, 2, 'B', -2.0 );
21         push( & sp, 3, 'C', -3.0 );
22
23         while( ! isEmpty( sp ) )
24         {
25             pop( & sp, & i, & c, & d );
26             prt_dat( i, c, d );
27         }
28     }
```

Das wär's. Nun erst einmal tief durchatmen und dann weiter schauen. Anschließend sollte sich jeder Programmieranfänger nochmals ein Speicherbildchen malen und die einzelnen Funktionsaufrufe per Hand simulieren. Zur Erinnerung: Diese Übungen erscheinen den meisten Anfängern recht lästig, mühsam und eher überflüssig zu sein, doch tragen sie sehr zum Verständnis der Konzepte bei.

## Nachlese

Auch wenn ihr froh seid, mit dem Programm fertig zu sein, so lohnt sich dennoch eine Nachlese. Auf Dauer sind diese langen Zeigerdefinitionen `struct stack *` recht mühsam. Wir erinnern uns an das `typedef` aus Kapitel 54. Die Syntax ist wie folgt: `typedef <bekannter_Typ> <neuer_Typ>`. Damit lassen sich neue Typen definieren, die einiges an Schreibarbeit ersparen. Für uns bieten sich folgende Definitionen an:

```
4 typedef struct user {
5     int i; char c; double d;          // the data items
6 } USER;
7
8 typedef struct _stack {
9     struct _stack *next;              // link to next entry
10    struct user data;                  // the user data record
11 } STACK, *SP;
```

Die Zeilen 8-11 zeigen deutlich, dass wir eine Struktur `STACK` sowie einen Zeiger-Typ `SP` definiert haben, die nichts anderes als `struct _stack` und `struct _stack *` sind. Desweiteren haben wir unsere Funktion `newElement()` dahingehend erweitert, dass wir ihr den `next`-Zeiger auch gleich mitgeben.

```
23 SP newElement( int i, char c, double d, SP next )
24 {
25     SP p = malloc( sizeof( STACK ) );
26     if ( p )
27     {
28         p->data.i = i; p->data.c = c;
29         p->data.d = d; p->next = next;
30     }
31     return p;
32 }
```

Dadurch wird dieser Zeiger im neuen Element nicht einfach auf null gesetzt, sondern bekommt einen Wert, der für die aufrufende Stelle sinnvoll ist, was das Einhängen neuer Elemente etwas vereinfacht:

```
34 int push( SP *stack, int i, char c, double d )
35 {
36     SP p = newElement( i, c, d, *stack );
37     if ( p )
38         *stack = p;
39     return p != 0;
40 }
```

### Der modifizierte Stack:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct user {
5      int i; char c; double d;          // the data items
6      } USER;
7
8  typedef struct _stack {
9      struct _stack *next;              // link to next entry
10     struct user data;                  // the user data record
11     } STACK, *SP;
12
13 SP newStack()
14 {
15     return 0;
16 }
17
18 int isEmpty( SP sp )
19 {
20     return sp == 0;
21 }
22
23 SP newElement( int i, char c, double d, SP next )
24 {
25     SP p = malloc( sizeof( STACK ) );
26     if ( p )
27     {
28         p->data.i = i; p->data.c = c;
29         p->data.d = d; p->next = next;
30     }
31     return p;
32 }
33
34 int push( SP *stack, int i, char c, double d )
35 {
36     SP p = newElement( i, c, d, *stack );
37     if ( p )
38         *stack = p;
39     return p != 0;
40 }
```



```

42 SP pop( SP p, int *ip, char *cp, double *dp )
43 {
44     SP old = p;
45     if ( ! isEmpty( p ) )
46     {
47         *ip = p->data.i; *cp = p->data.c; *dp = p->data.d;
48         p = p->next; free( old );
49     }
50     return p;
51 }
52
53 void prt_dat( int i, char c, double d )
54 {
55     printf( "i=%d c='%c' d=%e\n", i, c, d );
56 }
57
58 int main( int argc, char **argv )
59 {
60     SP sp;
61     int i; char c; double d;
62
63     sp = newStack();
64
65     push( & sp, 1, 'a', 1.0 );
66     push( & sp, 2, 'b', 2.0 );
67     push( & sp, 3, 'c', 3.0 );
68
69     sp = pop( sp, & i, & c, & d ); prt_dat( i, c, d );
70     sp = pop( sp, & i, & c, & d ); prt_dat( i, c, d );
71
72     push( & sp, 2, 'B', -2.0 );
73     push( & sp, 3, 'C', -3.0 );
74
75     while( ! isEmpty( sp ) )
76     {
77         sp = pop( sp, & i, & c, & d );
78         prt_dat( i, c, d );
79     }
80 }

```

Ganz nebenbei könnten wir die beiden Funktionen `newStack()` und `isEmpty()` (Zeilen 13 bis 21) durch zwei sehr einfache `#defines` ersetzen:

```

13 #define newStack()      0          // just like return 0
14 #define isEmpty( p )    (p == 0)  // like in the original

```

## Aufgabe 7: Kellerspeicher für Zeichenketten

Ziel dieser Aufgabe ist die Entwicklung eines weiteren Kellerspeichers. Diesmal sollen aber nicht Zahlen oder Buchstaben sondern ganze Zeichenketten auf dem Stack abgelegt werden. Wir haben bereits gelernt (beispielsweise in Übungspaket 24), dass Zeichenketten irgendwie anders als Zahlen sind. Daher wiederholen wir zunächst ein bisschen alten Stoff.

### 1. Vorüberlegungen und Stoffwiederholung

Welche der folgenden Werte sind einfach bzw. komplex? Notiere je ein Beispiel:

Einzelne Zahlen:	Einfacher Datentyp: <code>int i</code>
Einzelne Zeichen:	Einfacher Datentyp: <code>char c</code>
Mehrere Zahlen:	Komplexer Datentyp: <code>double coeff[ 10 ]</code>
Zeichenketten:	Komplexer Datentyp: <code>char str[ 6 ] = "hallo"</code>

Vervollständige das Speicherbildchen für die folgenden Programmzeilen:

```
1 int i = 4711, j = i;  
2 char *s1 = "hallo", *s2 = s1;
```

Segment:	Stack		Segment:	Konstanten
Adresse	Variable	Wert	Adresse	Wert
0xFE7C	<code>int i:</code>	4711	0xF80C	
0xFE78	<code>int j:</code>	4711	0xF808	
0xFE74	<code>char *s1:</code>	0xF800	0xF804	'o' '\0'
0xFE70	<code>char *s2:</code>	0xF800	0xF800	'h' 'a' 'l' 'l'

Wie oft ist die 4711 im Arbeitsspeicher?	Zwei Mal
Wie oft ist die Zeichenkette "hallo" im Arbeitsspeicher?	Ein Mal
Was wird bei einer Zahl kopiert?	Der Wert
Was wird bei einer Zeichenkette kopiert?	Nur ihre Adresse

Es hängt immer von der konkreten Anwendung und den verwendeten Algorithmen ab, ob das Arbeiten mit Zeichenketten ein Problem ist oder nicht. In den beiden folgenden Fällen ist es notwendig, Kopien einzelner Zeichenketten anzulegen, was wir bereits in Übungspaket 24 geübt haben: erstens, wenn alle Zeichenketten voneinander unterschieden werden müssen und zweitens, wenn sich einzelne Zeichenketten im Laufe des Programms ändern können. Kopien von Zeichenketten müssen insbesondere dann angelegt werden, wenn sie mittels eines Eingabepuffers von der Tastatur gelesen werden. Versäumt man dies, zeigen alle Zeiger in den Eingabepuffer, sodass sich die Inhalte zwar ständig ändern aber alle Zeichenketten immer identisch sind.

## 2. Aufgabenstellung

Entwickle einen Kellerspeicher, der ganze Zeichenketten auf dem Stack ablegt. Diese Aufgabe kann durch kleine Änderungen der vorherigen Programme gelöst werden. In unserem Fall sollen die einzelnen Zeichenketten unterschieden bzw. verändert werden können. Inwiefern müssen wir die `push()`-Funktion anpassen?

In der Funktion `push()` müssen die Zeichenketten immer dupliziert werden.

