Übungsaufgaben Programmieren in C

Autor: Martin Röhricht

Version: 2024-07-03

Vorwort

In der Vorlesung »Informatik2« für die Studiengänge »Elektrotechnik« und »Fahrzeugsysteme« an der Hochschule Esslingen soll den Studenten das Programmieren in C beigebracht werden. Obwohl die Programmiersprache C eine vergleichsweise alte Programmiersprache ist, bietet sie den Vorteil, dass ihr Sprachumfang – verglichen mit den meisten »modernen« Programmiersprachen – von überschaubarer Größe ist. Auf der anderen Seite muss sich ein Programmierer in C noch um einiges »von Hand« kümmern und kann vergleichsweise leicht Fehler machen, die ihm das Laufzeitsystem nicht verzeiht.

Um unseren Studenten dabei zu helfen, sich auf die anstehende Klausur vorzubereiten, habe ich mich auf die Suche nach geeigneten Übungsaufgaben gemacht. Manche Aufgaben sind allseits bekannt, bei manchen habe ich mich inspirieren lassen, manche habe ich mir selbst ausgedacht.

Die folgenden Aufgaben sollen Ihnen die Möglichkeit geben, das Programmieren in C weiter zu üben. Sie finden im ersten Abschnitt dieses Dokuments die verschiedenen Übungen, zu denen Sie im besten Fall eine eigenständige Lösung entwickeln.

Im zweiten Abschnitt des Dokuments finden Sie dann die jeweiligen Musterlösungen. Bedenken Sie dabei, dass es häufig sehr viele unterschiedliche Lösungen für dasselbe Problem geben kann.

Bei den Lösungsvorschlägen habe ich darauf geachtet, dass auf der einen Seite möglichst wenig »Spezialfunktionen« aus der Standardbibliothek verwenden werden und auf der anderen Seite durchaus von den gängigsten Funktionen (bspw. `strlen()`) Gebrauch gemacht wird. Grundsätzlich sollten auch keine anderen Bibliotheken außer der Standardbibliothek verwendet werden.

Die Code-Beispiele wurden so erstellt, dass sie trotz zahlreicher Compiler-Warnstufen[1] keine Warnungen ausgeben.

Stuttgart im Dezember 2023
Martin Röhricht
<martin.roehricht@hs-esslingen.de>

[1] `-Wall -Wextra -Wpedantic -Wshadow -Wformat=2 -Wcast-align -Wconversion -Wsign-conversion -Wnull-dereference` unter Clang und GCC

Inhaltsverzeichnis

Teil I.

Aufgaben

1. Finde die kleinste Zahl in einem Array

Thema: Arrays **Schwierigkeitsgrad:** ●○○○○

(a) Erstellen Sie ein Programm, welches aus einem gegebenen Array aus Zahlen die kleinste Zahl ermittelt.

Vorlage:

```c
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};

    int min;
    // Durchsuchen Sie das Array nach dem kleinsten
    // Element ...

    printf("Kleinstes Element: %d\n", min);
}
```

>_ Bildschirmausgabe

```
Kleinstes Element: 2
```

(b) Verändern Sie Ihr Programm derart, dass es eine eigene Funktion `find_min()` gibt, die als Argumente ein Array aus Zahlen und die Größe des Arrays entgegennimmt, und den kleinsten Wert aus dem Array an den Aufrufer zurückliefert.

2. Zähle die Häufigkeiten eines Werts in einem Array

Thema: Arrays **Schwierigkeitsgrad:** ●○○○○

(a) Erstellen Sie ein Programm, welches aus einem gegebenen Array aus Ganzzahlen und einer vorgegebenen Zahl die Häufigkeit ausgibt, mit der diese Zahl im Array vorkam.

Vorlage:

```c
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int to_find = 4;

    int count;
    // Zählen Sie, wie häufig die Zahl to_find
    // im Array vorkommt ...

    printf("Anzahl in Array: %d\n", count);
}
```

>_ Bildschirmausgabe

```
Anzahl in Array: 2
```

(b) Verändern Sie Ihr Programm derart, dass es eine eigene Funktion `occurrences()` gibt, die diese Operation durchführt und die Häufigkeit der gesuchten Zahl an den Aufrufer zurückliefert.

