Grundlagen der C-Programmierung **SS 2009**
Dr. Henrik Brosenne Georg-August Universität Göttingen
Zentrum für Informatik

Aufgabensammlung

Vorwort

Die Sammlung besteht aus drei Teilen. Der Teil „Vorbereitungsaufgaben“ enthält Aufgaben, die zur Verbesserung des Verständnisses der allerersten Vorlesung dienen. Da die Schritte detailliert vorgegeben sind, sind nur Zugang zu einem (Linux-)Rechner und Grundkenntnisse im Umgang damit nötig. Der Teil „Standardaufgaben“ enthält Aufgaben, die in der Reihenfolge direkt auf die Kapitel des Vorlesungsskripts abgestimmt sind und die wichtigsten der dort behandelten Aspekte von C abdecken. Die Übungen zum Kurs konzentrieren sich meist auf die Lösung dieser Aufgaben (es werden trotzdem vielleicht nicht alle dort genannten Aufgaben behandelt). Der Teil „Ergänzungsaufgaben“ enthält weitere Aufgaben zu dem Vorlesungsstoff, von denen auch einige in den Übungen besprochen werden. Im Verlauf des Kurses wird jeweils vorgegeben, welche Aufgaben zu behandeln sind. Alle Aufgaben sind auch als Anregung zu eigenen Programmierübungen zu sehen und beliebig ausbaubar.
[Bei manchen Aufgaben wird in eckigen Klammern darauf hingewiesen, welche Sprachkonstrukte, Aspekte der Programmiersprache etc. erforderlich bzw. geeignet sind, um die Aufgabe zu lösen.]

Vorbereitungsaufgaben

V1 Öffnen Sie zur Eingabe von Kommandos ein Terminalfenster (auch Konsole genannt). Erzeugen Sie dort mit Hilfe des Kommandos `mkdir vorbereitungsaufgaben` ein neues Unterverzeichnis und wechseln Sie dorthin mittels `cd vorbereitungsaufgaben`. Lassen Sie sich mittels `ls` den Inhalt des Verzeichnisses auflisten. Das Verzeichnis sollte anfangs leer sein.

V2 Rufen Sie (immer noch in `vorbereitungsaufgaben`) einen Editor auf durch Eingabe des entsprechenden Kommandos. Schreiben Sie damit den folgenden Programmcode ab und speichern ihn in einer Datei `hallo.c`.

```
1 /* Das allererste Programm */
2 #include <stdio.h>
3
4 int main( void)
5 {
6     printf( "Hallo␣Leute!\n");
7     return 0;
8 }
```

V3 Compilieren Sie das Programm, indem Sie in der Konsole `gcc hallo.c` aufrufen (Sollte es Fehlermeldungen geben, so haben Sie sich beim Abschreiben vertippt. Korrigieren Sie in diesem Fall.). Lassen Sie sich mittels `ls -l` den Inhalt des Verzeichnisses in langer Form auflisten. Sie sollten eine ausführbare Datei `a.out` vorfinden. Dies ist das kompilierte *Binärprogramm*, zu dem in `hallo.c` gespeicherten *Quelltext*. Führen Sie das Binärprogramm aus, indem Sie einfach `./a.out` eingeben.

V4 Löschen Sie das Wort `int` in dem Quelltext, speichern und compilieren Sie das Programm erneut. Die Auslassung sollte den Compiler bei Aufruf mit `gcc hallo.c` nicht stören — er ergänzt das Wort selbständig. Solche Auslassungen können in anderen Situationen aber auch unabsichtlich entstehen und zu inhaltlichen Fehlern führen. Deswegen betrachten wir diesen Quelltext streng genommen als nicht korrekt. Durch entsprechende Optionen beim Aufruf kann der Compiler manche derartige Unkorrektheiten entdecken und Warnungen ausgeben. Zur Demonstration compilieren Sie jetzt das Programm mit dem Aufruf `gcc -Wall hallo.c`.

Es gibt weitere Aufrufoptionen, die helfen, Nachlässigkeiten und Fehler zu vermeiden, die aber nicht alle bei diesem kurzen Programm demonstriert werden können. Gewöhnen Sie sich an, Ihre C-Programme bei Benutzung des `gcc` mindestens mit den Optionen `-ansi`, `-pedantic` und `-Wall` zu compilieren, also z.B.

```
gcc -Wall -pedantic -ansi hallo.c
```

Es ist nicht wichtig, in welcher Reihenfolge diese Optionen erwähnt werden.

V5 Der Name `a.out` für das ausführbare Programm ist nicht besonders aussagekräftig. Fügen Sie dem Aufruf des `gcc` noch die Option `-o hallo` hinzu, um anstelle von `a.out` ein ausführbares Programm `hallo` zu erhalten.

V6 Löschen Sie die Datei `hallo.c` mittels `rm hallo.c` und beenden Sie auch den Editor. Lernen Sie nun den obigen Quelltext auswendig. Das ist ernst gemeint — es wird Ihnen für das Weitere helfen! Lernen Sie auch den Compileraufruf in der langen Form auswendig. Wenn Sie fertig sind, führen Sie alle Schritte noch einmal ohne diese Vorlage aus.

