Universität Kassel
Fachbereich 16
Prof. Dr. B. Sick
Fachgebiet Intelligente Eingebettete Systeme

UNIKASSEL VERSITÄT
Intelligent Embedded Systems

Labor C/Embedded Systems
Teil 1 – Einführung in C
Übungsblatt 2

Hinweise: Denken Sie daran, bei Problemen das Skript und das in der Vorlesung genutzte Buch zu konsultieren. Fragen Sie bei den Tutoren, falls Sie nicht weiter kommen. Auch Wikipedia und andere Quellen liefern wertvolle Informationen.

Anmerkung: In diesem Übungsblatt geht es um Techniken, die Sie zum Programmieren in C zwingend können *müssen*. Nehmen Sie sich bitte Zeit zur Bearbeitung und „spielen Sie mit dem Computer"!

Überlegen Sie sich auch eigene Programmierideen!

Aufgabe 1: Datentypen, Casts

a) Runden Sie eine Zahl. Nutzen Sie dafür jedoch keine bereits vorgegebene C-Funktion aus der `math.h`! Nutzen Sie Cast-Operationen dazu. Angenommen, Sie haben irgend eine reelle Zahl eingelesen und Sie wollen diese runden, können Sie auf diese Zahl einfach den Summanden $0,5$ addieren. Nach einem Cast auf eine ganze Zahl wird Ihnen das korrekt gerundete Ergebnis geliefert. Prüfen Sie das nach!

b) Bitte lesen und verstehen Sie das folgende Listing:

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

int main (void) {
  float ergebnis;
  int dividend = 5;
  int divisor = 2;

  ergebnis = dividend / divisor;

  printf("%4.2f\n", ergebnis);

  return 0;
}
```

Sie erkennen, dass hier eine Division ($\frac{5}{2}$) durchgeführt und das Ergebnis in einer `float`-Variable gespeichert werden soll. Sobald Sie das Programm jedoch ausführen, erhalten Sie ein unerwartetes Ergebnis. Korrigieren Sie den Quelltext entsprechend, und zwar *ausschließlich durch die Verwendung von Casts*.

Aufgabe 2: Bitoperationen

a) Implementieren Sie ein Programm, in welchem Sie zwei `char`-Variablen einlesen. Diese sind bekanntlich jeweils 8 Bit groß. Bitte speichern Sie *beide* Zeichen nun in einer *einzigen* `short`-Variable (die bekanntlich 16 Bit groß ist). Nutzen Sie Bitoperationen, um beide `char`-Variablen in der `short`-Variable „unterzubringen“. Geben Sie dann die beiden eingelesenen Zeichen wieder aus – und zwar nutzen Sie zur Ausgabe *nicht* die beiden Variablen, in die Sie die Zeichen eingelesen haben, sondern wieder die `short`-Variable. Nutzen Sie also auch zur Ausgabe wieder Bitoperationen.

b) In der letzten Übung sollten Sie ein Programm implementieren, welches die größtmögliche positive Zahl, die in einer `int`-Variablen aufgenommen werden kann, bestimmt. Einen Lösungsvorschlag finden Sie im folgenden Listing. Ändern Sie den Quelltext derart ab, dass Sie vor der Ausgabe der größtmöglichen Zahl kein Dekrement mehr nutzen. Nutzen Sie eine Bitoperation (Tipp: Tilde!). Warum klappt das?

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

int main (void) {
  int n = 123;

  while (n > 0) {
    n++;
  }

  n--;

  printf("Die groesstmoegliche Zahl in einer int-Variable ist: %d\n",
      n);

  return 0;
}
```

c) Implementieren Sie ein Programm, in welchem Sie eine Zahl einlesen. Diese soll als Exponent im Ausdruck 2^n dienen. Ihr Programm soll diesen Ausdruck möglichst effizient berechnen. Nutzen Sie Bitoperationen!

d) Lesen Sie eine Ganzzahl ein. Stellen Sie mittels einer Bitoperation fest, ob es sich um eine positive oder eine negative Zahl handelt.