Überlegen Sie sich, welche Parameter eine solche Funktion benötigt.

3. Ein Array umkehren

Thema: Arrays **Schwierigkeitsgrad:** ●○○○○

(a) Erstellen Sie ein Programm, welches die Reihenfolge eines gegebenen Arrays aus Zahlen umkehrt.

Vorlage:

```c
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8};

    // Kehren Sie die Reihenfolge der Zahlen des
    // Arrays um ...

    printf("Umgekehrte Reihenfolge:\n");

    // Geben Sie die umgekehrte Reihenfolge aus ...
}
```

>_ Bildschirmausgabe

```
Umgekehrte Reihenfolge:
myArray[2] = 8
myArray[3] = 4
myArray[4] = 24
myArray[5] = 9
myArray[6] = 2
myArray[7] = 4
```

(b) Verändern Sie Ihr Programm derart, dass es eine eigene Funktion `reverse()` gibt, die als Argumente ein Array aus Zahlen und die Größe des Arrays entgegennimmt, und die Reihenfolge der Zahlen im übergebenen Array umkehrt.

4. Überprüfen, ob ein String ein Palindrom ist

Thema: Strings **Schwierigkeitsgrad:** ●●○○○

Ein Palindrom ist ein Wort, eine Phrase, eine Zahl oder eine andere Sequenz von Zeichen, die vorwärts und rückwärts gelesen identisch ist. Typische Beispiele für Palindrome sind Wörter wie »Anna« oder »Otto«, bei denen die Buchstabenfolge von beiden Enden aus gleich bleibt.

Erstellen Sie eine Funktion `isPalindrome()`, die überprüft, ob eine gegebene Zeichenkette (String) ein Palindrom ist. Die Funktion soll `true` zurückgeben, wenn der String ein Palindrom ist, und `false`, wenn er keines ist. Berücksichtigen Sie dabei, dass bei der Überprüfung Groß- und Kleinschreibung ignoriert werden sollte.

Vorlage:

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

int main(void) {
    char string1[] = "Kein Palindrom";
    char string2[] = "Reliefpfeiler";   // ist ein Palindrom
    char string3[] = "Rentner";         // ist ein Palindrom

    // Die im Folgenden genutzte isPalindrome()-Funktion
    // müssen Sie erstellen

    if (isPalindrome(string1))
        printf("\"%s\" ist ein Palindrom\n", string1);
    else
        printf("\"%s\" ist kein Palindrom\n", string1);

    if (isPalindrome(string2))
        printf("\"%s\" ist ein Palindrom\n", string2);
    else
        printf("\"%s\" ist kein Palindrom\n", string2);
```

```
    if (isPalindrome(string3))
        printf("\"%s\" ist ein Palindrom\n", string3);
    else
        printf("\"%s\" ist kein Palindrom\n", string3);
}
```

>_ Bildschirmausgabe

```
"Kein Palindrom" ist kein Palindrom
"Reliefpfeiler" ist ein Palindrom
"Rentner" ist ein Palindrom
```

5. Verkettung von Strings unter Verwendung dynamischer Speicherzuweisung

Thema: Strings, Dynamische Speicherverwaltung **Schwierigkeitsgrad:** ●●○○○

Erstellen Sie eine Funktion `stringAppend()`, die zwei Strings als Eingabeparameter entgegennimmt und diese miteinander verkettet. Die Funktion soll für den neuen, verketteten String dynamisch Speicher auf dem Heap reservieren und einen Pointer auf diesen Speicher an den Aufrufer zurückliefern.

Vorlage:

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

int main(void) {
    char first[] = "Übungsaufgaben Programmieren ";
    char second[] = "in C";

    char *newString = stringAppend(first, second);
    printf("%s\n", newString);

    free(newString);
}
```

>_ Bildschirmausgabe

```
Übungsaufgaben Programmieren in C
```

6. Finde das am häufigsten vorkommende Zeichen in einem String

Thema: Strings **Schwierigkeitsgrad:** ●●○○○

Erstellen Sie eine Funktion `printMaxChars()`, die basierend auf einem an die Funktion übergebenen String das Zeichen ermittelt, das am häufigsten in diesem String vorkommt und sowohl das Zeichen als auch die Anzahl auf dem Bildschirm ausgibt.