Standardaufgaben

S1 [Fehlermeldungen des Compilers]

Speichern Sie den nachfolgenden Quelltext in einer Datei `alphabet.c`. Compilieren Sie den Quelltext mit dem Aufruf

```
gcc -Wall -ansi -pedantic -o alphabet alphabet.c
```

und rufen Sie anschließend das Programm auf.

```c
/* Dieses Programm gibt die ersten 6 Grossbuchstaben des Alphabets
   aus, einen Buchstaben pro Zeile. */

#include <stdio.h>

int main(void)
{
    printf("A\n");
    printf("B\n");
    printf("C\n");
    printf("D\n");
    printf("E\n");
    printf("F\n");

    return 0;
}
```

Nehmen Sie nun nacheinander die untenstehenden Änderungen am Quelltext vor und versuchen Sie erneut zu compilieren. Der Compiler wird manchmal Fehler finden und auch eine Zeilnnummer dazu in der Quelltextdatei angeben (Dies ist nicht immer genau die Zeile, die den Fehler verursacht hat, sondern sehr oft eine Position weiter hinten im Quelltext!) Lesen Sie die jeweilige Fehlermeldung und versuchen Sie sie zu verstehen. Fragen Sie gegebenenfalls einen Betreuer oder einen fortgeschrittenen Teilnehmer. Für späteres Arbeiten ist es nützlich, sich eine Liste der Fehlermeldungen anzulegen mit stichpunktartigen Hinweisen auf die Fehlerursache.

Wichtig!: Nehmen Sie jede Änderung zurück, bevor Sie die nächste vornehmen!

- Löschen Sie die Zeile `#include <stdio.h>`
- Löschen Sie `*/` am Ende des ersten Kommentars.
- Löschen Sie `int` vor `main`.
- Löschen Sie irgendein Semikolon im Quelltext.
- Schreiben Sie `print` anstatt `printf`.
- Schreiben Sie `return O` (Buchstabe „Oh“) anstelle `return 0`.
- Löschen Sie die Zeile `return 0;`
- Schreiben Sie `retörn` anstelle `return`.

S2 [`printf`, Escape-Sequenzen zur Ausgabe besonderer Zeichen]

Schreiben Sie ein Programm, das die Wirkung der Escapesequenzen

`\b` `\n` `\r` `\t` `\"` `\\`

als Argumente der Funktion `printf` verdeutlicht.

S3 [Formatierte Ausgabe von Zahlwerten]

Lernen Sie folgenden Quelltext auswendig. Zur Kontrolle speichern Sie ihn in einer Datei, die Sie anschließend compilieren. Falls es Fehlermeldungen gibt, so korrigieren Sie, ohne auf die Vorlage zu sehen. Anschließend führen Sie das compilierte Programm aus.

```c
/* Ein paar hilfreiche Zahlen werden ausgegeben */
#include <stdio.h>

int main( void)
{
  printf( "Aller␣guten␣Dinge␣sind␣%d.\n", 3);
  printf( "Pi␣ist␣ungefaehr␣%f.\n", 3.141593);

  return 0;
}
```

S4 [Ausgabe von Zahlwerten mit `printf`, Standard-Konstanten]

Schreiben Sie ein Programm, das die Werte der Konstanten

`CHAR_MIN` `CHAR_MAX` `INT_MIN` `INT_MAX`
`FLT_MIN` `FLT_MAX` `DBL_MIN` `DBL_MAX`

ausgibt (Standard–Headerdateien `limits.h` bzw. `float.h`).

S5 [Ausgabe von Zeichen, `char`-Variablen, `while`]

Schreiben Sie ein Programm, das alle Zeichen ausgibt, die „größer“ oder gleich 'A' und „kleiner“ oder gleich 'z' sind.

S6 [`while`, Eingabe-Ende, Zahldarstellungen]

Es sollen wiederholt (bis zur Eingabe von `C-d`) nichtnegative Dezimalzahlen eingelesen und die Werte jeweils im Oktal– und Hexadezimalsystem ausgegeben werden

S7 [`while`, `scanf`, ganzzahlige Division, `%`-Operator]

Schreiben Sie ein Programm, das eine nichtnegative ganze Zahl einliest, ihre Quersumme iterativ berechnet und ausgibt.

S8 [`while`, `scanf`, einfache Berechnungen]

Schreiben Sie ein Programm, das eine beliebige Anzahl von Gleitkommazahlen einliest und ihren Mittelwert ausgibt. Die Zahlen müssen dazu nicht alle gleichzeitig gespeichert werden, sondern können nacheinander eingelesen und bearbeitet werden.

S9 [Felder, Sortieralgorithmus für Zahlen]

Es soll eine feste Anzahl ganzer Zahlen eingelesen und in einem Feld abgespeichert werden. Dann soll der Inhalt des Feldes ausgegeben, das Feld sortiert, und der Inhalt des Feldes erneut ausgegeben werden.

Realisieren Sie „Sortieren durch direkte Auswahl“: Wenn n Zahlen zu sortieren sind, bestimmt man zunächst das kleinste Element und vertauscht es mit dem, das an Position 1 steht. Dann bestimmt man von den Elementen an den Positionen $2, \ldots, n$ das kleinste Element und vertauscht es mit dem an Position 2, usw.

S10 [Arbeiten mit Strings]

Schreiben Sie ein Programm, das zwei Zeilen als Strings einliest und den zweiten String an den ersten String anhängt. Die eingelesenen Strings sollen jeweils maximal 50 Zeichen (einschließlich abschließendem Nullbyte) lang sein. Ist dieses nicht der Fall, soll das Programm mit einer entsprechenden Fehlermeldung abgebrochen werden.