## 3. Kodierung

Typdefinitionen und zwei Einzeiler:

```
1  #include <stdio.h>                                // for I/O
2  #include <string.h>                                // for strlen() and strcpy()
3  #include <stdlib.h>                                // malloc() and free()
4
5  typedef struct user {
6      char *str;                                     // the data items
7  } USER;
8
9  typedef struct _stack {
10     struct _stack *next;                          // link to next entry
11     USER data;                                     // the user data record
12 } STACK, *SP;
13
14 #define newStack()    0                            // just like return 0
15 #define isEmpty( p )  (p == 0)                    // like in the original
```

Erzeugen eines neuen Stack-Elementes:

```
17 SP newElement( char *str, SP next )
18 {
19     SP p = malloc( sizeof( STACK ) );
20     if ( p )
21         if ( p->data.str = malloc(strlen( str ) + 1))
22         {
23             strcpy( p->data.str, str );
24             p->next = next;
25         }
26     else {                                           // string duplication failed
27         free( p );                                  // don't forget this!
28         p = 0;
29     }
30     return p;
31 }
```

**Die push()-Funktion; easy and standard:**

```
33 int push( SP *stack, char *str )
34     {
35         SP p = newElement( str, *stack );
36         if ( p )
37             *stack = p;
38         return p != 0;
39     }
```

**Stack-Element entfernen: pop():**

```
41 SP pop( SP p, char **str )
42     {
43         SP old = p;
44         if ( ! isEmpty( p ) )
45         {
46             *str = p->data.str; p = p->next; free( old );
47         }
48         return p;
49     }
```

**Typdefinition und zwei Einzeiler:**

```
51 int main( int argc, char **argv )
52     {
53         SP sp;
54         char *str;
55
56         sp = newStack();
57
58         push( & sp, "hey buddy" );
59         push( & sp, "who is this" );
60
61         sp = pop( sp, & str ); printf( "%s\n", str );
62         sp = pop( sp, & str ); printf( "%s\n", str );
63
64         push( & sp, "vacation is over" );
65         push( & sp, "see you on the beach" );
66
67         while( ! isEmpty( sp ) )
68         {
69             sp = pop( sp, & str );
70             printf( "%s\n", str );
71         }
72     }
```

# Übungspaket 32

## Einfach verkettete, sortierte Liste

---

### Übungsziele:

1. Aufbau einer einfach verketteten, sortierten Liste
2. Traversieren von Listen
3. Vereinfachung durch ein Dummy-Element

### Skript:

Kapitel: 75 und insbesondere Übungspaket 29 und 31

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket werden wir endlich eine einfach verkettete, sortierte Liste programmieren. Dabei orientieren wir uns an der in der Vorlesung vorgestellten Wirth'schen Variante, die das Programmieren deutlich vereinfacht. Da das Programmieren sortierter Listen erst mal recht komplex ist, werden wir die wesentlichen Aspekte wiederholen bzw. gezielt darauf hin arbeiten. Also, nicht verzweifeln, sondern einfach probieren...

# Teil I: Stoffwiederholung

---

## Aufgabe 1: Allgemeine Fragen zu Listen

Erkläre mit eigenen Worten, was eine einfach verkettete, sortierte Liste ist.

Eine Liste besteht aus Elementen, die hintereinander aufgereiht sind. Der Begriff „linear“ besagt, dass es in dieser Liste keinerlei Verzweigungen oder dergleichen gibt. Mit Ausnahme von Anfang und Ende gibt es also immer genau einen Nachfolger sowie einen Vorgänger. Schließlich besagt „sortiert,“ dass die einzelnen Elemente hinsichtlich *irgendeines* Kriteriums auf- oder absteigend sortiert sind.

Was versteht man unter „Traversieren“?

Traversieren bedeutet, dass man eine komplexe Datenstruktur elementweise „abarbeitet“. Ein Beispiel wäre, dass man eine lineare Liste Element für Element ausgibt. Traversieren könnte aber auch sein, dass man Element für Element löscht.

Was ist der Unterschied zwischen einer linearen, sortierten Liste und einem Stack?

Von der Struktur her sind beide identisch. Ausgehend von einem Startzeiger folgt Element nach Element, bis man das Ende der Liste erreicht hat. Der einzige Unterschied betrifft die Stelle, an der man neue Elemente einfügt (und wieder entfernt). Bei einem Stack wird immer vorne eingefügt und auch wieder entfernt, weshalb man einen Stack auch mit dem Begriff FIFO für „first in first out“ attribuiert. In einer sortierte Liste hingegen, werden die neuen Elemente immer nur dort eingefügt, wo sie die Eigenschaft „sortiert“ aufrecht erhalten; dies kann prinzipiell also an jeder beliebigen Stelle der Liste sein.

## Aufgabe 2: Detailfragen zu Listen

Die folgenden Fragen beziehen sich immer auf eine einfach verkettete, sortierte Liste oder deren Elemente.

Wie viele Nachfolger hat jedes Element?	Einen, mit Ausnahme des letzten Elementes
Wie viele Vorgänger hat jedes Element?	Einen, mit Ausnahme des ersten Elementes
Woran erkennt man das Listenende?	Der „ <b>next</b> “-Zeiger ist ein Null-Zeiger
Wie findet man den Listenanfang?	Mittels eines Startzeigers
Wie kann man eine Liste ausgeben?	Am einfachsten mit einer <b>for</b> -Schleife
Wie kann man sie invertiert ausgeben?	Am einfachsten rekursiv
Wie muss man sich das vorstellen?	Erst den Rest, dann das aktuelle Element

## Aufgabe 3: Einfügen neuer Elemente in eine Liste

Im Skript haben wir recht ausführlich über die algorithmische Umsetzung des sortierten Einfügens geschrieben. Zur Rekapitulation haben wir eine Reihe von Fragen zusammengetragen.

Welche vier Fälle muss man beim Einfügen neuer Elemente beachten?

1. Die Liste kann leer sein; der Startzeiger ist ein Null-Zeiger
2. Das neue Element kommt vor das erste Element
3. Das neue Element kommt irgendwo mitten drin hinein
4. Das neue Element muss an das Ende angehängt werden

In wie vielen Fällen wird der Startzeiger verändert? 2 (zwei)

Ist das programmiertechnisch gut oder schlecht? It's: Pain in the Ass

Warum? Man benötigt Zeiger auf Zeiger

In den meisten Fällen benutzt man eine Schleife, um diejenige Stelle zu finden, an der man einfügen möchte. Welches Programmierproblem ergibt sich dadurch?

In der Regel bleibt man ein Element zu spät stehen. Daher benötigt man entweder einen zweiten Zeiger, der ein Element hinterher hinkt, oder muss bei der Schleifenbedingung ein weiteres Element vorausschauen. Beides ist nicht besonders schön, lässt sich aber im Standardalgorithmus nicht vermeiden.

## Aufgabe 4: Die Wirth'sche Variante

Im Skript haben wir auch die Variante von Wirth diskutiert. Was ist die wesentliche Idee?

Es gibt ein zusätzliches Element, das immer am Ende der Liste zu finden ist. Dieses Element gehört zwar technisch zur Liste, ist aber nicht Bestandteil der Datenelemente.

Die Wirth'sche Idee hat einige Besonderheiten und Vorteile. Hierzu folgende Fragen:

Wird der Startzeiger verändert? Nein, *nie*, er zeigt immer auf das selbe Element

Wie viele Fälle werden unterschieden? Es gibt nur einen Fall

Wofür ist das Dummy Element? Der „Suchzeiger“ bleibt spätestens hier stehen

Wo bleibt der „Suchzeiger“ stehen? Ein Element zu spät

Wie wird dann aber eingefügt? Durch geschicktes Vertauschen

Welchen Algorithmus nehmen wir? 100-pro den Wirth'schen: intelligent und einfach

## Teil II: Quiz

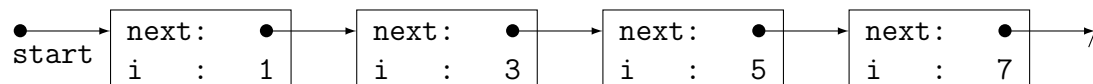
---

### Aufgabe 1: Positionierung innerhalb von Listen

Nehmen wir an, wir haben die folgende Definition einer Liste sowie die folgende Funktion zur Positionierung eines Zeigers.

```
1 typedef struct user { int i; } DATA;
2
3 typedef struct _element {
4     struct _element *next;
5     DATA data;
6 } ELEMENT, *EP;
7
8 EP position( EP listp, int val )
9 {
10     while( listp->data.i < val )
11         listp = listp->next;
12     return listp;
13 }
```

Nun gehen wir davon aus, dass wir bereits die folgende Liste (in unserem Hauptprogramm) aufgebaut haben:



Auf welche Elemente zeigen die Zeiger **start** und **p** nach folgenden Funktionsaufrufen:

Funktionsaufruf	start	p	Anmerkung
p = position( start, 3 )	Element „1“	Element „3“	
p = position( start, -1 )	Element „1“	Element „1“	
p = position( start->next, -1 )	Element „1“	Element „3“	
p = position( start, 6 )	Element „1“	Element „7“	
p = position( start, 7 )	Element „1“	Element „7“	
p = position( start, 8 )	-----	-----	Programmabsturz: Dereferenzieren eines Null-Zeigers

Schlussfolgerung: Programmabsturz, falls **val** größer als der größte Listenwert ist.