Vorlage:

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

int main(void) {
    char string[] = "Fischers Fritze fischt frische Fische, frische
    ↪  Fische fischt Fischers Fritze";

    printMaxChars(string);
}
```

>_ Bildschirmausgabe

```
Zeichen i kommt 10 mal vor.
```

7. Zähle die Anzahl an Wörtern in einem String

Thema: Strings **Schwierigkeitsgrad:** ●●●○○

Erstellen Sie eine Funktion `countWords()`, die zählt, wie häufig ein Wort innerhalb einer Zeichenkette (String) vorkommt. Die Funktion soll das Wort und die Zeichenkette als Eingabeparameter entgegennehmen und die Anzahl an den Aufrufer zurückliefern. Berücksichtigen Sie dabei, dass bei der Überprüfung Groß- und Kleinschreibung ignoriert werden sollen.

Vorlage:

```c
#include <stdio.h>

int main(void) {
    char string[] = "Fischers Fritze fischt frische Fische, frische
    ↪  Fische fischt Fischers Fritze";
    char word[] = "Fische";

    // Die im Folgenden genutzte countWords()-Funktion
    // müssen Sie erstellen
    int count = countWords(string, word);

    printf("Das Wort \"%s\" kam %d-mal im String vor", word,
    ↪  count);
}
```

>_ Bildschirmausgabe

```
Das Wort "Fische" kam 2-mal im String vor
```

8. Finde den Durchschnitt von Gruppen von Zahlen in einer Datei

Thema: Dateioperationen **Schwierigkeitsgrad:** ●●●○○

Erstellen Sie ein Programm, das aus einer Textdatei eine Zeile einliest, in der Zahlen in spezieller Gruppierung vorliegen. Die Anzahl der Zahlen einer jeden Gruppe wird durch die erste Zahl der Gruppe angegeben.

Bei einer Datei mit folgendem Inhalt:

Textdatei.txt

```
5 24 13 83 22 4 3 99 23 45 4 82 34 11 9 6 13 22 93 42 85 34
```

Gibt die erste Zahl 5 an, dass die nächsten fünf Zahlen zu einer Gruppe gehören. Die nach dieser Gruppe stehende Zahl 3 gibt dann an, dass die nächsten drei Zahlen zur nächsten Gruppe gehören, und so weiter.

Das Programm soll nun für jede Gruppe alle zugehörigen Zahlen einlesen und den Durchschnitt der Zahlen ermitteln. Z.B. soll die Ausgabe für die obenstehenden Zahlen folgendes ergeben:

Bildschirmausgabe

```
Gruppe mit 5 Elementen: 29.20
Gruppe mit 3 Elementen: 55.67
Gruppe mit 4 Elementen: 34.00
Gruppe mit 6 Elementen: 48.17
```

9. Finde die jüngste Person

Thema: Strukturen, Pointer **Schwierigkeitsgrad:** ●●●○○

Erstellen Sie eine Funktion, welche aus einem Array an Personen nach der jüngsten Person sucht. Die Funktion soll einen Pointer auf ein Array übergeben bekommen, welches seinerseits Pointer auf Strukturen von Personen enthält. Jede Person-Struktur enthält einen Vornamen, einen Nachnamen und ein Alter.

Die Funktion soll nun die Einträge des Arrays nach der jüngsten Person durchsuchen und einen Pointer auf diesen Array-Eintrag zurückliefern.

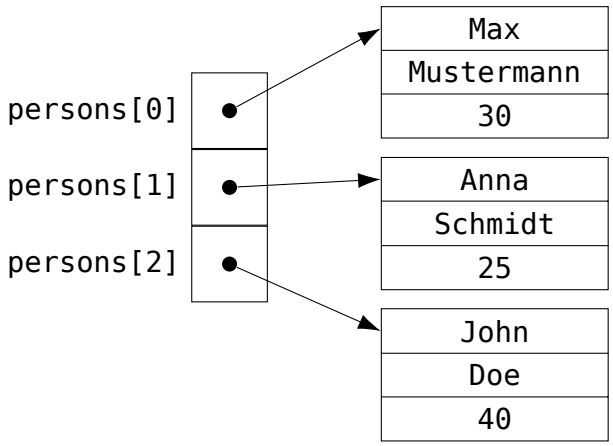
Vorlage:

```c
#include <stdio.h>

typedef struct {
    char firstName[50];
    char lastName[50];
    int age;
} Person;

int main(void) {
    Person person1 = {"Max", "Mustermann", 30};
    Person person2 = {"Anna", "Schmidt", 25};
    Person person3 = {"John", "Doe", 40};

    Person *persons[] = {&person1, &person2, &person3};

    int size = sizeof(persons) / sizeof(persons[0]);

    Person *youngest = findYoungestPerson(persons, size);

    printf("Name: %s %s, Age: %d\n", youngest->firstName,
    ↪  youngest->lastName, youngest->age);

    return 0;
}
```

>_ Bildschirmausgabe

```
Name: Anna Schmidt, Age: 25
```

Man kann sich die Struktur also wie folgt vorstellen:



10. Erstelle eine verkettete Liste

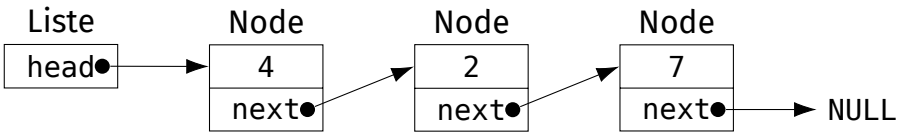
Thema: Strukturen, Pointer **Schwierigkeitsgrad:** ●●●●●

Erstellen Sie eine Datenstruktur in Form einer verketteten Liste, die es Ihnen erlaubt, dynamisch eine beliebige Anzahl an Elementen zu Ihrer Liste hinzuzufügen oder daraus zu entfernen.

Eine verkettete Liste besteht aus einer Menge an Knoten, die jeweils Daten enthalten, sowie einen Pointer auf den nächsten Knoten in der Liste. Im letzten Knoten der Liste enthält der Pointer auf den nächsten Knoten den Wert `NULL`. Daran kann man erkennen, dass man am Ende der Liste angekommen ist.

Die Knoten einer verketteten Liste können prinzipiell beliebige (aber gleichartige) Daten enthalten. In unserem Beispiel reicht es uns, wenn einfach ein Integer-Wert pro Knoten gespeichert wird.

Eine Liste mit drei Knoten und den Daten `4`, `2` und `7` können Sie sich wie folgt vorstellen:



Wir brauchen demnach eine Struktur für eine Liste, eine Struktur für einen Knoten und dann noch Funktionen zum

- Initialisieren einer Liste
- Hinzufügen von Knoten zu einer Liste
- Entfernen von Knoten aus einer Liste
- Löschen einer Liste mitsamt seiner Knoten
- (Optional) der Bildschirmausgabe aller Knoten einer Liste

Auf Basis des folgenden Codes, sollte dann die dahinterstehende Ausgabe erfolgen:

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

```c
// Strukturdefinition für einen Knoten `Node`
// ...

// Strukturdefinition für eine Liste `List`
// ...

// Funktion initList()
// ...

// Funktion addNode()
// ...

// Funktion removeNodesWithValue()
// ...

// Funktion printList()
// ...

// Funktion freeList()
// ...

int main(void) {
    List myList;
    initList(&myList);

    addNode(&myList, 1);
    addNode(&myList, 2);
    addNode(&myList, 3);
    addNode(&myList, 2);

    printf("Original Liste:\n");
    printList(&myList);

    removeNodesWithValue(&myList, 2);
    printf("Liste nach dem Entfernen aller 2er:\n");
    printList(&myList);

