FH | JOANNEUM
University of Applied Sciences

Systemnahe Programmierung
SS 2024

Unit 4

Helmut Lindner

Zeiger

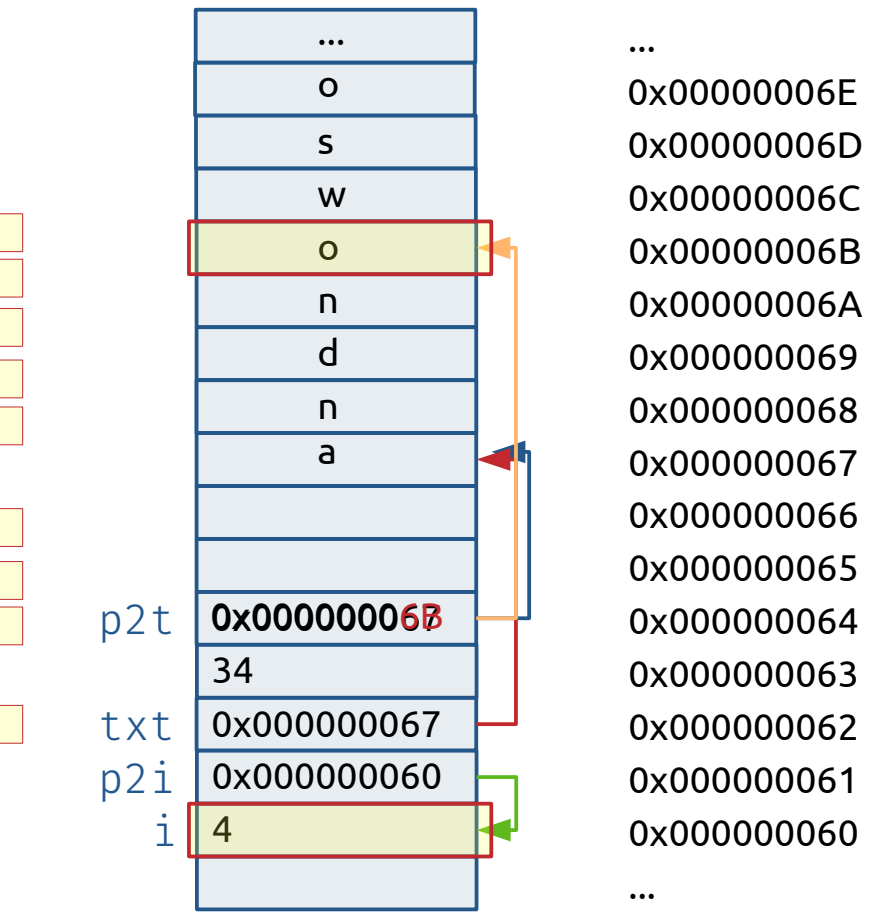
Wiederholung

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

int main() {
  int i = 4;
  int *p2i;
  char txt[] =
          "andnowsomethingcompletelydifferent";
  char *p2t;

  p2i = &i;
  p2t = txt;
  p2t += *p2i;

  printf("Position %d: %c\n", *p2i, *p2t);
}
```



Zeiger

Wiederholung

Immer von innen nach außen vorgehen ...
`[]` und `()` haben höhere Priorität als `*` und `&`

```
int *x[5]
    x ist ein Array
    x ist ein Array von Zeigern
    x ist ein Array von Zeigern auf int

    *x[] ist int,
    x[] ist ein Zeiger auf int,
    x ist ein Array von Zeigern auf int

int (*x)[5]
    x ist ein Zeiger
    x ist ein Zeiger auf ein Array
    x ist ein Zeiger auf ein Array von 5 integern

    (*x)[] ist int,
    (*x) ist ein Array von int
    x ist ein Zeiger auf ein Array von int
```

Zusammengesetzte Datentypen

Strukturen

Strukturen vereinen Daten verschiedenen Typs unter einem Namen.
Definition:

```
struct <name> { .... } [variable];
```

```
struct artikel {
    char artikelName[50];
    char artikelNummer[20];
};
```

Variable deklarieren:

```
struct artikel p;
```

oder

```
struct artikel {
    char artikelName[50];
    char artikelNummer[20];
} p;
```

Zusammengesetzte Datentypen

Strukturen - Verwendung