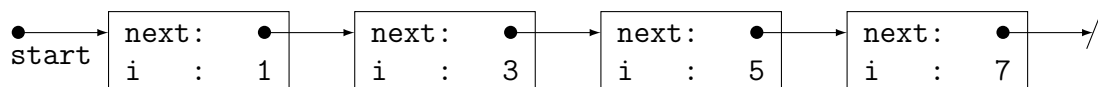
## Aufgabe 2: Positionierung: zweiter Versuch

Aufgrund des vorherigen Programmabsturzes bei Suchwerten, die größer als das größte Element der Liste waren, haben wir unser Programm wie folgt verändert:

```

1  typedef struct user { int i; } DATA;
2
3  typedef struct _element {
4      struct _element *next;
5      DATA data;
6  } ELEMENT, *EP;
7
8  EP position( EP listp, int val )
9  {
10     while( listp->next->data.i < val )
11         listp = listp->next;
12     return listp;
13 }
```

Wir betrachten wieder die gleiche Liste sowie die gleichen Funktionsaufrufe:



Auf welche Elemente zeigen die Zeiger **start** und **p** nach folgenden Funktionsaufrufen:

Funktionsaufruf	start	p	Anmerkung
<code>p = position( start, 3 )</code>	Element „1“	Element „1“	
<code>p = position( start, -1 )</code>	Element „1“	Element „1“	
<code>p = position( start-&gt;next, -1 )</code>	Element „1“	Element „3“	
<code>p = position( start, 6 )</code>	Element „1“	Element „5“	
<code>p = position( start, 7 )</code>	Element „1“	Element „5“	
<code>p = position( start, 8 )</code>	-----	-----	Programmabsturz: Dereferenzieren eines Null-Zeigers

Schlussfolgerung: Wieder Programmabsturz, falls `val` größer als der größte Listenwert ist.

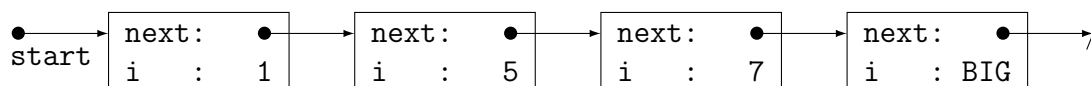
Der Versuch ging nach hinten los, meint unser Team von Star-Programmierern ;-))

## Aufgabe 3: Positionierung: dritter Versuch

So richtig schön waren die beiden vorherigen Versuche nicht. Versuchen wir es also nochmals, diesmal mit einem etwas modifizierten Algorithmus sowie einer modifizierten Liste.

```

1  typedef struct user { int i; } DATA;
2
3  typedef struct _element {
4      struct _element *next;
5      DATA data;
6  } ELEMENT, *EP;
7
8  EP position( EP listp, int val )
9  {
10     while( listp->data.i < val && listp->next != 0 )
11         listp = listp->next;
12     return listp;
13 }
```



Auf welche Elemente zeigen die Zeiger **start** und **p** nach folgenden Funktionsaufrufen:

Funktionsaufruf	start	p	Anmerkung
p = position( start, 3 )	Element „1“	Element „5“	
p = position( start, -1 )	Element „1“	Element „1“	
p = position( start, 6 )	Element „1“	Element „7“	
p = position( start, 7 )	Element „1“	Element „7“	
p = position( start, 8 )	Element „1“	Element „BIG“	Spätestens bei diesem Element bleibt p stehen.

Schlussfolgerung:	Kein Programmabsturz :-)
Position von p:	Beim gesuchten oder nächstgrößeren Element

## Aufgabe 4: Speicherallokation

Gegeben sei folgendes Programmstück:

```
1 typedef struct user { int i; } DATA;
2
3 typedef struct _element {
4     struct _element *next;
5     DATA data;
6 } ELEMENT, *EP;
7
8 EP newElement( int val, EP next )
9 {
10     EP new = malloc( sizeof( ELEMENT ) );
11     if ( new )
12     {
13         new->data.i = val; new->next = next;
14     }
15     return new;
16 }
```

Vervollständige das Speicherbild für die folgenden beiden Aufrufe von `newElement()`. Wie bei fast allen Übungen gehen wir davon aus, dass sowohl `int`-Werte als auch Zeiger immer vier Bytes im Arbeitsspeicher belegen. Ferner nehmen wir an, dass der Aufruf der Funktion `newElement()` die Adresse `0xFA00` liefert.

1. `EP p = newElement( 4711, 0 );`

Segment:	Stack		Segment:	Heap
Adresse	Variable	Wert	Adresse	Wert
0xFE7C	EP p:	0xFA00	0xFA04	4711
			0xFA00	0x0000

2. `ELEMENT el; EP p = newElement( 815, &el );`

Segment:	Stack		Segment:	Heap
Adresse	Variable	Wert	Adresse	Wert
0xFE7C	el.data.i:	-----	0xFA04	815
0xFE78	el.next :	-----	0xFA00	0xFE78
0xFE74	EP p :	0xFA00		

Vielen hilft es beim Verstehen, wenn sie zusätzlich noch die Zeiger in die Speicherbildchen einzeichnen.

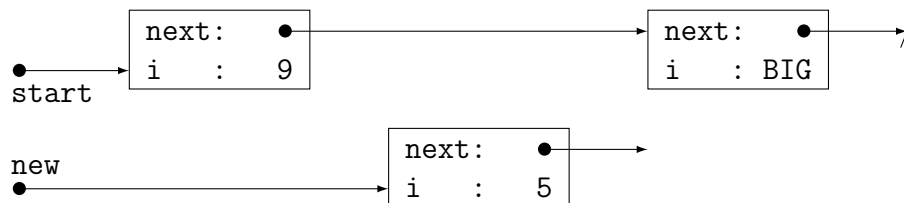
## Aufgabe 5: Elemente einfügen

Für die beiden letzten Quizaufgaben haben wir den Positionierungsalgorithmus aus Aufgabe 3 um ein paar Zeilen erweitert. Der erste Parameter dieser neuen Funktion ist die bisher im Arbeitsspeicher aufgebaute Liste. Der zweite Parameter ist ein Zeiger auf ein neues Element, wie wir es gerade eben in der vorherigen Aufgabe gesehen haben:

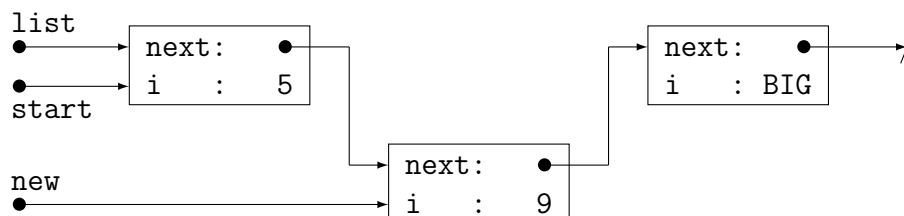
```
1 void insertElement( EP list, EP new )
2     {
3         DATA tmp;
4         while( list->data.i < new->data.i && list->next != 0 )
5             list = list->next;
6         tmp = new->data;
7         *new = *list;
8         list->next = new;
9         list->data = tmp;
10    }
```

In den beiden folgenden Aufgaben wird die Funktion `insertElement()` immer mit der selben Liste aufgerufen. Sie hat nur ein *Datenelement*. Beim ersten Mal wird das Datenelement 5, beim zweiten Mal 13 eingefügt. Der in den Abbildungen verwendete `start`-Zeiger ist der im Hauptprogramm verwaltete Startzeiger der Liste. Er dient nur zur Illustrierung und wird nicht weiter verwendet. Bearbeite nun die beiden Fälle:

