Einführung in die Programmierung
Aufgabenblatt 6

1) Definieren Sie dazu einen Strukturtypen **`struct`** `Angaben`,
   die folgenden Angaben eines Patienten enthält:
   `Geburtsjahr, Koerpertemperatur, PillenProTag`.

   Legen Sie zwei Variablen dieses Typs an und weisen Sie diesen komponentenweise Werte zu. Sinnvollerweise lassen Sie diese vorher vom Benutzer eingeben.

   Schreiben Sie nun eine Funktion `vergleich()`, die zwei Variablen dieses Strukturtyps auf Gleichheit prüft. Sie liefert als Resultat eine 1, wenn die beiden übergebenen Variablen komponentenweise gleich sind bzw. 0 für Ungleichheit. Realisieren Sie in einem ersten Schritt die Funktion auf Basis des folgenden Prototyps (`medizin1.c`):

   ```
   int vergleich(struct Angaben pat1, struct Angaben pat2);
   ```

   Im zweiten Schritt soll die Funktion zwei Referenzen als Argumente erwarten. Der Funktionsprototyp sieht nun also wie folgt aus (`medizin2.c`):

   ```
   int vergleich(struct Angaben *pat1, struct Angaben *pat2);
   ```

   Welchen Vorteil hat die zweite Version?
   Wie kann die zweite Version noch verbessert werden?

   Im dritten Schritt (`medizin3.c`) soll nun die Strukturdefinition mittels **`typedef`** vorgenommen werden. Vergeben Sie den Alias `ANGABEN` für die definierte Struktur.

2) Entwickeln Sie eine kleine Adressverwaltung. Dabei sollen von einer Person folgende Daten gespeichert werden: Name, Postleitzahl, Wohnort, Straße und eine fortlaufende ID. Verwenden Sie die folgenden Konstanten und globalen Definitionen:

```
#define NAME_LEN         30
#define ORT_LEN          30
#define STRASSE_LEN      50
#define ADRESSBUCH_LEN   100

typedef struct
{
     int  id;                       // ID
     char name[NAME_LEN];           // Name der Person
     char strasse[STRASSE_LEN];     // Strasse
     long plz;                      // Postleitzahl
     char ort[ORT_LEN];             // Wohnort
} TADRESSE;

TADRESSE Adressbuch[ADRESSBUCH_LEN];
int AnzahlAdressen;
```

Implementieren Sie folgende Funktionen (wahlweise in einem oder mehreren separaten Modulen):

Eingabe() liest einen kompletten Datensatz ein. Die Gesamtzahl der gespeicherten Adressen `AnzahlAdressen` wird um eins erhöht. Hinweis: Verwenden Sie Funktionen wie `getchar()` oder `gets()` um Zeichenketten mit Leerzeichen einzulesen.

Ausgabe() gibt alle Datensätze aus.

SucheName() emöglicht die Suche nach einem bestimmten Datensatz anhand des Namens einer Person. Schreiben Sie die Funktion so, dass nicht der komplette Name angegeben werden muss. Hinweis: Benutzen Sie die Funktion `strncmp()` aus `string.h`.

LoescheDatensatz() löscht einen Datensatz. Gehen Sie wie in der Funktion SucheName vor, um einen Datensatz zu finden. Bauen Sie eine Sicherheitsabfrage ein. Füllen Sie die entstandene "Lücke" durch Umkopieren des letzten Datensatzes auf und korrigieren Sie die Gesamtzahl der gespeicherten Datensätze.

3) Erzeugen Sie dynamisch Speicherplatz für n `double`-Zahlen (n ist vom Benutzer abzufragen). Schreiben Sie je eine Funktion zur Eingabe und Ausgabe der Zahlen. Ihr Programm soll wie folgt in Funktionen aufgeteilt sein:

```
speicherAnforderung(…)      // reserviert Speicher für n Zahlen
eingabe(…)                  // liest die Zahlenfolge ein
ausgabe(…)                  // gibt die Zahlenfolge aus
speicherFreigabe(…)         // gibt den reservierten Speicher wieder
                            // frei
main()                      // liest n vom Benutzer ein und ruft die
                            // anderen Funktionen auf
```

Um den Zeiger auf den reservierten Speicherbereich in allen Funktionen nutzen zu können, muss dieser entweder als globale Variable angelegt werden oder wie folgt von der Funktion `speicherAnforderung(…)` zurückgeliefert werden:

```
double* speicherAnforderung()
{
    double *p;
    p = (double *) malloc( ...
    ...
    return p;
}
```

Im Folgenden muss nun dieser zurückgelieferte Zeiger in der `main()`-Funktion gesichert und an die anderen drei Funktionen als Parameter übergeben werden:

```
int main()
{
    double *dynFeld;

    dynFeld = speicherAnforderung();
    eingabe(dynFeld);
    ...

    return 0;
}
```

Hinweis: Eine alternative Formulierung der Funktion `speicherAnforderung(…)` könnte den Zeiger sogar als Referenzparameter übergeben, d.h. als "Zeiger auf einen Zeiger"). So könnte der Rückgabewert entfallen oder für Erfolg bzw. Misserfolg der Speicheranforderung genutzt werden:

```
void speicherAnforderung(double **p)
{
    *p = (double *) malloc( ...
    ...
}
```

Zusatzaufgabe: Schreiben Sie eine Funktion `sort(…)`, welche die Zahlen der Folge aufsteigend sortiert und rufen Sie diese zwischen der Ein- und Ausgabe auf. Verwenden Sie dazu einen Sortieralgorithmus Ihrer Wahl.