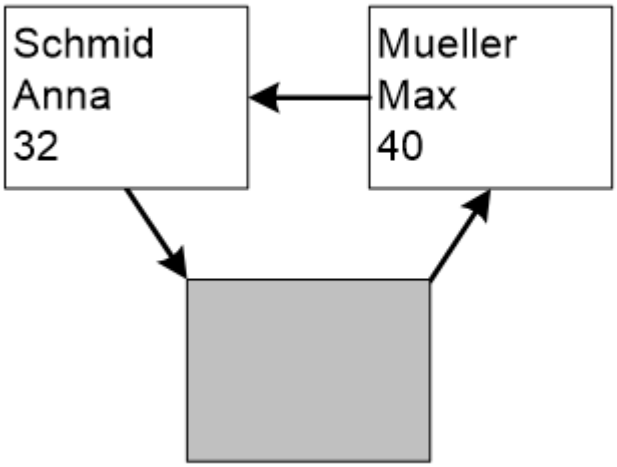
# SNP: Personen Verwaltung – Linked List






## 1 Übersicht

In diesem Praktikum schreiben Sie eine einfache Personenverwaltung. Dabei werden Sie etliche Elemente von C anwenden:

- Header Files selber schreiben, inklusive Include Guard
- Typen definieren
- Funktionen mit *by value* und *by reference* Parametern deklarieren und definieren
- einfache Variablen, Pointer Variablen, struct Variablen und Array Variablen benutzen
- Strukturen im Speicher dynamisch allozieren und freigeben
- I/O und String Funktionen aus der Standard Library anwenden
- Anwender Eingaben verarbeiten
- Fehlerbehandlung

# 2 Lernziele

In diesem Praktikum wenden Sie viele der bisher gelernten C Elemente an.

- Sie können anhand dieser Beschreibung ein vollständiges C Programm schreiben.
- Sie können Unit Tests schreiben welche die wesentlichen Funktionen des Programms individuell testen.

Die Bewertung dieses Praktikums ist am Ende angegeben.

Erweitern Sie die vorgegebenen Code Gerüste, welche im **`git`** Repository **`snp-lab-code`** verfügbar sind.

# 3 Personenverwaltung

## 3.1 Programmfunktion

Das Programm soll in einer Schleife dem Benutzer jeweils folgende Auswahl bieten, wovon eine Aktion mit Eingabe des entsprechenden Buchstabens ausgelöst wird:

***`I(nsert), R(emove), S(how), C(lear), E(nd)`***:

- **Insert**: der Benutzer wird aufgefordert, eine Person einzugeben
- **Remove**: der Benutzer wird aufgefordert, die Daten einer zu löschenden Person einzugeben
- **Show**: eine komplette Liste aller gespeicherten Personen wird in alphabetischer Reihenfolge ausgegeben
- **Clear**: alle Personen werden gelöscht
- **End**: das Programm wird beendet

## 3.2 Designvorgaben

### Verkettete Liste

Da zur Kompilierzeit nicht bekannt ist, ob 10 oder 10'000 Personen eingegeben werden, wäre es keine gute Idee, im Programm einen statischen Array mit z.B. 10'000 Personen-Einträgen zu allozieren. Dies wäre ineffizient und umständlich beim sortierten Einfügen von Personen. In solchen Situationen arbeitet man deshalb mit dynamischen Datenstrukturen, die zur Laufzeit beliebig (solange Speicher vorhanden ist) wachsen und wieder schrumpfen können. Eine sehr populäre dynamische Datenstruktur ist die verkettete Liste und genau die werden wir in diesem Praktikum verwenden.