S11 [`getchar`, `if`]

Schreiben Sie ein Programm, das eine Binärzahl einliest und den entsprechenden Dezimalwert ausgibt. Dabei sollen führende „white spaces“ überlesen und fehlerhafte Eingaben zuückgewiesen werden (z.B. Eingabe von Buchstaben oder Überschreitung des Wertebereichs).

S12 [Felder, ASCII-Zeichensatz]

Es soll ein Text von der Standardeingabe eingelesen und bestimmt werden, wie oft die einzelnen ASCII–Zeichen in ihm vorkommen.

S13 [Mehrdimensionale Felder, benannte Konstanten]

Schreiben Sie ein Programm, das zwei Matrizen liest, ihr Produkt berechnet und ausgibt.

Dimensionieren Sie die Matrizen mit benannten Konstanten. Das Programm kann dadurch (ohne weiteres) keine größeren Matrizen verarbeiten, aber es soll beliebige kleinere Matrizen verarbeiten können. Vom Benutzer sollte daher vorher das tatsächlich zu verarbeitende Format erfragt werden.

S14 [Felder, `while`, `if`, `%`-Operator]

Zu bestimmen sind alle Primzahlen kleiner als 1000. Zwei Verfahren:

a) Durch Division wird geprüft, ob die Zahlen n ($1 < n < 1000$) einen Teiler k ($1 < k < n$) besitzen. [Das kann man etwas abkürzen: Es genügt einen Teiler zu finden, der kleiner oder gleich $\sqrt{n}$ ist. In Aufgabe E12 finden Sie einen Hinweis zur Benutzung der entsprechenden Standardfunktion `sqrt` für die Quadratwurzel.]

b) Beim „Sieb des Eratosthenes“ (Eratosthenes von Kyrene, griechischer Mathematiker um 225 v. Chr.) werden zunächst alle zu untersuchenden Zahlen (hier 2 bis 999) aufgeschrieben. Jeder Schritt des eigentlichen Algorithmus besteht aus drei Einzelschritten:
   - Die erste nicht weggestrichene Zahl wird gesucht.
   - Diese Zahl wird als Primzahl notiert.
   - Ihre Vielfachen werden weggestrichen.

Realisieren Sie beide Algorithmen möglichst effizient.

S15 [Rekursion, `%`-Operator]

Schreiben Sie eine rekursive Funktion

```
unsigned long ggt (unsigned long a, unsigned long b);
```

die den größten gemeinsamen Teiler der nichtnegativen ganzen Zahlen a und b berechnet. Verwenden Sie die Formel

$$\mathrm{ggt}(a,b) = \begin{cases} \mathrm{ggt}(b, a \bmod b) & \text{falls} \quad b > 0 \\ a & \text{falls} \quad b = 0 \end{cases}$$

S16 [Rekursion, `%`-Operator]

Schreiben Sie eine *rekursive* Funktion zur Berechnung der Quersumme einer nichtnegativen ganzen Zahl (vgl. Aufgabe S7).

S17 [`%`-Operator, logische Operatoren]

Zu einer eingegebenen Jahreszahl soll bestimmt werden, ob es sich um ein Schaltjahr handelt oder nicht. Dies soll als `int`-Funktion realisiert werden, die den Wert `0` liefert, falls das Argument *keine* Schaltjahreszahl ist und einen Wert ungleich `0` sonst.

Der Gregorianische Kalender legt fest, dass jedes Jahr mit durch 4 teilbarer Jahreszahl ein Schaltjahr ist, außer wenn die Jahreszahl durch 100 teilbar ist. In diesem Fall handelt es sich nur dann um ein Schaltjehr, wenn die Jahreszahl auch durch 400 teilbar ist.

S18 [Backtracking, mehrdimensionale Felder]

Schreiben Sie ein Programm, das für ein (rechteckiges) Labyrinth bei vorzugebendem Startpunkt einen Weg (alle Wege, den kürzesten Weg, ...) zu einem Ausgang findet, wobei als Ausgang alle Punkte auf dem Rand des Labyrinths zählen.

Die Form des Labyrinths soll vom Benutzer erfragt werden; seine Eingabe könnte etwa in der Form

```
XXXXXXXXXX
X   X    X
X*XXX XX X
X     X  X
XXXXXXX XX
```

erfolgen, wobei `*` den gewünschten Startpunkt bezeichnet. Überlegen Sie sich selbst eine geeignete Form der Ausgabe.

S19 [mehrdimensionale Felder, Backtracking]

Gegeben sei ein $(n \times n)$–Spielfeld. Ein Springer, der nach den Schachregeln bewegt werden kann, wird auf das Feld mit den Koordinaten (x, y) gesetzt.

Schreiben Sie ein Programm, das alle Wege des Springers findet, die von (x, y) aus genau einmal über jedes der n^2 Felder führen. Überlegen Sie sich vorab: Bestimmte Konfigurationen von Feldgröße und Startpunkt **können keine** Lösung besitzen!

Bauen Sie für den Test ein, daß das Programm abgebrochen wird, sobald der erste Weg gefunden wurde. Wenn Sie diesen „Notausgang“ ausbauen, sollten Sie berücksichtigen:

- Bei einem (5×5)–Spielfeld gibt es je nach Startpunkt zwischen 56 und 304 Lösungen (sofern es überhaupt Lösungen gibt).
- Bei einem (6×6)–Spielfeld gibt es je nach Startpunkt zwischen ca. 50,000 und 500,000 Lösungen.
- Bei einem (7×7)–Spielfeld dürfte es für jeden Startpunkt weit über 100 Mio. Lösungen geben (sofern es überhaupt Lösungen gibt).