```
struct lagerBewegung {
    struct artikel a;
    float menge;
    time_t datum;
} bew1, bew2;
```

Komponenten der Struktur ansprechen:

```
bew1.menge = 10.0;
```

Als Zeiger:

```
struct lagerBewegung *p;
p→menge = 10.0;   → entspricht (*p).menge
```

Zuweisung:

```
bew1 = bew2;
bew1.artikel = a;
```

Zusammengesetzte Datentypen

Strukturen

Verschachtelung:

```
struct artikel {
    char artikelName[50];
    char artikelNummer[20];
};

struct lagerBewegung {
    struct artikel a;
    int menge;
    time_t datum;
    struct lagerBewegung *next;
};
```

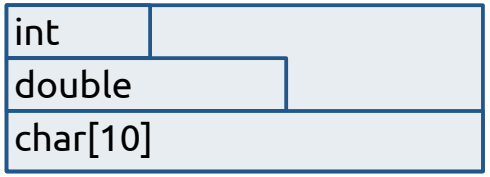
Zusammengesetzte Datentypen

Unions

Im Gegensatz zu einer Struktur liegen bei einer Union die Datenelemente virtuell übereinander. Es handelt sich also um einen Speicherplatz der immer nur eines der Elemente beinhaltet. Es wird der Speicherplatz für das Größte der Elemente reserviert.

```
union nummer {
    int i;
    double f;
    char s[10];
} u1;
```

```
u1.i=2;
u1.f=10.2;
u1.char=“Test“;
```



Es kann immer nur einer der Werte aktiv sein!

Die Verwendung ist analog zur Struktur.

Zusammengesetzte Datentypen

Unions

Beispiel „discriminated Union“:

```
struct punkt {
    int x;
    int y;
};
struct kreis {
    struct punkt mittelpunkt;
    double radius;
};

struct linie {
    struct punkt start;
    struct punkt ende;
};
struct quadrat {
    struct punkt linksOben;
    double seite;
};

struct element {
    int typ;  // 0 ... Punkt, 1  ... Linie, 2 ... Kreis, 3 ... Quadrat,
    union {
        struct punkt p;
        struct linie l;
        struct kreis k;
        struct quadrat q;
    };
};
struct element liste[10];
struct kreis mk={.mittelpunkt={3,4},.radius=3.14};
liste[0].typ=2; liste[0].k=mk;
printf("Typ: %d %d\n",liste[0].typ,liste[0].k.mittelpunkt.x);
```

Eigene Datentypen

typedef

Mit `typedef` kann für bestehende Datentypen ein neuer Name vergeben werden.

```
typedef unsigned char BYTE;
```

Vor allem auch für Strukturen praktisch, verkürzt die Deklarationen.

```
struct punkt {
    int x;
    int y;
};

typedef struct punkt PUNKT;
PUNKT p;
```

oder:

```
typedef struct punkt {
    int x;
    int y;
} PUNKT;
```

Dynamische Speicherverwaltung

Bereiche im Virtuellen Speicher

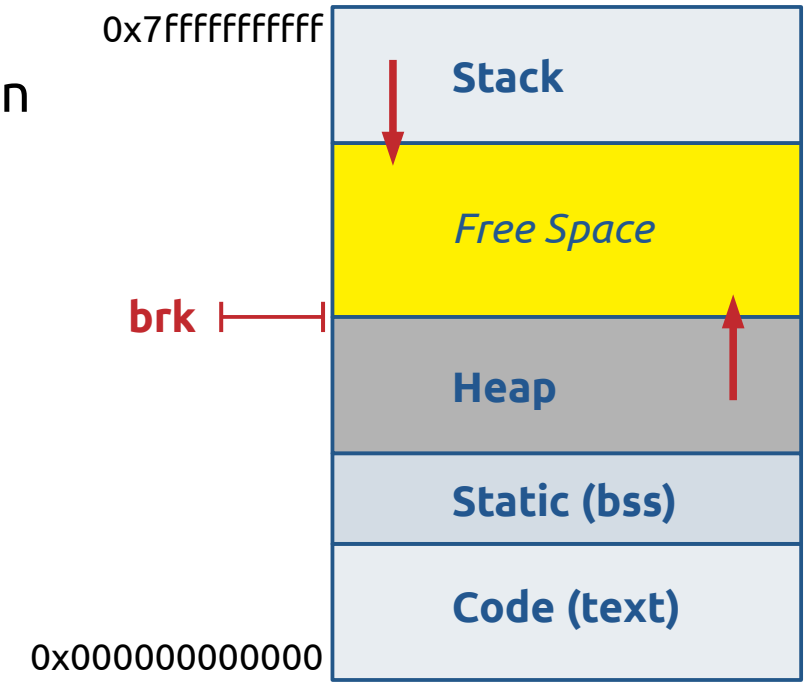
Jeder Prozess bekommt vom Betriebssystem seinen eigenen virtuellen Adressraum. Vom C-Runtime-System wird er in folgende Segmente eingeteilt:

Stack:
- Stack-Frames von Funktionsaufrufen
- Lokale Variable

Heap:
- Speicher für dynamische Speicheranforderungen

Static: Globale Variable und Konstante

Code: Executable (.o)



Dynamische Speicherverwaltung

Heap Speicher

Der Speicher für deklarierte Variable wird beim Start des Programms oder beim Aufruf einer Funktion automatisch reserviert und zugeordnet.

Der Heap-Speicher kann dynamisch alloziert und wieder freigegeben werden.

Die Funktion `malloc()` reserviert einen zusammenhängenden Speicherbereich im Heap.
Heap-Speicher kann nur mit Zeigern verwendet werden.