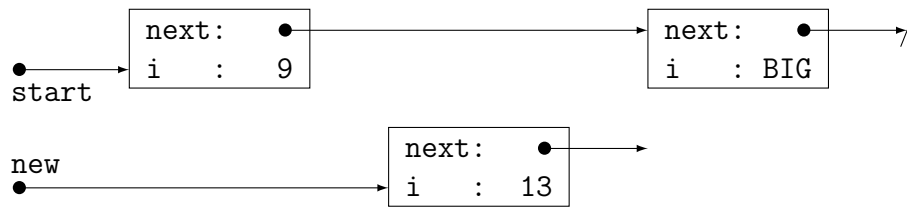
1. Einfügen des „Datenelementes“ 5: Zeile 3:



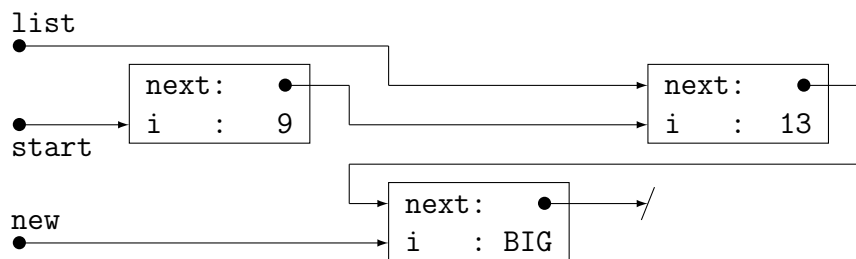
Ende Zeile 9:



2. Einfügen des „Datenelementes\ 13: Zeile 3:



Ende Zeile 9:



Damit wären wir jetzt für den Anwendungsfall gerüstet.

## Teil III: Fehlersuche

---

### Aufgabe 1: Ausgabe von Listen

Die meisten Fehler- und Problemfälle haben wir bereits besprochen. Doch DR. L. IST-WIRTH hat das Ausgeben Wirth'scher Listen noch nicht im Griff. Was lief hier schief...?

```
1 void prtList( EP p )
2     {
3         printf( "Liste:" );
4         do {
5             p = p->next;
6             printf( " %i", p->data.i );
7         } while( p != 0 );
8         printf( "\n" );
9     }
```

Leider erscheinen hier einige Anweisungen in genau der falschen Reihenfolge.

Zeile	Fehler	Erläuterung	Korrektur
5		Hier wird bereits zum nächsten Element weiter gegangen. Das wäre eigentlich nicht so schlimm, doch ist bis hier beim ersten Schleifendurchlauf noch nichts ausgegeben worden. Insofern würde das erste Datenelement verborgen bleiben.	auf später verschieben
7		Die Schleife läuft so lange, wie der Zeiger <i>p</i> kein Null-Zeiger ist. Dadurch wird auch das letzte Element der Liste ausgegeben. In einer Wirth'schen Liste ist dies aber das Dummy-Element und gehört <i>nicht</i> zum Datenbestand und sollte auch nicht ausgegeben werden.	<i>p-&gt;next != 0</i>

Programm mit Korrekturen:

```
1 #include <stdio.h>
2
3 void prtList( EP p )
4     {
5         printf( "Liste:" );
6         for( ; p->next; p = p->next )
7             printf( " %i", p->data.i );
8         printf( "\n" );
9     }
```

## Teil IV: Anwendungen

---

### Aufgabe 1: Definition einer geeigneten Datenstruktur

#### 1. Aufgabenstellung

Definiere eine Datenstruktur für lineare Listen, in der jedes Element ein einzelnes Zeichen aufnehmen kann.

#### 2. Kodierung

In Anlehnung an Übungspaket 31 definieren wir folgende Datenstruktur:

```
1 typedef struct user {
2     char c;
3     } DATA, *DP;           // the user data
4
5 typedef struct _list_element {
6     struct _list_element *next; // the admin part
7     DATA data;                // the user data part
8     } ELEMENT, *EP;
```

### Aufgabe 2: Allokation eines neuen Elementes

#### 1. Aufgabenstellung

Entwickle eine Funktion, die dynamisch ein neues Listenelement generiert.

#### 2. Entwurf

Deklaration: neues Element allozieren: `EP newElement( char c, EP next );`

#### 3. Kodierung

```
1 EP newElement( char c, EP next )
2 {
3     EP p = malloc( sizeof( ELEMENT ) );
4     if ( p )           // do we have a new element ?
5     {
6         p->data.c = c; p->next = next;    // initialize
7     }
8     return p;
9 }
```

# Aufgabe 3: Sortiertes Einfügen eines neuen Elementes

## 1. Aufgabenstellung

Nun wird's ernst. Entwickle eine Funktion, die ein neues Element an der richtigen Stelle in die Liste einfügt. Wir gehen wieder davon aus, dass die Elemente der Liste aufsteigend sortiert werden sollen.

## 2. Entwurf

Vervollständige zunächst die Funktionsdeklaration:

Neues Element einfügen: `void insertElement( EP oldList, EP newElement );`

## 3. Kodierung

```
1 void insertElement( EP oldList, EP newElement )
2     {
3         DATA tmp = newElement->data;
4         while( oldList->next
5             && newElement->data.c > oldList->data.c )
6             oldList = oldList->next;
7         *newElement = *oldList;
8         oldList->next = newElement;
9         oldList->data = tmp;
10    }
```

Um später die Daten leichter einfügen zu können, schreiben wir uns gleich noch eine kleine Hilfsfunktion, die das Ausgeben möglicher Fehlermeldungen kapselt. Damit können wir uns jederzeit ein paar sinnvolle Testausgaben generieren, um beispielsweise den aktuellen Zustand der Liste zu „erfragen“.

```
1 int insertData( EP list, char c )
2     {
3         EP p = newElement( c, 0 );           // a new element
4         if ( p == 0 )                         // ran out of memory ?
5         {
6             fprintf( stderr, "can't insert %c", c );
7             fprintf( stderr, "; no space available\n" );
8         }
9         else insertElement( list, p );        // ok, all clear
10        return p != 0;
11    }
```

Durch den Ausdruck der `return`-Anweisung in Zeile 10 erfährt die aufrufende Stelle, ob die Daten eingefügt werden konnten.



## Aufgabe 4: Ausgabe der Liste

### 1. Aufgabenstellung

Entwickle eine Funktion, die nacheinander alle Elemente einer übergebenen Liste ausgibt.

### 2. Entwurf

Vervollständige zunächst die Funktionsdeklaration:

Liste ausgeben: `void prtList( EP list );`

### 3. Kodierung

Diese Funktion ist sehr einfach. Wir müssen nur alle Elemente außer dem letzten in einer Schleife ausgeben. Das letzte Element erkennen wir daran, dass sein `next`-Zeiger ein Null-Zeiger ist.

```
1 void prtList( EP p )
2     {
3         printf( "Liste:" );
4         for( ; p->next; p = p->next )
5             printf( " %c", p->data.c );
6         printf( "\n" );
7     }
```

## Aufgabe 5: Initialisierung der Liste

### 1. Aufgabenstellung

Jetzt fehlt noch die richtige Initialisierung einer neuen Liste. Im Skript haben wir gesehen, dass der konkrete Datenwert des Dummys unerheblich ist. Wichtig ist hingegen, dass der Next-Zeiger ein Null-Zeiger ist. Dies lässt sich einfach realisieren.

### 2. Kodierung

```
1     EP first = newElement( 'x', 0 ); // dummy element
```

Zur Erinnerung: Bei der Wirth'schen Alternative ist es egal, was für ein Wert im letzten Element steht. Wir haben hier *völlig willkürlich* ein `'x'` gewählt; es hätte auch jeder beliebige andere Wert sein können. Der entscheidende Punkt ist, wie bereits mehrfach gesagt, dass der Next-Zeiger ein Null-Zeiger ist.

## Aufgabe 6: Ein Hauptprogramm zum Testen

Den bereits erlernten `argc/argv` Mechanismus (siehe auch Übungspaket 25) können wir hier sehr gut zum intelligenten Testen unseres Listen-Programms verwenden. Wenn wir auf diesen Mechanismus zurückgreifen, brauchen wir nicht alles fest im Hauptprogramm zu kodieren und können leicht und umfangreich testen.