Lassen Sie sich vom Programm die benötigte Rechenzeit melden.

S20 [Felder mit `const`-Inhalt]

Schreiben Sie Funktionen, die ein Datum in den Tag im Jahr (z.B. der 11.2. ist der 42. Tag im Jahr) bzw. einen Tag im Jahr in ein Datum umrechnen und schreiben Sie auch eine entsprechende Testumgebung dazu.

S21 [Felder mit `const`-Inhalt, `%`-Operator]

Lassen Sie bei Aufgabe S20 auch „exotische“ Eingaben zu wie in diesen Beispielen: Der 366. Tag im Jahr 1991 ist der 1.1.1992; der 0. Tag im Jahr 1992 ist der 31.12.1991; usw. Verwenden Sie dazu Ihre Lösung von Aufgabe S17 wieder.

S22 [Modularisierung]

Fassen Sie Funktionen aus den Aufgaben S17 und S20 zu einem Modul `datumsfunktionen` zusammen, der aus der Implementationsdatei `datumsfunktionen.c` und der Header-Datei `datumsfunktionen.h` besteht.

Nutzen Sie die Funktionen für ein Programm zur Lösung dieser oder ähnlicher Aufgaben: Welches Datum hat der 100. Tag vor oder nach einem einzugebenden Datum?

S23 [Stringfelder, `%`-Operator, Modularisierung]

Schreiben Sie ein Programm, das zu einem einzulesenden Datum den Wochentag berechnet. Dazu sollten Sie Ihre Lösung von Aufgabe S22 benutzen können.

Hinweise:

- Das Datum soll in der Form `tt.mm.jjjj` eingegeben werden.
- Der Wochentag soll in Klarschrift gemeldet werden (keine Kennziffer!).
- Unzulässige Daten (z.B. 30.2.1992) müssen zurückgewiesen werden.
- Bei uns gilt der „Gregorianische Kalender“, der durch Papst Gregor XIII. am 15. 10.1582 unserer Zeitrechnug eingeführt wurde. In ihm ist ein Jahr ein Schaltjahr, wenn seine Jahreszahl entweder durch 4 und nicht durch 100 oder durch 400 teilbar ist.
- Die Schaltjahrsregel des Gregorianischen Kalenders ist nicht ganz exakt: In circa 3000 Jahren wird eine zusätzliche Korrektur erforderlich. Das Programm sollte deshalb neben Daten vor dem 15.10.1582 auch Daten aus dem Jahr 5000 oder später zurückweisen.
- Der 1.1.1600 war ein Sonnabend.

Verwenden Sie ggf. Modul-weit definierte Variablen (z.B. eine Tabelle der Monatslängen). Achten Sie besonders auf die Strukturierung des Programms!

Sie können sich die Programmierung etwas erleichtern, wenn Sie zunächst darüber nachdenken, wie man diese Aussage nachweisen kann: „In dem Aberglauben, daß man an einem Freitag den 13. besonders häufig Pech hat, steckt ein Körnchen Wahrheit – es gibt nämlich keinen Tag, der häufiger ist als Freitag der 13.“ (Wenn Sie wollen, können Sie die Lösung natürlich auch programmieren.)

S24 [Arbeit mit Strings]

Realisieren Sie die folgenden Funktionen:

- `int strend (const char *s, const char *t)` liefert 1, wenn der String `t` am Ende des String `s` steht, und 0 sonst.
- `char *strrchr (const char *s, int c)` liefert den Zeiger auf das letzte Vorkommen des Zeichens `c` im String `s` bzw. den `NULL`–Zeiger, wenn `c` in `s` nicht vorkommt.
- `char *strstr (const char *s, const char *t)` liefert den Zeiger auf das erste Vorkommen des String `t` im String `s` bzw. den `NULL`–Zeiger, wenn `t` in `s` nicht vorkommt.

Testen Sie die Funktionen in einem Rahmenprogramm.

S25 [Modularisierung, dynamische Speicherverwaltung]

Schreiben Sie einen Modul (Headerdatei und Implementation) zur dynamischen Bereitstellung (und Freigabe) von Speicher für Vektoren und Matrizen. Realisieren Sie in einem weiteren Modul die Matrizenmultiplikation als Funktion.

Testen Sie beide Module mit einem geeigneten Rahmenprogramm.

S26 [Strukturen, Modularisierung]

Definieren Sie den Typ `BRUCH` als Struktur mit zwei `int`–Komponenten `zaehler` und `nenner`. Realisieren Sie für diesen Typ die „Standardoperationen“

- Eingabe eines Bruches
- Ausgabe eines Bruches
- Addition, Subtraktion, ... zweier Brüche
- Berechnung des Quotienten (Funktionswert `float`!)
- Erweitern um einen (als Parameter zu übergebenden) Faktor
- Kürzen

als Funktionen in einem Modul mit Headerdatei.

Testen Sie die Funktionen mit einem geeigneten Rahmenprogramm.