```
#include <stdlib.h>

void *malloc(size_t size);
    → „size“ Bytes reservieren.

void *calloc(size_t num, size_t size);
    → „num*size“ Bytes reservieren und mit 0 initialisieren.
```

Dynamische Speicherverwaltung

Beispiel

Platz für einen String mit max. 100 Zeichen reservieren:

```
char *string;
int laenge=101;

string = malloc(sizeof(char)*laenge);

if ( string == NULL ) {   → kein Speicher verfügbar → immer prüfen!
    ... Fehlerbehandlung ...
}
```

Dynamische Speicherverwaltung

Speicher freigeben

Mit `malloc()` oder `calloc()` angeforderter Speicher wird für die gesamte Lebenszeit des Prozesses belegt. Nicht mehr benötigter Speicherplatz muss explizit mit `free()` freigegeben werden.

```
#include <stdlib.h>
void free (void* ptr);
```

-> nicht benötigten Speicherplatz so früh wie möglich frei geben.

Dynamische Speicherverwaltung

Einen Speicherbereich ändern

Die Größe eines reservierten Speicherbereiches kann mit `realloc()` geändert werden.

```
#include <stdlib.h>
void *realloc(void *p, size_t new_size);
```

Beispiel:

```
char *buffer = malloc(1000);
...
buffer = realloc(buffer, 2000);
if ( buffer == NULL ) {
    ...
}
```

Die Operation ist aufwändig, da ev. der ursprüngliche Speicherbereich kopiert werden muss.

Sollte `realloc()` fehlschlagen wird der ursprüngliche Speicherbereich nicht freigegeben.

Dynamische Speicherverwaltung

Korrekte Verwendung - Zuwenig Heap Speicher

Um Probleme zu vermeiden, muss der von `malloc()` zurückgegebene Zeiger immer auf `NULL` geprüft werden.

```
char *buffer = malloc(sizeof(char)*laenge);
if ( buffer == NULL ) {
    ...
}
```

Dynamische Speicherverwaltung

Korrekte Verwendung - Memory Leaks

Um Memory Leaks zu vermeiden, muss nicht mehr benötigter Heap-Speicher freigegeben werden.

```
void funktion() {
    char *buffer = malloc(sizeof(char)*laenge);
    if ( buffer == NULL ) {
    }
    ...
    free(buffer); → wenn das nicht gemacht wird, bleibt der Speicher
                    reserviert und kann nicht mehr freigegeben werden.
}
```

Werden Speicherbereiche über einen längeren Zeitraum hinweg benötigt, so muss z.B.: über Indikatoren oder eigene Speicherverwaltungsfunktionen sichergestellt werden, das der Speicher so früh wie möglich wieder freigegeben wird.

Dynamische Speicherverwaltung

Korrekte Verwendung - Dangling Pointer

Wenn Zeiger auf bereits freigegebene Speicherbereiche verwendet werden hilft nichts mehr ...

```
char *buffer = malloc(sizeof(char)*laenge);
...
free(buffer);
...
printf("Inhalt %s\n", buffer);
```

Dynamische Speicherverwaltung

Korrekte Verwendung - free()

`free()` niemals auf bereits freigegebene Zeiger anwenden.