### 1. Aufgabenstellung

Entwickle ein Hauptprogramm, mit dessen Hilfe wir unsere Listen-Funktionen in geeigneter Art und Weise testen können. Mit Hilfe des `argc/argv`-Mechanismus soll das Hauptprogramm die folgenden Funktionalitäten anbieten:

1. Egal, wie wir unser Programm aufrufen, soll es die Liste direkt nach der Initialisierung sowie vor dem Programmende ausgeben.
2. Das erste Argument (`argv[1]`) soll alle Zeichen enthalten, die wir in die Liste aufnehmen. Beispiel: `./myListe vacation` soll am Ende zu folgender Ausgabe führen: `a a c i n o t v`
3. Sollte noch ein zweites Argument (`argv[2]`) angegeben werden, dann soll die Liste nach jedem Einfügen ausgegeben werden.

Mittels der bisherigen Vorübungen, insbesondere Übungspaket 25, sollte die Umsetzung keine größeren Schwierigkeiten bereiten.

### 2. Kodierung

```
1  int main( int argc, char **argv )
2      {
3          char *p;
4          EP first = newElement( 'x', 0 ); // dummy element
5          prtList( first ); // check after initialization
6          if ( argc > 1 )
7          {
8              for( p = argv[ 1 ]; *p; p++ )
9              {
10                 insertData( first, *p );
11                 if ( argc > 2 ) // debug mode
12                     prtList( first );
13             }
14             prtList( first );
15         }
16         else printf( "sorry, keine Daten vorhanden\n" );
17     }
```

Die nächsten beiden Seiten geben das Programm nochmals vollständig wieder.

### Definition der Datenstrukturen:

```
1  #include <stdio.h>                                // for I/O
2  #include <stdlib.h>                                // for malloc()
3
4  typedef struct user {
5      char c;
6      } DATA, *DP;                                // the user data
7
8  typedef struct _list_element {
9      struct _list_element *next;    // the admin part
10     DATA data;                    // the user data part
11     } ELEMENT, *EP;
```

### Allokation eines neuen Elementes:

```
13 EP newElement( char c, EP next )
14 {
15     EP p = malloc( sizeof( ELEMENT ) );
16     if ( p )                                // do we have a new element ?
17     {
18         p->data.c = c; p->next = next;    // initialize
19     }
20     return p;
21 }
```

### Einfügen eines neuen Elementes:

```
23 void insertElement( EP oldList, EP newElement )
24 {
25     DATA tmp = newElement->data;
26     while( oldList->next
27           && newElement->data.c > oldList->data.c )
28         oldList = oldList->next;
29     *newElement = *oldList;
30     oldList->next = newElement;
31     oldList->data = tmp;
32 }
```

### Einfügen neuer Daten:

```
34 int insertData( EP list, char c )
35 {
36     EP p = newElement( c, 0 );           // a new element
37     if ( p == 0 )                        // ran out of memory ?
38     {
39         fprintf( stderr, "can't insert %c", c );
40         fprintf( stderr, "; no space available\n" );
41     }
42     else insertElement( list, p );        // ok, all clear
43     return p != 0;
44 }
```

### Ausgabe der Liste:

```
46 void prtList( EP p )
47 {
48     printf( "Liste:" );
49     for( ; p->next; p = p->next )
50         printf( " %c", p->data.c );
51     printf( "\n" );
52 }
```

### Das Hauptprogramm:

```
54 int main( int argc, char **argv )
55 {
56     char *p;
57     EP first = newElement( 'x', 0 ); // dummy element
58     prtList( first );               // check after initialization
59     if ( argc > 1 )
60     {
61         for( p = argv[ 1 ]; *p; p++ )
62         {
63             insertData( first, *p );
64             if ( argc > 2 )           // debug mode
65                 prtList( first );
66         }
67         prtList( first );
68     }
69     else printf( "sorry, keine Daten vorhanden\n" );
70 }
```

Endlich fertig :-)

## Nachlese

Die meisten werden froh sein, die Listenverarbeitung halbwegs hinbekommen zu haben. Wenn man sich aber die Lösung mit etwas zeitlichem Abstand nochmals anschaut, wird man die eine oder andere Stelle finden, die nicht ganz so schön geworden ist. Dies betrifft insbesondere die Funktionen `newElement()` und `insertElement()`. Warum? Ganz einfach. Ein Teil dieser Funktionen ist recht generisch, d.h. auch bei anderen Aufgaben einsetzbar. Andererseits sind einige Komponenten genau auf die gestellte Aufgabe zugeschnitten. Also nix da mit Kopieren, wenn man die Algorithmen für die nächste Aufgabe verwenden möchte.

Im Umkehrschluss heißt das: Für die Wiederverwendbarkeit wäre es schön, wenn wir einen aufgabenspezifischen und einen generellen (auch generisch genannten) Teil hätten. Und das ist gar nicht so schwer. Wenn man erst einmal die richtige Position hat (Zeiger `pos` wie im Quiz), kann man das Erzeugen und Einfügen eines neuen Elementes wie folgt realisieren:

```
1 EP newElement( DATA data, EP next )           // generic
2 {
3     EP p = malloc( sizeof( ELEMENT ) );
4     if ( p )                                     // do we have a new element ?
5     {
6         p->data = data; p->next = next;          // initialize
7     }
8     return p;
9 }
10
11 int insertElement( EP pos, DATA data )         // generic
12 {
13     EP new = newElement( pos->data, pos->next );
14     if ( new )
15     {
16         pos->next = new;
17         pos->data = data;
18     }
19     return new != 0;
20 }
```

Für unsere Zeichen-Liste ist das Finden der richtigen Position sehr einfach:

```
1 EP findPosition( EP liste, char c )           // specific: based on c
2 {
3     while( liste->data.c < c && liste->next )
4         liste = liste->next;
5     return liste;
6 }
```

Um ein neues Zeichen in die Liste einzufügen, müssen wir nun alle drei Funktionen in der richtigen Art und Weise verknüpfen. Dabei nutzen wir aus, dass wir das neue Element nicht mit irgendwelchen Daten vorinitialisieren sonder das gefundene Element einfach duplizieren:

```

1  int insertData( EP list, char c )                // specific
2      {
3          DATA data = { c };                      // wrapping into struct
4          if ( ! insertElement( findPosition( list, c ), data ))
5              {
6                  fprintf( stderr, "can't insert %c", c );
7                  fprintf( stderr, "; no space available\n" );
8                  return 0;
9              }
10         return 1;
11     }

```

Und der Rest bleibt unverändert:

```

1  void prtList( EP p )
2      {
3          printf( "Liste:" );
4          for( ; p->next; p = p->next )
5              printf( " %c", p->data.c );
6          printf( "\n" );
7      }
8
9  int main( int argc, char **argv )
10     {
11         char *p;
12         DATA data = { 'x' };
13         EP first = newElement( data, 0 );          // dummy element
14         prtList( first );                          // check after initialization
15         if ( argc > 1 )
16             {
17                 for( p = argv[ 1 ]; *p; p++ )
18                     {
19                         insertData( first, *p );
20                         if ( argc > 2 )              // debug mode
21                             prtList( first );
22                     }
23                 prtList( first );
24             }
25         else printf( "sorry, keine Daten vorhanden\n" );
26     }

```

## Aufgabe 7: Doppelt verkettete Listen

In dieser letzten Übungsaufgabe geht es um doppelt verkettete Listen. Die Bearbeitung ist vor allem theoretischer Natur, weshalb das Implementieren und Testen am Rechner freiwillig sind.

Wohin zeigen die Zeiger bei doppelt verketteten Listen?

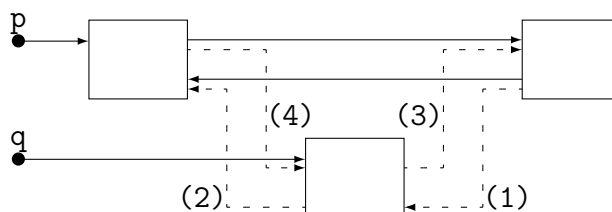
Bei einer doppelt verketteten Liste benötigt man immer zwei Zeiger, einer zeigt zum Nachfolger, der andere zum Vorgänger.

Definiere eine Datenstruktur für ein Element einer doppelt verketteten Liste, die ein `int` und zwei `double` Variablen aufnehmen kann.

```
1 typedef struct user
2     {
3         int i;
4         double d1, d2;
5     } DATA, *DP;
6
7 typedef struct _element
8     {
9         struct _element *previous, *next;
10        DATA data;
11    } D_LIST, *DLP;
```

Skizziere an einem Beispiel, welche Zeiger beim Einfügen eines neuen Elementes in eine bereits vorhandene Liste in welcher Form umgehängt werden müssen.