e) Lesen Sie eine Ganzzahl ein. Stellen Sie mittels einer Bitoperation fest, ob es sich um eine gerade oder ungerade Zahl handelt.

g) Schreiben Sie eine Funktion, der Sie als Parameter zwei Zahlen (beides `int`-Variablen) übergeben. Beide Zahlen sollen in der Funktion binär verundet werden (wenn Sie also die Zahlen a und b übergeben, soll die Operation $a \wedge b$ ausgeführt werden). Geben Sie das Ergebnis der Berechnung zurück. Machen Sie das für alle Ihnen bekannten binären Operationen. Auch für die einwertigen, d. h. auch für das bitweise Negieren.

h) Schreiben Sie eine Funktion, der Sie als Parameter zwei Zahlen (beides `int`-Variablen) übergeben. Sie sollen nun prüfen, ob in der binären Darstellung der Zahl, die Sie als erstes übergeben, an der Stelle, die durch die zweite Zahl bezeichnet ist, ein Bit gesetzt ist. Unterrichten Sie den Benutzer per Konsolenausgabe, ob an der angegebenen Stelle ein Bit gesetzt ist. Geben Sie aus der Funktion außerdem den Wert `1` zurück, falls das Bit gesetzt ist und geben Sie `0` zurück, falls das Bit in der übergebenen Zahl an der angegebenen Stelle nicht gesetzt ist.

i) Schreiben Sie eine Funktion, der Sie als Parameter drei Zahlen (alles `int`-Variablen) übergeben. Sie sollen nun in der binären Darstellung der Zahl, die Sie als erstes übergeben, an der Stelle, die durch die zweite Zahl bezeichnet ist, das Bit auf `0` oder `1` setzen, wobei Sie `0` oder `1` als dritten Parameter Ihrer Funktion übergeben. Geben Sie die Zahl, in der Sie ein Bit auf `0` oder `1` gesetzt haben zurück.

j) Spielen Sie mit `shift.c` (moodle), verstehen Sie es!

k) Implementieren Sie ein Programm, das eine *positive Ganzzahl* in dezimaler Darstellung in eine Zahl in binärer Darstellung umrechnet. Recherchieren Sie nochmals die Begriffe *MSB* (Most Significant Bit) und *LSB* (Least Significant Bit) und stellen Sie die von Ihnen berechnete Binärzahl so dar, dass das MSB links steht. Orientieren Sie sich an `shift.c` (moodle). Machen Sie sich vor allem klar, warum zuerst eine 1 in der Bitmaske nach links geshiftet wird (und warum das clever ist).

Aufgabe 3: Funktionen und Parameter

a) Das folgende Listing enthält eine Funktion, die eine per Parameter übergebene Zahl negieren und die negierte Zahl als Rückgabewert zurückliefern soll. Ändern Sie die Funktion derart, dass das geforderte Verhalten erreicht wird. Achten Sie außerdem darauf, auch einen Wert (der einen korrekten *Typ* haben muss) zurück zu geben! Tipp: Sie dürfen Änderungen nur in der `negiere()`-Funktion durchführen. Ändern Sie die Signatur und den Funktionsrumpf.