```
char *buffer = malloc(sizeof(char)*laenge);
...
free(buffer);
...
free(buffer);
```

Niemals Speicherbereiche freigeben, die nicht dynamisch alloziert wurden.

```
char *buffer = “Ich bin ein String“;
...
free(buffer);
```

Dynamische Speicherverwaltung

Korrekte Verwendung - Aliasing

Zeiger werden als *Alias* bezeichnet, wenn sie auf den gleichen Speicherbereich zeigen.

```
buffer1 = malloc(1000);
buffer2 = malloc(1000);
...
buffer2 = buffer1;  → buffer2 ist nicht mehr referenzierbar, erzeugt
                      ein Memory Leak von 1000 Bytes!
```

Noch gemeiner:

```
buffer1 = malloc(1000);
buffer2 = malloc(1000);
...
buffer2 = buffer1;
...
free(buffer1);
free(buffer2); → zeigt auf buffer1 und der wurde bereits freigegeben
```

Dynamische Speicherverwaltung

Good Practice

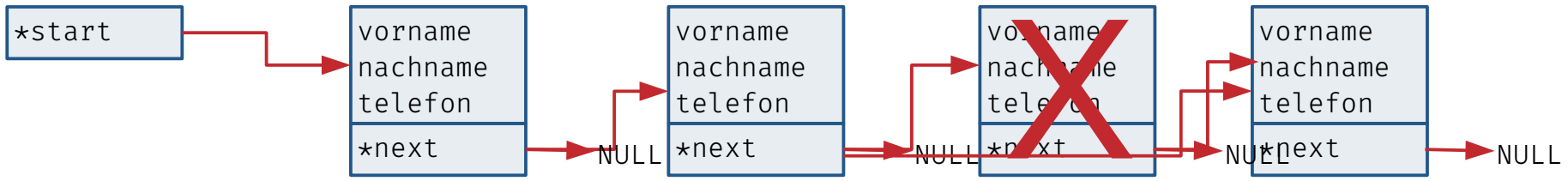
Zeiger die noch nicht oder bereits freigegeben wurden auf `NULL` setzen

```
char *buffer1 = NULL;
...
buffer1 = malloc(1000);
...
if (buffer1 ≠ NULL) {
    free(buffer1);
    buffer1=NULL;
}
```

Dynamische Speicherverwaltung

Übung 1

Schreiben sie ein Programm, mit dem sie ein Personenverzeichnis erstellen können. Verwenden sie als zugrundeliegende Datenstruktur eine einfach verkettete Liste.



Eine leere Liste ist durch `start==NULL` gekennzeichnet.
Neue Elemente werden am Ende der Liste angefügt.
Das Ende der Liste ist durch `next==NULL` gekennzeichnet.

Dynamische Speicherverwaltung

Übung 1

Bilden sie die Daten in folgender Struktur ab:

```
struct person {
  int id;
  char vorname[255];
  char nachname[255];
  char telefon[50];
  char email[255];
  struct person *next;
};
```

Die Liste soll auf dem Heap angelegt werden. Die `id` soll für jede Person eindeutig sein (Zähler).

Die Liste kann wie folgt deklariert werden:

```
typedef struct person person; // empfehlenswert
person *start=NULL;
```

Dynamische Speicherverwaltung

Übung 1

Implementieren sie im ersten Schritt folgende Funktionen:

- Eine neue Person anlegen:
  `person *personAnlegen(char *vorname, char *nachname, char *telefon, char *mail);`
  → Speicher allozieren
  → Daten in die Felder der Struktur kopieren
- Person in Liste einfügen:
  `void personEinfuegen(person *p) ;`
  → fügt einen neuen Personen-Eintrag am Ende der Liste ein.
- Eine Liste der Personen ausgeben:
  Vom Startknoten bis zum Ende der Liste durchgehen und jedes Listenelement (Person) ausgeben.
  Pro Person eine Zeile ausgeben. Die Datenfelder einer Person mit fester Breite und links ausgerichtet ausgeben.

Erzeugen sie zum Testen mind. 7 Personeneinträge, die sie in die Liste einfügen.
Geben sie alle Elemente der Liste aus.

Dynamische Speicherverwaltung

Übung 1

Beispiel Listenausgabe:

id	Vorname	Nachname	Telefon	E-Mail
1	David	Pfeifer	06886167181	DavidPfeifer@gmx.com
2	Doreen	Baier	06608191358	DoreenBaier@cuvox.de
3	Martin	Shuster	06643231711	MartinShuster@gmx.com
4	Felix	Fink	06814857140	FelixFink@gmx.com
5	Dieter	Schreiner	06884132145	DieterSchreiner@gmx.com
6	Dirk	Saenger	06508629090	DirkSaenger@gmx.com
7	Martina	Gloeckner	06509758086	MartinaGloeckner@gmx.com

Dateioperationen

Übersicht

Die C-Standardbibliothek unterstützt zwei Levels von Ein- und Ausgabefunktionen.

Standard I/O (stdio)
- Großer Umfang an Funktionen
- Gebufferte Ein- und Ausgabe
- Textbasierte und binäre Modi

Low-Level I/O
- universell (auch für Netzwerk Streams usw.)
- mehr Kontroll-Optionen als stdio

Filedeskriptoren:

- Standard I/O: `*FILE`
- Low-Level I/O: `int fd`

Dateioperationen

Schema

Alle Dateioperationen können auf Dateien oder andere Datenströme (z.B.: Netzwerkverbindung usw.) angewendet werden und folgen diesem Schema:

Datei öffnen

```
open()
```

Man erhält ein „Handle“ zur Identifkation des Streams.

Operationen auf dem Stream durchführen

```
read() / write() / seek()
```

Stream/Datei schliessen

```
close()
```

Standard I/O

Dateien öffnen und schliessen

Standard I/O arbeitet mit sog. Streams. Ein Stream wird durch einen Zeiger auf die die Struktur `FILE` identifiziert.

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

Die Standardstreams sind folgendermassen definiert:

```
FILE* stdin, stdout, stderr
```

Eine Datei im angegebenen Modus öffen:

```
FILE *fopen(const char *pfad, const char *mode);
```

Beispiel

```
FILE *stream = fopen("/home/test/prog.c", "r+");
```

Im Fehlerfall liefern diese Funktion einen NULL Wert zurück.

Dateistream schliessen:

```
int fclose(FILE *stream);
```

Standard I/O

Dateimodi

Modus	Operatoren
r	Öffnet die Datei zum Lesen. Wenn die Datei nicht existiert, liefert fopen den Wert NULL.
w	Öffnet die Datei zum Schreiben. Wenn die Datei nicht existiert wird sie angelegt. Eine vorhandene Datei wird überschrieben.
a	Öffnet die Datei zum Schreiben. Die geschriebenen Daten werden am Ende an die Datei angehängt. Wenn die Datei nicht existiert wird sie angelegt.
r+	Öffnet die Datei zum Lesen und Schreiben. Wenn die Datei nicht existiert liefert fopen den Wert NULL.
w+	Öffnet die Datei zum Lesen und Schreiben. Wenn die Datei nicht existiert wird sie angelegt.
a+	Öffnet die Datei zum Lesen und Schreiben. Die Daten werden am Ende an die Datei angehängt. Wenn die Datei nicht existiert wird sie angelegt.
...t	Öffnen als Textdatei (LF-Zeichen werden mit CR-Zeichen ergänzt).
...b	Öffnen als binäre Datei.

Standard I/O

Textbasierte Ein-/Ausgabe - Zeichenweise

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int ungetc(int c, FILE *stream);
int getc(FILE *stream);
int putchar(int c);
int getchar(void);
```

Textbasierte Ein-/Ausgabe - Zeilenweise

```
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
char *gets(char *s); → Nicht verwenden, höchst unsicher!
int puts(char *s);
```

Funktionen mit `int` Returnwert liefern `EOF` falls ein Fehler auftritt.
Funktionen mit `char*` Returnwert liefern `NULL` falls ein Fehler auftritt.

Achtung: manche dieser Funktionen liefern das ‘`\n`‘ mit, manche nicht.
z.B.: `fgets()` mit ‘`\n`‘, `gets()` ohne.