Ergänzungsaufgaben

E1 [Formatierte Ausgabe von Zahlwerten]

Speichern Sie den folgenden Quelltext in einer Datei, die Sie anschließend compilieren und ausführen. Vergleichen Sie die Formatbeschreiber mit dem Ausgabeergebnis. Merken Sie sich die Systematik und experimentieren Sie ein wenig damit herum.

```
1  /* Zahlen werden mit vorgegebener Stellenzahl ausgegeben */
2  #include <stdio.h>
3
4  int main( void)
5  {
6    printf( "Aller␣guten␣Dinge␣sind␣%d.\n", 3);
7    printf( "Aller␣guten␣Dinge␣sind␣%3d.\n", 3); /* mind. 3 Stellen */
8    printf( "Aller␣guten␣Dinge␣sind␣%6d.\n", 3); /* mind. 6 Stellen */
9
10   printf( "Pi␣ist␣etwa␣%f.\n", 3.141593);
11
12   /* mind. 6 Stellen, davon mind. 3 nach dem Punkt  */
13   printf( "Pi␣ist␣etwa␣%6.3f.\n", 3.141593);
14
15   /* mind. 9 Stellen nach dem . */
16   printf( "Pi␣ist␣etwa␣%.9f.\n", 3.141593);
17
18   return 0;
19 }
```

E2 [Schleifen, Formatierte Ausgabe von Zahlwerten, benannte Konstanten]

Lassen Sie eine Multiplikationstafel ausgeben, etwa so:

```
 * |   1   2   3   4   5
-------------------------
 1 |   1   2   3   4   5
 2 |   2   4   6   8  10
 3 |   3   6   9  12  15
 4 |   4   8  12  16  20
 5 |   5  10  15  20  25
```

Verwenden Sie eine benannte Konstanten für die Größe der Multiplikationstafel, so dass das Programm einfach auf andere Größen angepasst werden kann.

E3 [Escape-Sequenzen]

Schreiben Sie ein Programm, das folgende Texte ausgibt:

```
Er kam lässig heran und sagte nur "Na, wie geht's?".
Kommentare beginnen mit /* und enden mit */. Verwechseln Sie
das bitte nicht mit \* bzw. *\!
```

E4 [`printf`, `char`, `for`, `ctype.h`, Funktion `isprint`]

Rufen Sie in der Konsole die online-Hilfe zum ASCII-Zeichensatz auf: `man ascii`. Schreiben Sie ein Programm, das die dort angegebene Tabelle erzeugt.

E5 [Felder, `for`, `scanf`, benannte Konstanten, `while`, `EOF`]

Schreiben Sie ein Programm, dass die Koeffizienten eines Polynoms $p(x)$ mit vorgegebenem maximalen Grads einliest und anschließend bis zum Eingabeende Stellen a (Gleitkommawerte) anfordert und den Wert $p(a)$ des Polynoms an dieser Stelle berechnet.

Verwenden Sie zur Berechnung des Werts das Hornerschema. Beispiel:

$$p(x) = 5.1 * x^3 - 1.8 * x^2 - 0.02 * x + 17.3 = 17.3 + x * (-0.02 + x * (-1.8 + 5.1 * x)).$$

E6 [`scanf`, Rekursion, `%`-Operator]

Rufen Sie in der Konsole den Befehl `factor 144` auf. Die Ausgabe `144: 2 2 2 2 3 3` gibt die Primzahlzerlegung von 144 an. Programmieren Sie dies in C nach, um beliebige ganze Zahlen bis zur Größe `UINT_MAX` verarbeiten zu können. Die zu verarbeitende Zahl soll mit `scanf` eingelesen werden.

E7 [`getc`, `putchar`, Escapesequenzen]

Textdateien sehen unter UNIX und DOS/Windows unterschiedlich aus. Während in UNIX-Textdateien eine Zeile lediglich durch ein „Linefeed“-Zeichen `\n` (oder Newline) beendet wird, steht in DOS-Textdateien dort die Zeichenfolge `\r\n`. Schreiben Sie zwei Programme, die die Umformungen von-UNIX-nach-DOS bzw. von-DOS-zu-UNIX vornehmen. Die Programme sollen jeweils zeichenweise von der Standardeingabe lesen und auf die Standardausgabe ausgeben. Das erste Programm fügt vor jedem gelesenen `\n` einfach ein `\r` ein und reicht die anderen Zeichen unverändert weiter. Das zweite Programm „verschluckt“ einfach die `\r`-Zeichen, die nach einem `\n` auftreten.

E8 [Felder, `for`-Schleife, `while`-Schleife]

Ein Permutation der Länge 9 ist eine Abbildung der Menge $\{1, 2, \ldots, 9\}$ auf sich selbst. Permutationen kann man durch zweireihige Matrizen angeben. Die folgende Permutation zum Beispiel bildet 1 auf 3 ab, 2 auf 4, 3 auf 9 usw.:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 4 & 9 & 1 & 7 & 5 & 2 & 8 & 6 \end{pmatrix}$$

Da die Einträge der oberen Reihe immer die gleichen sind, würde es auch genügen, nur die untere Reihe 3, 4, 9, 1, 7, 5, 2, 8, 6 anzugeben.

Schreiben Sie ein Programm, das Folgendes leistet: Der Nutzer wird aufgefordert, eine Permutation der Zahlen 1 bis 9 einzugeben (durch Angabe der unteren Reihe, wie oben beschrieben). Es wird geprüft, ob die Eingabe korrekt war (d.h., ob jede der Zahlen 1 bis 9 genau einmal angegeben wurde). Bei Korrektheit wird die Permutation in einem Feld gespeichert (ansonsten wird eine erneute Eingabe verlangt) und zur Kontrolle wie eine zweireihige Matrix ausgegeben.