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

void negiere() {
  //eine Zahl negiert man so: negierteZahl = zahl * -1;
}

int main() {
  int zahl = 42;
  printf("Die Zahl %d ist negiert: %d", zahl, negiere(zahl));
  return 0;
}
```

b) Implementieren Sie in einem Programm eine Funktion mit der Signatur `void teilbareZahlen(int n, int m)`. Diese Funktion soll alle durch n teilbaren Zahlen von 0 bis einschließlich m ausgeben. Rufen Sie diese Funktion aus der `main()`-Funktion heraus auf.

c) Lesen Sie zwei Zahlen ein. Diese sollen die Seitenlängen eines Rechtecks repräsentieren. Berechnen Sie die Fläche und geben Sie die Fläche aus. Lagern Sie die Flächenberechnung in eine Funktion aus, erledigen Sie die Ausgabe jedoch in der aufrufenden Funktion!

d) Implementieren Sie Aufgabe d nochmals. Lagern Sie jedoch zusätzlich die Aufforderung zum Eingeben zweier Zahlen und die entsprechende Eingabefunktionalität in eine weitere Funktion, zusätzliche zur Flächenberechnungsfunktion, aus.

Aufgabe 4: Felder

a) Legen Sie ein `int`-Array an. Belegen Sie es schon bei der Initialisierung mit den Werten von 0 bis 9. Das Array soll dann also zehn Elemente enthalten. Beachten Sie, dass Sie die Größe bei der direkten Initialisierung nicht mit angeben brauchen.

b) Erweitern Sie Aufgabe a derart, dass Sie das Array Wert für Wert in einer Schleife ausgeben.

c) Ein Array kann im eindimensionalen Fall als Vektor verstanden werden (und im mehrdimensionalen Fall als Matrix). Im folgenden Listing wird ein Vektor mit vom Benutzer eingegebenen Werten belegt. Erweitern Sie das Programm so, dass eine Skalarmultiplikation aus Vektor (`array`) und beliebigem Skalar entsteht, wobei Sie den Skalarwert entweder fest vorgeben oder vom Benutzer abfragen können.

Tipp: Führen Sie die Berechnung der Skalarmultiplikation zwischen der Eingabe- und der Ausgabeschleife durch, so dass die mit dem Skalar multiplizierten Elemente an der richtigen Stelle wieder direkt im Array stehen. In der Ausgabeschleife wird dann automatisch der „skalierte“ Vektor ausgegeben. Beachten Sie außerdem den Unterschied zwischen Skalarmultiplikation und Skalarprodukt!

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

int main() {
  int array[10];

  for (int i = 0; i < 10; i++) {
    scanf("%d", &array[i]);
  }

  for (int i = 0; i < 10; i++) {
    printf("%d, ", array[i]);
  }

  return 0;
}
```

d) Lesen Sie zwei unterschiedliche Vektoren, die jedoch die gleiche Länge haben, ein. Bilden Sie die Differenz aus beiden Vektoren. Legen Sie den Ergebnisvektor in einem dritten Array ab.

e) Lesen Sie ein Array ein, welches Sie dann in einer Schleife „umdrehen“. Legen Sie zum Umdrehen aber kein zweites Array an!

f) Legen Sie ein mehrdimensionales Array aus Integerwerten an. Es soll sich um eine 3×3 Matrix handeln. Belegen Sie das Array direkt, also bei der Deklaration, mit beliebigen Integer-Werten. Geben Sie das Array dann aus. Tipp: Benutzen Sie zwei `for`-Schleifen, die geschachtelt sind. Die Ausgabe in der Konsole sollte z. B. wie folgt aussehen (natürlich mit möglicherweise anderen Werten):

```
+---+---+---+
|  1|  2|  3|
+---+---+---+
|  4|  5|  6|
+---+---+---+
|  7|  8|  9|
+---+---+---+
```

Aufgabe 5: Strings

a) Lesen Sie auf der Konsole Ihren Vornamen ein und geben Sie ihn aus!

b) Lesen Sie auf der Konsole Ihren Vornamen *und* Ihren Nachnamen ein und geben Sie beide aus! Benutzen Sie *nur eine* `scanf()`-Anweisung mit einem geeigneten Format-String!

c) Lesen Sie auf der Konsole ein beliebiges Wort ein und stellen Sie fest, wie viele Buchstaben es hat!

d) Schauen Sie sich nochmals eine `ASCII`-Tabelle an. Implementieren Sie dann ein Programm, welches beliebige Wörter (ohne Sonder- und Leerzeichen, aber mit beispielsweise Bindestrichen) einliest. Ausgegeben werden soll dann das „invertierte“ Wort; alle Großbuchstaben sollen zu Kleinbuchstaben werden und umgedreht. Bindestriche o. ä. ignorieren Sie bitte, lassen Sie sie also unverändert. „Anti-Falten-Creme“ wird dann zu „aNTI-fALTEN-cREME“.