Standard I/O

Textbasierte Ein-/Ausgabe Beispiel:

```
char buffer[BUFFER_SIZE];
FILE *fp;

if ((fp=fopen("datei.txt","r") == NULL) {
  printf(„Fehler …“\n);
  exit(EXIT_FAILURE);
}

while (fgets(buffer,BUFFER_SIZE,fp) ≠ NULL) {
  fprintf(stderr, "Zeile: %s", buffer);
}

fclose(fp);
```

Standard I/O

Formatierte Ein-/Ausgabe

```
int fprintf( FILE *stream, const char *format, ... );
int fscanf( FILE *stream, const char *format, ... );
```

Fehlerbehandlung

EOF und Fehlerstatus abfragen:

`int feof( FILE *stream );` -> != 0 wenn EOF gesetzt ist

`int ferror( FILE *stream );` -> != 0 wenn ein Fehler aufgetreten ist

EOF und Fehlerstatus löschen:

```
void clearerr( FILE *stream );
```

Standard I/O

Blockweise (binäre) Operationen

Im Gegensatz zu den textbasierten Stdio-Funtionen, werden hier keine Zeichen interpretiert!

```
size_t fread(void *ptr, size_t groesse,size_t anzahl, FILE *stream);
size_t fwrite(void *ptr, size_t groesse, size_t anzahl, FILE *stream);
```

Positionieren im Stream

Die aktuelle Position im Stream setzen:

```
int fseek( FILE *stream, long offset, int mode );
```

mode:

```
    SEEK_SET ... Offset vom Start des Streams
    SEEK_CUR ... Offset von aktueller Position
    SEEK_END ... Offset vom Ende des Streams
```

Aktuelle Position im Stream abfragen:

```
long ftell( FILE *stream );
```

Schreib-/Leseposition an den Anfang der Datei setzen:

```
void rewind( FILE *stream );
```

Standard I/O

Beispiel - Binäre Dateien

```
struct artikel {
  char artikelName[50];
  char artikelNummer[20];
  float gewicht;
  int preis;
} art;
```

Schreiben:

```
FILE *f = fopen("artikel.dat","w+");
if (f==NULL) {
    → Datei konnte nicht zum Schreiben geöffnet bzw. angelegt werden
}
fwrite(&art, sizeof(artikel), 1, f);
...
fclose(f);
```

Lesen:

```
FILE *f = fopen("artikel.dat","r");
if (f==NULL) {
    → Datei existiert nicht
}
fread(&art, sizeof(artikel), 1, f);
...
```

Standard I/O

Probleme mit Zeichenströmen

Einlesen von `n` Bytes, was passiert mit überschüssigen Zeichen?
Beispiel:

```
fgets(buffer, 5, stdin);
…
while(!end) {
    cmd = getchar();
     ...
}
```

Eingabe: `Guten Tag!\n`
`fgets()` liefert den String: `Gute\0`
`getchar()` liefert: `n Tag!\n`

Lösung:

`fflush(stdin);` → nicht POSIX konform! Funktioniert nicht unter Linux.

Zeichen „überlesen“:

`z.B.: while ((getchar()) ≠ '\n');`

Dynamische Speicherverwaltung

Übung 2

Wir erweitern das Programm nun um ein einfaches Menüsystem und eine Möglichkeit zur Datenerfassung.
Das Menüsystem besteht darin, dass wir einzelne Zeichen als Befehle von `stdin` abfragen
Implementieren sie folgende Befehle:

n ... Neue Person anlegen
→ Daten für eine Person via `stdin` erfassen
→ Personenrecord anlegen und in Liste einfügen

a ... Liste ausgeben
→ Liste aller Personen ausgeben (wie gehabt)

e ... Ende
→ Programm beenden

Geben sie den Speicher der durch die Liste belegt wird, vor dem Beenden des Programms frei.

Dynamische Speicherverwaltung

Übung 2

So soll es aussehen:

nach Eingabe von `'n'`:

```
(n)Neu (a)ausgeben (e)Ende
n
Vorname: Helmut
Nachname: Lindner
Telefon: 0123456
E-Mail: helmut.lindner@fh-joanneum.at
Person wurde angelegt.

(n)Neu (a)ausgeben (e)Ende
```

nach Eingabe von `'a'`:

```
(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
a

|id    | Vorname                   | Nachname                  | Telefon             | E-Mail
-------------------------------------------------------------------------------------------
|1     | Helmut                    | Lindner                   | 0123456             | helmut.lindner@fh-joanneum.at

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
```

Standard I/O

Übung 3

Ergänzen sie das Programm „Personenverzeichnis“ aus dem letzten Übungsbeispiel um die Funktionen „Speichern“ und „Lesen“.