    freeList(&myList);
}
```

>_ Bildschirmausgabe

```
Original Liste:
Element 1: data = 1
Element 2: data = 2
Element 3: data = 3
Element 4: data = 2
Liste nach dem Entfernen aller 2er:
Element 1: data = 1
Element 2: data = 3
```

(a) Erstellen Sie eine Datenstruktur `Node`, die aus zwei Elementen besteht: einmal einem Integer (`data`) und einmal einem Pointer auf eine Struktur `Node`.

(b) Erstellen Sie eine Datenstruktur `List`, die aus zwei Elementen besteht: einmal einem Pointer auf eine Struktur `Node` und einmal einem Integer `size`, in dem die Größe der Liste verwaltet wird.

(c) Erstellen Sie eine Funktion `initList()`, die einen Pointer auf eine Liste übergeben bekommt und die Elemente dieser Liste mit den Werten `NULL` bzw. `0` initialisiert.

(d) Erstellen Sie eine Funktion `addNode()`, die einen Pointer auf eine Liste und einen Integer übergeben bekommt. Die Funktion soll dynamisch Speicherplatz für einen neuen Knoten auf dem Heap allokieren und dann den an die Funktion übergebenen Integerwert dem neuen Knoten hinzufügen.

Wichtig ist nun, dass Sie prüfen, ob die Liste leer ist oder nicht. Falls die Liste nicht leer sein sollte, soll das neue Element an das Ende der Liste gehängt werden.

(e) Erstellen Sie eine Funktion `printList()`, die alle Knoten der Liste inklusive ihrer `data`-Werte auf dem Bildschirm ausgibt.

(f) Erstellen Sie eine Funktion `freeList()`, die die Daten aller Knoten in der Liste vom Heap freigibt und die Liste auf den Ursprungszustand zurücksetzt.

(g) Erstellen Sie eine Funktion `removeNodesWithValue()`, in der nach allen Knoten in der Liste gesucht wird, die einen bestimmten Wert enthalten. Dazu bekommt die Funktion einen Pointer auf eine Liste und einen Integerwert übergeben.

Alle Knoten, deren `data`-Element den Wert enthalten, der an die Funktion übergeben wurde, sollen aus der Liste entfernt werden.

11. Duplikate aus einem Array entfernen

Thema: Pointer, Dynamische Speicherverwaltung **Schwierigkeitsgrad:** ●●●●○

Erstellen Sie eine Funktion `removeDuplicates()`, die doppelte aufeinanderfolgende Elemente aus einem sortierten Array entfernt. Die Funktion soll dabei den Speicherbereich der Liste entsprechend anpassen.

Beachten Sie hierbei folgendes: das Array wird in der `main()`-Funktion dynamisch auf dem Heap allokiert. Da der Speicherbereich durch die `removeDuplicates()`-Funktion möglicherweise angepasst werden muss, muss die Funktion einen *Pointer auf das Array* sowie die Größe des Arrays als Argumente entgegennehmen.

Vorlage:

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

int main(void) {
    int size = 10;
    int *myArray = malloc((size_t)size * sizeof(int));

    if (myArray == NULL) {
        printf("Speicher konnte nicht reserviert werden.\n");
        return 1;
    }

    // Hilfsarray, um myArray zu initialisieren
    int initArray[] = {1, 2, 3, 3, 4, 5, 5, 5, 6, 7};
    for (int i = 0; i < size; i++) {
        myArray[i] = initArray[i];
    }

    // Ausgabe der Array-Elemente vor der Bearbeitung
    printf("Array vor der Bearbeitung:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", myArray[i]);
    }
```

```
    printf("\n");

    removeDuplicates(&myArray, &size);

    // Ausgabe der Array-Elemente nach der Bearbeitung
    printf("Array nach der Bearbeitung:\n");
    for (int i = 0; i < size; i++) {
        printf("%d ", myArray[i]);
    }
    printf("\n");

    free(myArray);
    return 0;
}
```

>_ Bildschirmausgabe

```
Array vor der Bearbeitung:
1 2 3 3 4 5 5 5 6 7
Array nach der Bearbeitung:
1 2 3 4 5 6 7
```

12. Datenstruktur sortieren

Thema: Pointer, Dyn. Speicherverwaltung, Sortieren **Schwierigkeitsgrad:** ●●●○○

Erstellen Sie ein Programm, welches ein Team von Fußballspielern verwalten und sortieren kann. Ein Fußballspieler soll durch folgende Datenstruktur repräsentiert werden:

```c
#define NAME_LENGTH 64

typedef struct {
    char firstName[NAME_LENGTH];
    char lastName[NAME_LENGTH];
    int age;
} Player;
```

(a) Erstellen Sie eine Funktion `createPlayer()`, die drei Parameter entgegennimmt: den Vornamen, den Nachnamen und das Alter eines Spielers. Die Funktion soll dynamisch Speicher für einen Spieler allokieren, den Speicher gemäß den übergebenen Daten initialisieren und einen Pointer auf den Speicher an den Aufrufer zurückliefern.