E9 [`rand`, `srand`]

Schreiben Sie eine `int`-Funktion `muenze`, die bei jedem Aufruf (pseudo)zufällig entweder 0 oder 1 als Funktionswert liefert. Testen Sie mit dieser Funktion, ob die „Münze“ tatsächlich wie erhofft etwa ebenso häufig den Wert 1 liefert, wie den Wert 0.

Testen Sie auch, wie oft Ergebniskombinationen wie `00`, `01`, etc. entstehen.

E10 [`rand`, `srand`]

Schreiben Sie eine `int`-Funktion `gezinkterwuerfel`, die bei jedem Aufruf (pseudo)zufällig einen ganzzahligen Wert zwischen 1 und 6 liefert. Dabei soll die 6 etwa doppelt häufig auftauchen, wie jeder andere Wert. Testen Sie dies durch entsprechend häufiges „würfeln“ mit dieser Funktion.

E11 [Felder, `rand`, `srand`, Schleifen]

Schreiben Sie ein Programm, das pseudozufällig eine Permutation der Zahlen 1 bis 9 erzeugt. Gehen Sie dabei wie folgt vor: Die Permutation wird wie in Aufgabe E8 durch ein `int`-Feld der Länge 9 beschrieben. Anfangs ist in jeder Position noch jeder der Werte 1 bis 9 erlaubt (denn keiner ist bereits vergeben und damit „verboten“ für weitere Positionen). Beginnend bei der Feldkomponente mit Index 0 wird nun 8 mal hintereinander ein noch nicht verbotener Wert pseudozufällig gewählt und in der nächsten freien Feldkomponente gespeichert. Die „Menge der noch erlaubten Werte“ wird bei jeder Iteration aktualisiert, so dass nach der 8ten Iteration nur noch ein einziger Wert möglich ist, der in der letzten Komponente gespeichert ist.

E12 [`math.h`, `sqrt`]

Lernen Sie den folgenden Programmtext auswendig:

```
1 /* Quadratwurzelberechnung */
2 #include <stdio.h>
3 #include <math.h>
4
5 int main( void)
6 {
7   printf( "Die␣Wurzel␣aus␣%f␣ist␣etwa␣%f\n", 2.0f, sqrt( 2.0f));
8   return 0;
9 }
```

Schreiben Sie dann ohne Vorlage das Programm in eine Datei namens `wurzel.c`, compilieren sie mit dem Aufruf

```
gcc -Wall -ansi -pedantic -o wurzel wurzel.c -lm
```

und führen das Programm aus.

Lernen Sie auch den Compile-Aufruf auswendig. Die Option `-lm` am Ende ist für den Linker bestimmt. Er wird damit angewiesen, die Bibliothek der mathematischen Funktionen zu verwenden.

E13 [`switch`, `if`, (für Ergänzung: Stringfelder)]

Olympische Spiele der Neuzeit finden in Jahren statt, deren Jahreszahl gerade ist und zwar Sommerspiele, falls die Jahreszahl durch 4 teilbar ist und Winterspiele sonst. Schreiben Sie ein Programm, das den Nutzer nach einer Jahreszahl fragt und die Information ausgibt, ob Olympische Spiele in diesem Jahr stattfinden und ob es sich um Winterspiele oder Sommerspiele handelt.

Die oben genannte Regelung gilt erst seit einigen Jahren. Finden Sie im Internet heraus, seit wann. Finden Sie auch heraus, seit wann überhaupt Olympische Spiele der Neuzeit veranstaltet werden und in welchen Jahren es Pausen gab. Bauen Sie dann das Programm aus, in dem Sie zusätzlich die aktuelle Jahreszahl ermitteln lassen (entweder durch Anfrage an den Nutzer oder durch Benutzen einer Bibliotheksfunktion) und die Antwort zeitlich und grammatisch korrekt geben, wie etwa: „Im Jahre 2002 fanden olympische Winterspiele statt.“ oder „Im Jahre 2008 werden olympische Sommerspiele stattfinden.“

Sie können auch eine deLuxe-Variante erstellen, indem Sie die bekannten Austragungsorte in einem Feld konstanter Strings ablegen und die Ausgabe damit ergänzen.

E14 [`math.h`, `sqrt`, `scanf`]

Schreiben Sie ein C-Programm, das vom Nutzer zunächst die Eingabe einer Zahl verlangt, dann intern die Quadratwurzel berechnet und den Nutzer dann die Quadratwurzel „erraten“ lässt: Gibt der Nutzer irgendeinen Wert ein, so reagiert das Programm so lange wahrheitsgemäß mit „Größer! Nächster Versuch:“ oder „Kleiner! Nächster Versuch:“ bzw. „Richtig!!!!“ auf die Eingabe, bis genügende Übereinstimmung mit dem vom Rechner ermittelten Wert erzielt wurde. Die Umschreibung „Genügende Übereinstimmung“ soll andeuten, dass auch der Rechner die Wurzel nur näherungsweise ermitteln kann.