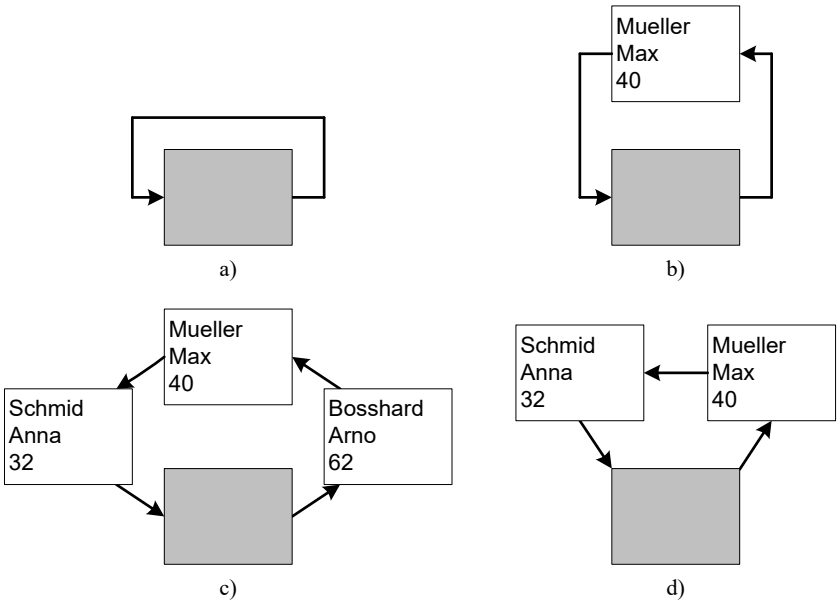



Abbildung 1: Zyklisch verkettete Liste

Eine verkettete Liste bedeutet, dass ein Knoten der verketten Liste einen Datensatz einer Person speichert und zusätzlich einen Pointer auf den nächsten Knoten in der Liste aufweist (siehe Abbildung 1). In dieser Pointer Variablen (***next*** in der ***node_t*** Struktur unten) steht also einfach die Adresse des nächsten Knotens.

Die leere Liste besteht aus einem einzelnen Element, welches keine spezifische Person abspeichert und welches auf sich selbst zeigt (Abbildung 1 a). Dieses Element ist der Einstiegspunkt der Liste (auch Anker oder Wurzel genannt) und ist das einzige Element, das Sie im Programm direkt kennen und einer Variablen zuweisen. Dieses Element können Sie statisch allozieren (z.B. ***node_t anchor;***, siehe Details weiter unten), denn es existiert während der gesamten Ausführungszeit. Alle anderen Elemente erreichen Sie ausgehend vom Anker, indem Sie einmal, den Pointern folgend, im Kreis herum gehen. Abbildung 1 b zeigt die Liste nach dem Einfügen der Person ***Max Mueller***, ***40*** Jahre. Nach dem Einfügen von zwei weiteren Personen sieht die Datenstruktur aus wie in Abbildung 1 c. Das Entfernen der Person ***Arno Bosshard*** führt zu Abbildung 1 d.

Eine Person kann zugefügt werden, indem dynamisch ein neuer Knoten erzeugt wird und dieser in die verkettete Liste eingefügt wird. Beim Einfügen müssen die Adressen der Knoten so den Pointern zugewiesen werden, dass die Kette intakt bleibt.

Ein Knoten wird entfernt, indem der entsprechende Knoten aus der Verkettung herausgelöst wird (**`next`** des Vorgängerknotens soll neu auf **`next`** des herauszulösenden Knotens zeigen) und dann der Speicher des entsprechenden Knotens freigegeben wird.

### Personen und Knoten Records

Die für je eine Person zu speichernden Daten sollen in folgendem C **`struct`** zusammengefasst sein.

```
#define NAME_LEN 20

typedef struct {
  char         name[NAME_LEN];
  char         first_name[NAME_LEN];
  unsigned int age;
} person_t;
```

Jeder Knoten der verketteten Liste soll aus folgendem C **`struct`** bestehen.

```
typedef struct node {
  person_t     content;        // in diesem Knoten gespeicherte Person
  struct node *next;           // Pointer auf den nächsten Knoten in der Liste
} node_t;
```

### Vorschlag: zyklisch verkettete Liste

Erkennen des Endes der Liste: bei der zyklisch verketteten Liste zeigt das letzte Element wieder auf den Anker, die Liste bildet also einen Kreis. Dies ist in Abbildung 1 so abgebildet.

Alternativ könnte man das Ende erkennbar machen, indem die Kette anstelle von zyklisch, mit einem NULL Pointer endet.

Die Wahl ist ihnen überlassen ob sie die eine oder andere Art der End-Erkennung implementieren. In der Beschreibung wird angenommen, dass es sich um eine zyklisch verkettete Liste handelt.

### Sortiertes Einfügen

Die Personen Records sollen sortiert in die Liste eingefügt werden. Dies bedeutet, dass vom Anker her gesucht werden soll, bis der erste Knoten gefunden wurde dessen Nachfolgeknoten entweder „grösser“ ist als der einzufügende Knoten, oder wo das Ende der Liste erreicht ist. Die Ordnung (grösser, gleich, kleiner) soll so definiert sein:

```
// if (p1 > p2) { ... }
if (person_compare(&p1, &p2) > 0) { ... }
```

```
/**
 * @brief  Compares two persons in this sequence: 1st=name, 2nd=first_name, 3rd=age
 * @param  a [IN] const reference to 1st person in the comparison
 * @param  b [IN] const reference to 2nd person in the comparison
 * @return =0 if all record fields are the same
 *         >0 if all previous fields are the same, but for this field, a is greater
 *         <0 if all previous fields are the same, but for this field, b is greater
 * @remark strncmp() is used for producing the result of string field comparisons
 * @remark a->age - b->age is used for producing the result of age comparison
 */
int person_compare(const person_t *a, const person_t *b);
```

### Eingabe

Fehlerhafte Wahl der Operation in der Hauptschleife soll gemeldet werden, ansonsten aber ignoriert werden.