Die Grafik zeigt zwei Elemente einer doppelt verketteten Liste. Der Zeiger `q` zeigt auf das neue Element, das hinter das Element eingefügt werden soll, auf das `p` zeigt. Die Nummern beziehen sich auf folgendem Programmausschnitt.



```
1 p->next->previous = q;
2 q->previous      = p;
3 q->next          = p->next;
4 p->next          = q;
```

# Übungspaket 33

## Binäre Bäume

---

### Übungsziele:

1. Definition eines binären Baums
2. Das L-K-R Ordnungsprinzip
3. Konstruktion eines binären Baums
4. Traversieren eines binären Baums

### Skript:

Kapitel: 76

### Semester:

Wintersemester 2022/23

### Betreuer:

Benjamin, Thomas und Ralf

### Synopsis:

In diesem Übungspaket beschäftigen wir uns mit Bäumen. Diese wachsen in der Informatik nicht in den Himmel sondern nach unten. Aber das ist eigentlich kein Problem, der Stack wächst ja auch nach unten und daran haben wir alle uns langsam gewöhnt.

Das Hauptproblem mit Bäumen kommt meist aus der Datenstruktur: Jedes Element ist eine Struktur mit mindestens zwei Zeigern, die wieder auf andere Elemente des selben Typs zeigen. Das bereitet oftmals Kopfzerbrechen. Aber hat man sich daran erst einmal gewöhnt, findet man den Rest sehr einfach und Bäume plötzlich cool. . .



# Teil I: Stoffwiederholung

---

## Aufgabe 1: Allgemeines zu Bäumen

Erkläre mit eigenen Worten, was man in der Informatik unter einem *Baum* versteht:

Ein Baum besteht aus mehreren Elementen, die man auch Knoten nennt. In diesen Knoten sind die eigentlichen Daten abgespeichert. Jeder Knoten kann Nachfolger haben. Bei einem *binären* Baum sind dies zwei, bei einem *tertiären* Baum drei usw. Die einzelnen Knoten sind durch Pfeile miteinander verbunden. Diejenigen Knoten, die keine weiteren Nachfolger haben, nennt man wie in der Natur auch Blätter. Denjenigen Knoten, der keinen Vorgänger hat, nennt man in Anlehnung an die Natur auch Wurzel. Wie in der Informatik üblich, kann ein Baum auch leer sein, sodass er aus keinem einzigen Knoten besteht.

Erkläre mit eigenen Worten, was man unter dem L-K-R Ordnungsprinzip versteht:

Wie oben schon ausgeführt speichern die einzelnen Knoten die Daten des Nutzers. Ferner wissen wir, dass jeder Knoten zwei oder mehr Nachfolger haben kann. Damit man beim *Suchen* der Daten nicht alle Knoten anschauen muss, werden Bäume meist sortiert. Bei einem L-K-R Baum gilt, dass alle Knoten des linken Teilbaums kleinere Werte als der betrachtete Knoten haben und der rechte Teilbaum nur größere. Dadurch weiß man beim Suchen immer, ob man ggf. links oder rechts weitersuchen muss.

## Aufgabe 2: Detailfragen zu Bäumen

In welche Richtung wachsen Informatik-Bäume?	Von oben nach unten
Wie sind die Knoten miteinander verbunden?	Mit Pfeilen
Wie viele Nachfolger hat ein binärer Baum?	2 (zwei) (two) (due)
Wie heißen die beiden Teile üblicherweise?	linker und rechter Teilbaum
Wie heißen die „untersten“ Knoten?	Blätter
Wie heißt der „oberste“ Knoten?	Wurzel
Wie druckt man Bäume am elegantesten?	Rekursiv
Wie sucht man in Bäumen am elegantesten?	Rekursiv
Wie fügt man Knoten am elegantesten hinzu?	Rekursiv (ist aber nicht einfach)
Welches Übungspaket sollte ich unbedingt können?	<b>22:</b> Rekursion

## Teil II: Quiz

### Aufgabe 1: Konstruktion eines Baumes

Nehmen wir an, wir haben einen Baum zum Ablegen von `int`-Zahlen, der L-K-R sortiert sein soll. Nehmen wir ferner an, wir haben bereits einen Knoten in dem sich die Zahl 22 befindet. Wie sieht der resultierende Baum aus, wenn wir der Reihe nach die folgenden Zahlen in den Baum einfügen: 1 6 4 -1 5 50 10 30 12 60?

Der resultierende Baum sieht in ASCII Art wie folgt aus (Nullzeiger sind nicht illustriert):

```

                                tree --+
                                    v
                                -----
          +-----+-----+ ( 22 ) -----+-----+
          v               v               v
        -----
    +-----+ ( 1 ) -----+          +-----+ ( 50 ) -----+
    v       ---          v              v              v
  -----
( -1 )    +-----+ ( 6 ) -----+    ( 30 )          ( 60 )
-----    v       ---          v              v
          -----
          ( 4 ) -+      ( 10 ) -+
          ---  v      -----  v
              ---      -----
                ( 5 )      ( 12 )
                  ---      -----
```

### Aufgabe 2: Drucken eines Baumes

Nehmen wir ferner an, wir haben für das Drucken eines Baumes folgenden Algorithmus:

Funktion zum Drucken eines Baums

Parameter: Pointer to Node: tree

wenn tree  $\neq$  Null-Zeiger

dann druckeBaum( tree dereferenziert rechts )

Ausgabe tree dereferenziert Zahl

druckeBaum( tree dereferenziert links )

Aus welchen Zahlen besteht die Ausgabe? 