(b) Erstellen Sie eine Funktion `printPlayer()`, die einen Pointer auf einen Spieler entgegennimmt und daraufhin die Informationen eines Spielers in formatierter Form auf dem Bildschirm ausgibt.

(c) Erstellen Sie eine Vergleichsfunktion `comparePlayerByLastName(const void *pa, const void *pb)`, die zwei Spieler nach ihren Nachnamen lexikografisch vergleicht. Die beiden Pointer `const void *pa` und `const void *pb` sollen zwei Pointer auf die zu vergleichenden Array-Elemente darstellen, d. h. in unserem Beispiel handelt es sich jeweils um *einen Pointer auf einen Pointer auf ein `Player`-Objekt*.

Damit der Vergleich funktioniert, soll die Funktion folgenden Rückgabewert an den Aufrufer zurückliefern:

- Eine negative Zahl, wenn der Nachname von `pa` lexikographisch *vor* dem Nachnamen von `pb` kommt.

- `0`, wenn die Nachnamen gleich sind.
- Eine positive Zahl, wenn der Nachname von `pa` lexikographisch *nach* dem Nachnamen von `pb` kommt.

(d) Eine Funktion `freeTeam()`, die den für das Team allokierten Speicher wieder freigibt.

Vorlage:

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

int main(void) {
    const int NUM_PLAYERS = 5;
    Player **team = malloc(NUM_PLAYERS * sizeof(Player *));

    team[0] = createPlayer("Thomas", "Müller", 34);
    team[1] = createPlayer("Florian", "Wirtz", 21);
    team[2] = createPlayer("İlkay", "Gündoğan", 33);
    team[3] = createPlayer("Jamal", "Musiala", 21);
    team[4] = createPlayer("Manuel", "Neuer", 38);

    printf("Vor der Sortierung:\n");
    for (int i = 0; i < NUM_PLAYERS; i++)
        printPlayer(team[i]);

    qsort(team, NUM_PLAYERS, sizeof(Player *),
     ↪  comparePlayerByLastName);

    printf("\nNach der Sortierung:\n");
    for (int i = 0; i < NUM_PLAYERS; i++)
        printPlayer(team[i]);

    freeTeam(team, NUM_PLAYERS);

    return 0;
}
```

>_ Bildschirmausgabe

```
Vor der Sortierung:
Thomas Müller, 34 Jahre
Florian Wirtz, 21 Jahre
İlkay Gündoğan, 33 Jahre
Jamal Musiala, 21 Jahre
Manuel Neuer, 38 Jahre

Nach der Sortierung:
İlkay Gündoğan, 33 Jahre
Jamal Musiala, 21 Jahre
Thomas Müller, 34 Jahre
Manuel Neuer, 38 Jahre
Florian Wirtz, 21 Jahre
```

Die Vorlage verwendet für ein Team aus Spielern ein dynamisch allokiertes Array aus Pointern auf `Player`-Objekte:

```
Player **team = malloc(NUM_PLAYERS * sizeof(Player *));
```

Sie können sich die Struktur wie folgt vorstellen:

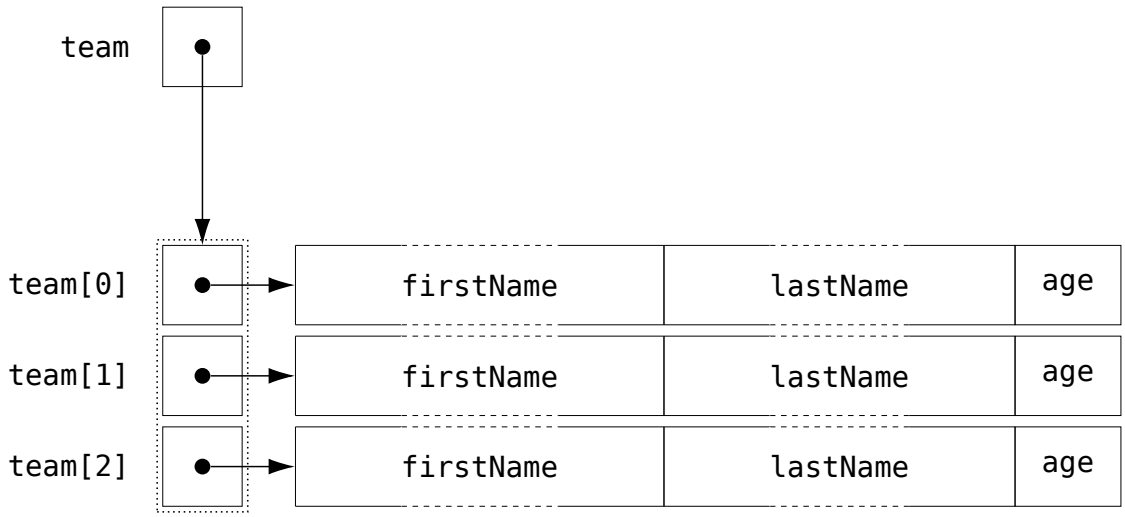


Abbildung 12.1.: *Darstellung des Pointers auf ein Array von Pointern auf `Player`-Objekte*

Vergegenwertigen Sie sich an dieser Stelle auch den Inhalt der Pointer der Vergleichsfunktion:

```
comparePlayerByLastName(const void *pa, const void *pb)
```

Die beiden Pointer `pa` und `pb` zeigen beide jeweils auf ein Array-Element des Arrays `team` und enthalten dementsprechend selbst wieder einen Pointer auf ein `Player`-Objekt.

Teil II.

Lösungen

1. Finde die kleinste Zahl in einem Array

(a) Zuerst initialisieren wir den Wert des gesuchten Minimums auf das erste Element des Arrays. Im nächsten Schritt durchlaufen wir das Array Eintrag für Eintrag in einer for-Schleife. Falls der Wert des aktuellen Eintrags kleiner ist als das bisher gefundene Minimum, ersetzen wir den Wert des Minimums durch den Wert des Eintrags.

Hinweis: für die generische und damit automatische Ermittlung der Größe des Arrays verwenden wir hier den Ausdruck:

```
int arraySize = sizeof(myArray) / sizeof(int);
```

```c
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int arraySize = sizeof(myArray) / sizeof(int);

    int min;

    min = myArray[0];
    for (int i = 0; i < arraySize; i++) {
        if (myArray[i] < min) {
            min = myArray[i];
        }
    }

    printf("Kleinstes Element: %d\n", min);
}
```

(b) In diesem Fall erstellen wir eine Funktion, die zwei Eingabeparameter besitzt: ein int-Array und einmal die Größe des Arrays. Der Funktionssignatur kann das Array in zwei gleichbedeutenden Formen entgegennehmen: entweder als `int array[]` oder als `int *array`. Die Größe des Arrays muss der Funktion übergeben werden, da sie diese ansonsten nicht erkennen kann.

```c
#include <stdio.h>

int find_min(int array[], int size) {
    int min = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] < min) {
            min = array[i];
        }
    }
    return min;
}

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int arraySize = sizeof(myArray) / sizeof(int);

    int min = find_min(myArray, arraySize);

    printf("Kleinstes Element: %d\n", min);
}
```

2. Zähle die Häufigkeiten eines Werts in einem Array

(a) Zuerst legen wir eine Variable `count` an, die wir mit dem Wert 0 initialisieren. Im nächsten Schritt durchlaufen wir das Array Eintrag für Eintrag in einer for-Schleife. Falls der Wert des aktuellen Eintrags dem gesuchten Wert entspricht, erhöhen wir den Zähler count um Eins.

Hinweis: für die generische und damit automatische Ermittlung der Größe des Arrays verwenden wir hier den Ausdruck:

```
int arraySize = sizeof(myArray) / sizeof(int);
```

```
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int to_find = 4;
    int arraySize = sizeof(myArray) / sizeof(int);

    int count = 0;
    for (int i = 0; i < arraySize; i++) {
        if (myArray[i] == to_find)
            count++;
    }

    printf("Anzahl in Array: %d\n", count);
}
```

(b) In diesem Fall erstellen wir eine Funktion, die drei Eingabeparameter besitzt: ein int-Array, dann die Größe des Arrays und schließlich noch die Zahl, nach der gesucht werden soll.

Auch hier ist es so, dass wir die Größe des Arrays an die Funktion übergeben müssen, da sie sonst innerhalb der Funktion nicht ermittelt werden kann.