Fehlerhafte Eingabe der Personenangaben sollen gemeldet werden und die gesamte Operation (z.B. Insert) verworfen werden.

Zu prüfende Fehler bei Personeneingaben:

- für die Namen
  - zu lange Namen
- für das Alter
  - keine Zahl
- Duplikat
  - derselbe Record soll nicht doppelt in der Liste vorkommen

Weitergehende Prüfungen sind nicht erwartet.

Zu beachten: bei fehlerhafter Eingabe darf kein „Memory Leak“ entstehen, d.h. potentiell auf dem Heap allozierter Speicher muss im Fehlerfall freigegeben werden.

### 3.3 Bestehender Programmrahmen

Der Programmrahmen besteht aus den unten aufgelisteten Files. Es sollen weitere Module in **`src`** hinzugefügt werden und die bestehenden Files ergänzt werden gemäss den Aufgaben.

| | |
|---|---|
| **`Makefile`** | → **zu ergänzen** mit neuen Modulen |
| **`tests/tests.c`** | → **zu ergänzen** gemäss Aufgaben (implementieren von Unit Tests) |
| **`src/main.c`** | → **zu ergänzen** gemäss Aufgaben (Hauptprogramm) |

# 4 Aufgabe 1: Modularisierung – API und Implementation main.c

Kreieren Sie folgende Files in **`src`** und implementieren Sie **`main.c`** basierend auf dem unten von Ihnen gegebenen API.

| File | Inhalt |
|---|---|
| **`person.h`** | Typ Definitionen:<br>- **`person_t... // siehe Beschreibung oben`**<br>Funktionsdeklarationen:<br>- **`// siehe Beschreibung oben`**<br>**`int person_compare(const person_t *a, const person_t *b);`**<br>- gegebenenfalls weitere Funktionen für die Bearbeitung von Personen |
| **`list.h`** | Typ Definitionen:<br>- **`node_t ... // siehe Beschreibung oben`**<br>Funktionsdeklarationen:<br>- Funktionen für insert, remove, clear Operationen auf der Liste |

Das Hauptprogramm soll die Eingabeschleife implementieren und die obigen Funktionen (wo angebracht) aufrufen.

# 5 Aufgabe 2: Implementierung von person.c und list.c

Fügen Sie die beiden Implementationsfiles **`person.c`** und **`list.c`** zu **`src`**. Fügen Sie die beiden Module im **`Makefile`** zu der vorgegebenen Variablen **`MODULES`** hinzu, so dass sie beim **`make`** Aufruf auch berücksichtigt werden.

### 5.1 Teilaufgabe: Implementierung von person.c

Implementieren Sie die Funktionen aus **`person.h`**.

Falls nötig, stellen Sie weitere statische Hilfsfunktionen in **`person.c`** zur Verfügung.

### 5.2 Teilaufgabe: Implementierung von list.c

Implementieren Sie die Funktionen aus **`list.h`**.

Falls nötig, stellen Sie weitere statische Hilfsfunktionen in **`list.c`** zur Verfügung.

# 6 Aufgabe 3: Unit Tests

Schreiben Sie Unit Tests für mindestens die folgenden Funktionen

- **`person.h`**:
    - **`person_compare`**
- **`list.h`**:
    - **`list_insert`**
    - **`list_remove`**
    - **`list_clear`**

Es existieren in **`tests/tests.c`** schon vier Test Rahmen für diese Test Cases.

In diese Test Cases sollen die entsprechenden Funktionen unter verschiedenen Bedingungen isoliert aufgerufen werden und deren Verhalten überprüft werden.

Verwenden Sie für die Überprüfung die CUnit **`CU_ASSERT_...`** Makros.

Siehe dazu auch **`man CUnit`**.

Wenn die obigen Teilaufgaben erfolgreich umgesetzt sind, laufen die Tests ohne Fehler durch.

# 7 Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

| **Aufgabe** | **Kriterium** | **Punkte** |
|---|---|---|
| | Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären. | |
| 1 | API von list.h und person.h plus die Implementation von main.c | 2 |
| 2 | Teilaufgabe: person.c | 2 |
| | Teilaufgabe: list.c | 2 |
| 3 | Unit Tests | 2 |