Die Funktion Speichern soll:
- Einen Dateinamen abfragen
- Die Personenliste in der Datei im Binärformat speichern.

Die Funktion Lesen soll:
- Einen Dateinamen abfragen
- Die Personenliste aus der Datei lesen und die verkettete Liste wieder aufbauen.

Mögliche Fehler wie Datei nicht gefunden usw. sollen abgefangen werden. Beim Lesen nicht vergessen den Zähler für die ID richtig zu setzen und falls bereits eine Liste existiert, nachzufragen was getan werden soll und diese gegebenenfalls zu löschen.

Dynamische Speicherverwaltung

Übung 4 (Optional)

Das Programm zur Personendatenverwaltung soll um weitere Funktionen erweitert werden:

l ... Person löschen
→ Die id einer Person soll abgefragt werden (`scanf()`) und der entsprechende Listeneintrag soll gelöscht werden. Der Speicher für den Eintrag soll freigegeben werden.

s ... Personen suchen
→ ein Suchtext soll abgefragt werden und alle Personeneinträge deren Vor- oder Nachname den Suchstring enthält sollen ausgegeben werden.
Verwenden sie die Funktion `strstr()` (Einfacher Vergleich, keine Wildcards)

Dynamische Speicherverwaltung

Übung 4

So soll es aussehen:
Löschen eines Eintrags mit `'l'`:

```
(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
a

|id   | Vorname                  | Nachname                 | Telefon             | E-Mail
-------------------------------------------------------------------------------------------
|1    | David                    | Pfeifer                  | 06886167181         | DavidPfeifer@gmx.com
|2    | Doreen                   | Baier                    | 06608191358         | DoreenBaier@cuvox.de
|3    | Martin                   | Shuster                  | 06643231711         | MartinShuster@gmx.com
|4    | Felix                    | Fink                     | 06814857140         | FelixFink@gmx.com
|5    | Dieter                   | Schreiner                | 06884132145         | DieterSchreiner@gmx.com
|6    | Dirk                     | Saenger                  | 06508629090         | DirkSaenger@gmx.com
|7    | Martina                  | Gloeckner                | 06509758086         | MartinaGloeckner@gmx.com

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
l
Person loeschen ID eingeben: 1
Person mit der ID 1 wurde gelöscht.

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
a

|id   | Vorname                  | Nachname                 | Telefon             | E-Mail
-------------------------------------------------------------------------------------------
|2    | Doreen                   | Baier                    | 06608191358         | DoreenBaier@cuvox.de
|3    | Martin                   | Shuster                  | 06643231711         | MartinShuster@gmx.com
|4    | Felix                    | Fink                     | 06814857140         | FelixFink@gmx.com
|5    | Dieter                   | Schreiner                | 06884132145         | DieterSchreiner@gmx.com
|6    | Dirk                     | Saenger                  | 06508629090         | DirkSaenger@gmx.com
|7    | Martina                  | Gloeckner                | 06509758086         | MartinaGloeckner@gmx.com

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
```

Dynamische Speicherverwaltung

Übung 4

So soll es aussehen:
Suchen mit dem Befehl `'s'` und dem Suchtext `"Di"`:

```
(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
a

|id   | Vorname                  | Nachname                 | Telefon             | E-Mail
-------------------------------------------------------------------------------------------
|2    | Doreen                   | Baier                    | 06608191358         | DoreenBaier@cuvox.de
|3    | Martin                   | Shuster                  | 06643231711         | MartinShuster@gmx.com
|4    | Felix                    | Fink                     | 06814857140         | FelixFink@gmx.com
|5    | Dieter                   | Schreiner                | 06884132145         | DieterSchreiner@gmx.com
|6    | Dirk                     | Saenger                  | 06508629090         | DirkSaenger@gmx.com
|7    | Martina                  | Gloeckner                | 06509758086         | MartinaGloeckner@gmx.com

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
s
Suchtext eingeben: Di

|id   | Vorname                  | Nachname                 | Telefon             | E-Mail
-------------------------------------------------------------------------------------------
|5    | Dieter                   | Schreiner                | 06884132145         | DieterSchreiner@gmx.com
|6    | Dirk                     | Saenger                  | 06508629090         | DirkSaenger@gmx.com

(n)Neu (l)löschen (s)suchen (a)ausgeben (e)Ende
```