E15 [`while`, Newton-Iteration, benannte Konstanten, ganzzahlige Division]

Schreiben Sie ein C-Programm, das vom Nutzer die Eingabe einer Zahl a verlangt und eine Näherung b für deren Quadratwurzel wie unten beschrieben durch Newton-Iteration berechnet. Prüfen Sie die Güte des Ergebnisses, indem Sie b^2 und a bei verschiedenen Größenordnungen von a vergleichen und auch die Anzahl der Iterationen ausgeben. Geben Sie die Genauigkeit $\varepsilon > 0$ durch eine benannte Konstante vor.

Die Idee des Newton-Verfahrens ist folgende: Die Quadratwurzel von a ist eine Nullstelle der Funktion $f(x) = a - x^2$. Ausgehend vom Startwert $x_0 = 1$ werden iterativ $x_1, x_2, \ldots, x_n$ berechnet, bis $|f(x_n)|$ kleiner als ε ist. Dabei wird x_{i+1} aus x_i bestimmt als Schnittpunkt der x-Achse mit der Tangente an den Graphen von $f(x)$ im Punkt x_i. Der Schnittpunkt kann mit Hilfe der Ableitung von $f(x)$ bestimmt werden. Im unserem Fall ergibt sich die folgende Iterationsvorschrift:

$$x_{i+1} = \frac{1}{2} * (a/x_i + x_i).$$

Treffen Sie Vorkehrungen, damit bei der Umsetzung obiger Vorschrift nicht ganzzahlig dividiert wird! Felder sind für die Lösung dieser Aufgabe nicht erforderlich.

E16 [`sizeof`]

Benutzen Sie den `sizeof`-Operator, um zu ermitteln, wieviel Bytes für Werte der Ganzzahltypen reserviert werden. Ermitteln Sie dann rechnerisch (z.B. mittels `kcalc`) die größten darstellbaren Werte der einzelnen Ganzzahltypen und vergleichen Sie diese mit den Werten aus Aufgabe S4.

Hinweis. Zur Speicherung von Werten des Typs `char` wird ein Byte verwendet.

E17 [`fgetc`, `fputc`, Kommandozeilenargumente]

Unter UNIX bewirkt der Aufruf `cp` *`dateiname1 dateiname2`*, dass versucht wird, den Inhalt der ersten Datei in der zweiten abzulegen. Falls die Anzahl der Kommandozeilenargumente nicht stimmt, so wird der Nutzer darauf hingewiesen und das Programm beendet. Das Gleiche passiert, falls die erste Datei nicht zum Lesen oder die zweite nicht zum Schreiben geöffnet werden kann.

Programmieren Sie dieses Verhalten nach.

E18 [`fgetc`, `stdlib.h`, `atoi`]

Verändern Sie Ihr Programm aus Aufgabe E6 dahingehend, dass die zu verarbeitende Zahl jetzt per Kommandozeilenargument als String übergeben wird und mittels `atoi` als Ganzzahl interpretiert wird. Die Dokumentation zu dieser Funktion erhalten Sie durch Benutzung der online-Hilfe `man 3 atoi`. Die Funktion `atoi` überprüft zwar nicht ohne weiteres, ob ein Ganzzahlüberlauf stattfindet (und auch andere Probleme bei der Eingabe bleiben unbehandelt)— Sie brauchen sich in Ihrer Lösung aber nicht darum kümmern.

E19 [`fgetc`, `switch`, `ctype.h`, `isspace`, Kommandozeilenargumente]

Unter UNIX bewirkt der Aufruf `wc` *`dateiname`*, dass in der angegeben Datei die Anzahl der Zeichen, der Wörter und der Zeilen gezählt und ausgegeben wird. Was ein Zeichen ist, sollte klar sein. Ein „Wort“ ist eine beliebige Folge von „non whitespace-Zeichen“, deren Anfang entweder durch den Dateianfang oder eine Whitespace-Zeichen begrenzt wird und deren Ende entweder durch ein Whitespace-Zeichen oder das Dateiende begrenzt wird — die Anzahl der Wörter lässt sich so aus der Anzahl der Übergänge whitespace-non whitespace ermitteln. Eine Zeile ist schließlich eine Folge von Zeichen, deren letztes entweder das Newline-Zeichen (`char`-Wert `'\n'`) oder das Dateiende ist.

Probieren Sie dies an einigen Testdateien aus und programmieren Sie dann das Verhalten nach.

E20 [`fgetc`, `fputc`, Kommandozeilenargumente, Strings]

Schreiben Sie Ihre Programme von Aufgabe E7 um, so dass es ein Kommandozeilenargument als Dateinamen interpretiert, den Inhalt der angesprochenen Datei in die gewünschte Form umwandelt und das Ergebnis in einer neuen Datei abspeichert. Über Fehler (z.B. nicht lesbare Datei) soll der Nutzer informiert werden.

E21 [Felder als Funktionsargumente]

Programmieren und testen Sie eine Funktion `int permtest( const unsigned int* p, int laenge)`, die von einem übergebenen Feld der Länge `laenge` testet, ob der Inhalt eine Permutation repräsentiert (Vgl. Aufgabe E8) und in diesem Fall einen Wert ungleich 0 zurückliefert und sonst 0.

E22 [mehrdimensionale Felder als Funktionsargumente]

Eine Permutationsmatrix ist eine quadratische, ganzzahlige Matrix, die in jeder Zeile und jeder Spalte genau einen Eintrag 1 und sonst nur 0-Einträge hat. Programmieren und testen Sie eine Funktion `int permatcheck(const int** p, int laenge)`, die von einer durch ein zweidimensionales `int`-Feld der Größe `laenge`×`laenge` testet, ob es sich um eine Permutationsmatrix handelt.