60	50	30	22	12	10	6	5	4	1	-1
----	----	----	----	----	----	---	---	---	---	----

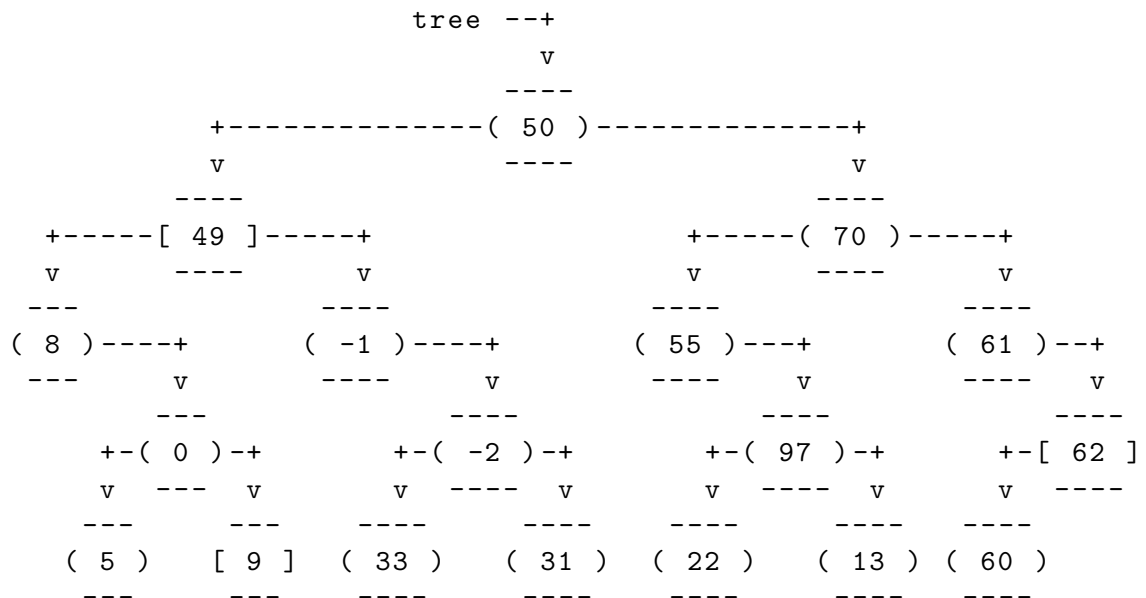
## Teil III: Fehlersuche

---

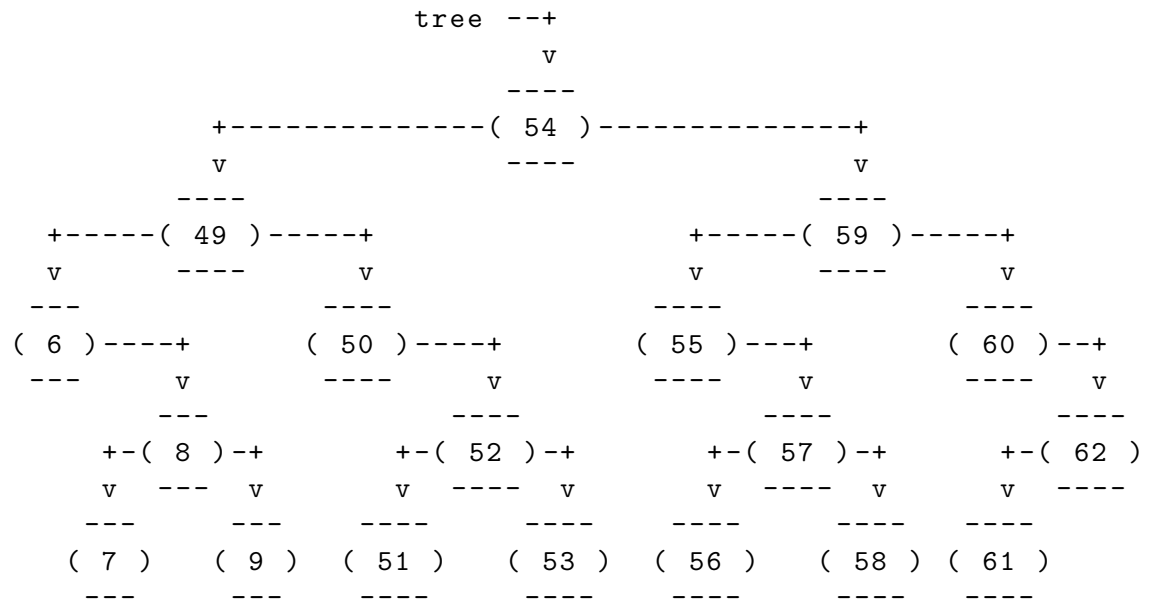
### Aufgabe 1: L-K-R Bäume

DR. BiB ELKAER hat versucht, einen binären Baum mit einer L-K-R Sortierung zu erstellen. Ein erster Blick verrät: das hat nicht ganz geklappt. Wie bei einem bekannten Japanischen Zahlenrätsel sei gesagt, dass diejenigen Zahlen, die in eckigen Klammern stehen, mit Sicherheit richtig sind. Alle anderen Knoten enthalten falsche Zahlen, die aus dem Baum entfernt werden müssen. Ferner sei verraten, dass die kleinste Zahl des Baums 6 ist. Versucht den Baum so zu korrigieren, dass es sich tatsächlich um eine L-K-R Sortierung handelt und obige Randbedingungen nicht verletzt sind. Viel Spass :-))

**Ein fehlerhafter L-K-R Baum:**



**Der korrigierte Baum:** (eine weitere, korrekte Lösung gibt es nicht)



# Teil IV: Anwendungen

---

## Aufgabe 1: Binärer Baum am einfachsten Beispiel

### 1. Aufgabenstellung

Ziel dieser Aufgabe ist die Entwicklung eines einfachen binären Baums, der in seinen Knoten je ein `int` und ein `char`-Wert aufnehmen kann. Die Sortierung soll dabei nach der `int`-Komponente erfolgen.

### 2. Testdaten

Bei der Wahl der Testdaten greifen wir auf die obige Quizaufgabe zurück, da wir uns ja dort schon Gedanken über die richtige Ausgabe gemacht haben:

Datensätze: (22, 'a'), ( 1, 'b'), ( 6, 'c'), ( 4, 'd'), (-1, 'e')  
( 5, 'f'), (50, 'g'), (10, 'h'), (30, 'i'), (12, 'j')  
(60, 'k')

### 3. Entwurf

Definiere zunächst die Struktur des Knotens, die wir hier mit `NODE` bezeichnen, sowie einen Zeiger auf selbige, der den Namen `NP` trägt:

```
1 typedef struct user {
2     int i;
3     char c;
4 } DATA, *DP;           // the user data
5
6 typedef struct _node {
7     struct _node *left, *right; // the admin part
8     DATA data;                // the user data part
9 } NODE, *NP;
```

Vervollständige folgende Funktionsdeklarationen:

Neuen Knoten generieren: `NP newNode( int i, char c );`

Baum drucken: `void prtTree( NP tree );`

Knoten einfügen: `void insertNode( NP *tree, NP newNode );`

### 4. Kodierung

Unser Programm befindet sich auf den nächsten beiden Seiten.

### Datenstrukturen und Einzeiler:

```
1  #include <stdio.h>                                // for I/O
2  #include <stdlib.h>                                // for malloc()
3
4  typedef struct user {
5      int i;
6      char c;
7      } DATA, *DP;                                // the user data
8
9  typedef struct _node {
10     struct _node *left, *right;    // the admin part
11     DATA data;                    // the user data part
12     } NODE, *NP;
13
14 #define newTree()      0           // new tree: null pointer
15 #define emptyTree(tp)  (tp == 0)  // test on empty tree
```

### Knoten erzeugen:

```
17 NP newNode( int i, char c )
18 {
19     NP p = malloc( sizeof( NODE ) );
20     if ( p )                                // do we have a new node?
21     {
22         p->data.i = i; p->data.c = c; // initialize data
23         p->left = p->right = 0;      // initialize admin
24     }
25     return p;
26 }
```

### Baum drucken:

```
28 void prtTree( NP tree )
29 {
30     if ( ! emptyTree(tree))    // do we have a tree ?
31     {
32         prtTree( tree->left );    // print left
33         printf( "i=%d c=%c\n",
34             tree->data.i, tree->data.c );
35         prtTree( tree->right );    // print right
36     }
37 }
```

### Knoten einfügen:

```
39 void insertNode( NP *tree, NP newNode )
40     {
41         NP thisNode;
42         if ( ! emptyTree(*tree))           // ok or zero ?
43         {
44             thisNode = *tree;           // to make it easier
45             if ( newNode->data.i < thisNode->data.i )
46                 insertNode( & thisNode->left, newNode );
47             else insertNode( & thisNode->right, newNode );
48         }
49         else *tree = newNode; // replace the null pointer
50     }
```

### Hilfsfunktion zum leichten Einfügen: (if-Abfrage und ggf. Fehlermeldungen):

```
52 int insertData( NP *tree, int i, char c )
53     {
54         NP p = newNode( i, c );           // getting a new node
55         if ( p == 0 )                     // ran out of space ?
56         {
57             fprintf( stderr, "can't insert (%d %c)", i, c );
58             fprintf( stderr, "; no space available\n" );
59         }
60         else insertNode( tree, p ); // ok, all clear :-)
61         return p != 0; // so, the caller knows the result
62     }
```

### Hilfsfunktion und Hauptprogramm:

```
64 int main( int argc, char **argv ){
65     NP myTree = newTree();           // initialize
66     insertData( & myTree, 22, 'a' );
67     insertData( & myTree, 1, 'b' );
68     insertData( & myTree, 6, 'c' );
69     insertData( & myTree, 4, 'd' );
70     insertData( & myTree, -1, 'e' );
71     insertData( & myTree, 5, 'f' );
72     insertData( & myTree, 50, 'g' );
73     insertData( & myTree, 10, 'h' );
74     insertData( & myTree, 30, 'i' );
75     insertData( & myTree, 12, 'j' );
76     insertData( & myTree, 60, 'k' );
77     prtTree( myTree );           // print the entire tree
78     return 0;                     // done
79 }
```

# Anhänge





# Anhang I

## Anleitung zu den Übungspaketen

In den letzten Jahren sind immer wieder allgemeine Fragen zur „richtigen“ Bearbeitung der Übungspakete aufgetreten. Diese Fragen betrafen insbesondere die Auswahl der geeigneten Übungsaufgaben sowie die konkrete Form der Ausführung. In der Konsequenz haben diese Fragen des Öfteren zu einer „Lähmung“ der eigenen Aktivitäten geführt. Aus diesem Grund gibt es hier ein paar allgemeine Hinweise.

### Worum geht es in den Übungspaketen?

Jedes einzelne Übungspaket und insbesondere jede einzelne Übungsaufgabe behandelt ausgewählte Elemente der C-Programmiersprache. Diese Elemente werden an einfachen Aufgaben eingeübt. Diese Aufgaben sind so gestaltet, dass das Anwendungsproblem einfach ist und nach Möglichkeit einen Bezug zu späteren Problemstellungen hat.

Im Laufe der Bearbeitung der einzelnen Übungsaufgaben kommen immer wieder verschiedene „Programmiersmuster“ und „Lösungsstrategien“ zur Anwendung. Um diese Programmiersmuster und Lösungsstrategien zu verinnerlichen, müssen diese aber eigenständig erlebt werden. Bildlich gesprochen, erlernt man das Autofahren auch nicht einen Tag vor der Fahrprüfung aus dem Golf GTI Handbuch.