E23 [mehrdimensionale Felder als Funktionsargumente]

Eine *Sudoku-Matrix* ist eine 9×9-Matrix, deren Einträge ganze Zahlen zwischen 1 und 9 sind und die weitere Bedingungen erfüllen. Zur Formulierung der Bedingungen betrachten wir die Matrix als Blockmatrix aus 9 Blöcken von 3×3-Matrizen, die durch die Zeilen bzw. Spalten 1 bis 3, 4 bis 6 und 7 bis 9 gebildet werden. Jede dieser 3×3-Matrizen bezeichnen im weiteren als *Block*. Nun können wir die Bedingungen formulieren, die aus einer Matrix obiger Form eine Sudoku-Matrix machen:

- Jede Zeile und jede Spalte gibt eine Permutation der Länge 9 an.
- Jeder Block enthält jeden der Werte 1 bis 9 genau einmal.

Programmieren und testen Sie eine Funktion `int sudokucheck(const int** s)`, die von einer durch ein zweidimensionales `int`-Feld der Größe 9×9 testet, ob es sich um eine Sudoku-Matrix handelt.

E24 [`rand`, `srand`, mehrdimensionale Felder, Felder als Funktionsargumente]

Schreiben Sie ein Programm, dass pseudozufällig eine Sudoku-Matrix generiert. Dazu gibt es z.B. diese Möglichkeiten:

- Sie speichern ein festes Sudoku (z.B. aus der Zeitung) in einem zweidimensionalen Feld und wenden eine zufällige Permutation der Länge 9 darauf an.
- Lassen Sie zeilenweise den Inhalt des Sudokus zufällig wählen. Bei der ersten Zeile besteht noch völlig freie Wahl, bei jeder weiteren bis zur achten Zeile sind die Ausschlussbedingungen zu berücksichtigen, die sich aus den bereits gewählten Zeilen ergeben. Die letzte Zeile ist dann durch die noch fehlenden Spalteneinträge gegeben.

Zwecks möglicher Wiederverwendung sollte die Aufgabe am besten durch eine Funktion `sudokugenerate` realisiert werden, die einen Zeiger auf den Anfang eines `int`-Feldes der Größe 9×9 zurückliefert, das die generierte Sudoku-Matrix speichert.

E25 [`rand`, `srand`, mehrdimensionale Felder, `printf`]

Ein Sudoku ist ein Rätsel, das darin besteht, in einer unvollständig gegebenen Sudoku-Matrix die fehlenden Einträge zu ermitteln.

Schreiben Sie ein Programm, das Sudoku-Rätsel generiert. Benutzen Sie zunächst die Funktion `sudokugenerate` aus Aufgabe E24 um eine Sudoku-Matrix zufällig generieren zu lassen. Dann lassen Sie die Matrix in geeigneter Form (mit Rahmen, damit man die Blöcke erkennt etc.) ausgeben, wobei aber bestimmte Einträge durch Leerzeichen ersetzt werden. Welche Einträge ausgegeben werden, können Sie durch ein „Besetzungsmuster“ steuern. Dies soll hier eine beliebige 9×9-Matrix sein. Ein Eintrag der Sudoku-Matrix wird nur dann ausgegeben, wenn der entsprechende Eintrag des Besetzungsmusters verschieden von 0 ist. Passende Besetzungsmuster können Sie z.B. aus in Zeitungen abgedruckten Sudokus ermitteln. Sie können aber auch versuchen, Besetzungsmuster generieren zu lassen, indem mehrere Permutationsmatrizen (siehe Aufgabe E22) generiert werden und deren Summe als Besetzungsmuster verwendet wird.

E26 [mehrdimensionale Felder, Rekursion, Backtracking]

Schreiben Sie eine Funktion `sudokusolve`, die eine unvollständige Sudoku-Matrix (fehlende Einträge durch Wert 0 angegeben) als Argument erhält und versucht, diese zu vervollständigen. Der Funktionswert sollte anzeigen, ob der Versuch erfolgreich war oder nicht.

E27 [`malloc`, `free`, Strings]

Schreiben Sie ein Programm, dass eine natürliche Zahl z und eine weitere natürliche Zahl b zwischen 2 und 36 dezimal einliest und die b-adische Darstellung von z ausgibt. Für die Ziffern der b-adischen Darstellung sollen die Zeichen `0`, `1`, ..., `9`, `a`, `b`, ..., `z` in dieser Reihenfolge benutzt werden. Ermitteln Sie zunächst, welche Länge die b-adische Darstellung von z hat. Dann allokieren Sie Speicher für einen String entsprechender Länge, das die Ziffern aufnimmt, der String soll dann ausgegeben werden. Anschließend geben Sie den Speicherplatz für den String wieder frei.

E28 [`malloc`, Strings, Datei-Ein- und Ausgabe]

Schreiben Sie ein Programm, das den Inhalt einer Datei in einen String einliest. Dazu wird die Datei zunächst zum Lesen geöffnet und die Zeichen werden gezählt. Obwohl es auch eleganter geht, wird nun wie folgt verfahren: Die Datei wird wieder geschlossen. Dann wird Speicher für den String allokiert, die Datei erneut zum Lesen geöffnet und die einzelnen Zeichen in den String übertragen. Anschließend wird die Datei geschlossen, der String zur Kontrolle ausgegeben und der Speicher darauf wieder frei gegeben.