### Was ist das Ziel der Übungspakete?

Das erste, offensichtliche Ziel der Übungspakete ist natürlich die erfolgreiche Bearbeitung der Aufgaben. Aber das ist nur eines der Ziele. Das zweite wichtige Ziel besteht im Finden eines Lösungswegs! Daher ist dieser oft auch nicht vorgegeben.

### Welche Arbeitsweise wird vom Ingenieur erwartet?

Aufgrund der gemachten Erfahrungen sind es viele Studenten gewohnt, genaue Arbeitsanweisungen zu bekommen, die einfach Schritt für Schritt abzuarbeiten sind. Diese Herangehensweise widerspricht aber in einigen Punkten *grundlegend* den späteren Anforderungen an einen Ingenieur. Von einem Ingenieur wird in der Regel erwartet, dass er eine ihm gestellte Aufgabe *eigenständig* löst.

Später wird und kann der Kunde dem Ingenieur nicht sagen, wie er das Problem zu lösen hat; dann könnte er es nämlich auch selbst lösen und bräuchte den Ingenieur nicht. Vielmehr bekommt der Ingenieur eine Aufgabenstellung und eine Art Anforderungsprofil in dem

Sinne: „Was soll dabei herauskommen?“ Der Ingenieur muss sich selbst überlegen, wie er die gesteckten Ziele erreicht.

Um sich an diese Anforderung zu gewöhnen, sind auch die Übungspakete entsprechend gestaltet. In den Übungsaufgaben befinden sich keine Anleitungen, die einem genau sagen, was man einzutippen hat, ob man überhaupt tippen muss und dergleichen.

Somit besteht die erste Aufgabe darin, sich selber zu überlegen, was eigentlich das Ziel der Aufgabenstellung ist!

### Wie lerne ich das Meiste?

Das ist eine gute Frage. Die Übungsaufgaben, bzw. deren Lösungen, werden von uns nicht eingesammelt und somit auch nicht benotet. Vielmehr geht es darum, dass ihr etwas lernt!

Aber wie lernt man am besten? Hierfür eignet sich insbesondere die folgende Standardaufgabe: „Wie ist die Standardfunktion `strlen( char *p )` implementiert?“ Folgende Ansätze sind offensichtlich:

**Ansatz 1 :** Ich nehme mir ein Buch und schaue nach.

**Lernerfolg:** Close to zero!

**Ansatz 2 :** Ich führe mir die Musterlösungen zu Gemüte.

**Lernerfolg:** Close to zero!

**Ansatz 3 :** Ich versuche die Aufgabe mit Papier und Bleistift selber zu lösen.

**Lernerfolg:** Here we go!

**Ansatz 4 :** Wie Ansatz 3 aber mit zusätzlicher Implementierung auf meinem  
: Rechner.

**Lernerfolg:** Even better.

**Ansatz 5 :** Wie Ansatz 4 aber zusätzlich vergleiche ich meine Lösung mit der  
: Musterlösung.

**Lernerfolg:** Yes, that's it!

### Was wird von mir erwartet?

Für die Lehrveranstaltung bekommt man sechs ECTS Punkte. Dies bedeutet einen normierten studentischen Arbeitsaufwand von 180 Stunden. Das Semester hat 14 Wochen, so dass durch Vorlesungen und Übungen etwa  $14 \times 6 = 84$  Stunden „verbraucht“ sind. Abzüglich Prüfungsvorbereitung etc. bleiben pro Woche weitere  $96/14 = 7$  Stunden für die eigenständige Arbeit übrig. Daraus ergeben sich von unserer Seite folgende Erwartungen:

1. 7 (in Worten „sieben“) Stunden selbstständige Beschäftigung mit den Übungsaufgaben und dem Skript *zu Hause* oder einem anderen geeigneten Arbeitsort.

2. Vorbereiten der Übungsaufgaben zu Hause! Viele fangen erst an, sich mit den Übungsaufgaben zu beschäftigen, wenn sie vor Ort am Rechner sitzen. Das ist zu spät, denn die Zeit vergeht zu schnell. Nutzt die Zeit in den Übungen, um Fragen zu stellen!
3. Wir erwarten, dass ihr die ersten drei Teile jedes Übungspakets bereits zu Hause weitestgehend bearbeitet habt.
4. Da euch anfänglich noch die Übung fehlt, solltet ihr die Aufgaben nicht gleich am Rechner lösen sondern erst auf einem Zettel entwerfen und mittels Papier und Bleistift simulieren. Erst dann lohnt sich in der Regel eine Implementierung auf dem Rechner.

### **Aufgabenpensum zu groß?**

Vielen erscheint der Umfang der Übungsaufgaben zu groß. Ist das ein wirkliches Problem? Nein! Alle Aufgaben sind optional (siehe aber auch den nächsten Abschnitt!). Empfehlenswert ist immer, die ersten Teilaufgaben zu lösen. Da sich dann die Struktur der Aufgaben wiederholt, können bei Zeitknappheit weitere Aufgaben weggelassen werden. *Aber*, das weitere Bearbeiten der Übungsaufgaben kann nur helfen! Der auf der Webseite angegebene Zeitplan dient nur der Orientierung. Dies ist mit häuslicher Vor- und Nachbereitung zu schaffen. Aber auch wer das Pensum nicht schafft, sollte an den Übungstunden teilnehmen und ist dort gern gesehen.

### **Gruppenarbeit**

Wir empfehlen, dass ihr in kleinen Arbeitsgruppen bestehend aus zwei oder drei Leuten zusammen arbeitet. Dies hilft beim Lernen und fördert obendrein die Sozialkompetenz.

### **Was ist der Lohn?**

Die Mühen zahlen sich in vielfältiger Weise unmittelbar aus:

1. Nach *erfolgreicher* Bearbeitung der Übungsaufgaben ist der Aufwand für die Prüfungsvorbereitung fast vernachlässigbar.
2. Mit dem Erlernten kann man seine weiteren Studienaufgaben deutlich vereinfachen.
3. Man lernt selbstständig Arbeiten, was die Handlungskompetenz als Ingenieur und damit die Berufsaussichten deutlich verbessert.

# Anhang II

## Präcedenztabelle

Operator	Beschreibung	Assoziativität
() [] . -> ++ --	runde Klammern (Gruppierung) eckige Klammern (Array Indizes) Komponentenzugriff über Variablenname Komponentenzugriff über Zeiger Post Inkrement/Dekrement	von links nach rechts
++ -- + - ! ~ (type) * & sizeof	Pre Inkrement/Dekrement Vorzeichen Logisch Negation/1er Komplement Cast Zeiger-Dereferenzierung Adressoperator Größe in Bytes	von rechts nach links
* / %	Multiplikation/Division/Modulo	von links nach rechts
+ -	Addition/Subtraktion	von links nach rechts
<< >>	Bit-Shift links/rechts	von links nach rechts
< <= > >=	Relation Kleiner/Kleiner-Gleich Relation Größer/Größer-Gleich	von links nach rechts
== !=	Relation Gleich/Ungleich	von links nach rechts
&	Bitweises „UND“	von links nach rechts
^	Bitweises „XOR“	von links nach rechts
	Bitweises „ODER“	von links nach rechts
&&	Logisches „UND“	von links nach rechts
	Logisches „ODER“	von links nach rechts
?:	Dreiwertige Bedingung	von links nach rechts
= += -= *= /= %= &= ^=  = <<= >>=	Zuweisung Addition/Subtraktion-Zuweisung Multiplikation/Division-Zuweisung Modulo/bitweises „UND“ Zuweisung Bitweises „XOR“/„ODER“ Zuweisung Bit-Shift links/rechts Zuweisung	von rechts nach links
,	Komma, Ausdrucks-Liste	von links nach rechts

# Anhang III

## Literaturverzeichnis

- [1] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1990.
- [2] Brian W. Kernighan und Dennis Ritchie. *The C Programming Language*. Prentice Hall Software, 1978.
- [3] Reinhold Kimm, Wilfried Koch, Werner Simonsmeier, Friedrich Tontsch. *Einführung in Software Engineering*. Walter de Gruyter Verlag, 1979.
- [4] Niklaus Wirth. *Algorithmen und Datenstrukturen*. Vieweg+Teubner Verlag, 1991.