```c
#include <stdio.h>

int occurrences(int array[], int size, int to_find) {
    int count = 0;

    for (int i = 0; i < size; i++) {
        if (array[i] == to_find)
            count++;
    }
    return count;
}

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int to_find = 4;
    int arraySize = sizeof(myArray) / sizeof(int);

    int count = occurrences(myArray, arraySize, to_find);

    printf("Anzahl in Array: %d\n", count);
}
```

3. Ein Array umkehren

(a) Um die Elemente eines Arrays umzukehren, müssen wir eine typische `swap`-Operation realisieren. Dazu brauchen wir eine temporäre Variable (`temp`), in der wir das auszutauschende Element zwischenspeichern.

Beim Durchlaufen des Arrays vertauschen wir schrittweise die Elemente der ersten Hälfte in aufsteigender Form mit den Elementen der zweiten Hälfte in absteigender Form.

Beachten Sie hier den Ausdruck:

```
myArray[arraySize - i - 1]
```

Da Arrays vom Index 0 beginnend adressiert werden und nur bis zum Index *Größe minus 1*, müssen wir die Zahl 1 bei der Berechnung abziehen.

Beachten Sie, dass die Berechnung sowohl für eine gerade als auch eine ungerade Anzahl an Elementen funktioniert. Das liegt an der Art und Weise, wie in C eine Ganzzahl-Division durchgeführt wird, bei der das Ergebnis auf die nächstkleinere Zahl gerundet wird (also nicht mathematisch gerundet).

Hinweis: für die generische und damit automatische Ermittlung der Größe des Arrays verwenden wir hier den Ausdruck:

```
int arraySize = sizeof(myArray) / sizeof(int);
```

```c
#include <stdio.h>

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int arraySize = sizeof(myArray) / sizeof(int);
    int temp = 0;

    for (int i = 0; i < (arraySize / 2); i++) {
        temp = myArray[i];
        myArray[i] = myArray[arraySize - i - 1];
        myArray[arraySize - i - 1] = temp;
    }

    printf("Umgekehrte Reihenfolge:\n");
```

```
    for (int i = 0; i < arraySize; i++) {
        printf("myArray[%d] = %d\n", i, myArray[i]);
    }
}
```

(b) Bei der Auslagerung der Berechnung in eine eigenständige Funktion müssen enstprechend das Array und die Größe an die Funktion übergeben werden. Der Rückgabewert ist **void**, da die Funktion die Elemente im Array direkt umkehrt und daher keinen Wert an den Aufrufer zurückliefern muss.

```
#include <stdio.h>

void reverse(int array[], int size) {
    int temp = 0;

    for (int i = 0; i < (size / 2); i++) {
        temp = array[i];
        array[i] = array[size - i - 1];
        array[size - i - 1] = temp;
    }
}

int main(void) {
    int myArray[] = {4, 2, 9, 24, 4, 8, 5, 24};
    int arraySize = sizeof(myArray) / sizeof(int);

    reverse(myArray, arraySize);

    printf("Umgekehrte Reihenfolge:\n");
    for (int i = 0; i < arraySize; i++) {
        printf("myArray[%d] = %d\n", i, myArray[i]);
    }
}
```

4. Überprüfen, ob ein String ein Palindrom ist

Die `isPalindrome()`-Funktion bekommt einen String übergeben und gibt einen Booleschen Wert (`true` oder `false`) an den Aufrufer zurück. Dafür haben wir den Header `stdbool.h` eingebunden.

Zuerst ermitteln wir die Länge und die Mitte des Strings mithilfe der `strlen()`-Funktion aus dem `string.h`-Header. Das Ergebnis ist der Index, der die Mitte des Strings markiert. Bei der Überprüfung auf ein Palindrom wird dieser Wert verwendet, um nur bis zur Mitte des Strings zu iterieren, da die zweite Hälfte des Strings spiegelbildlich zur ersten Hälfte sein sollte, wenn es sich um ein Palindrom handelt.

In der Funktion wird eine for-Schleife verwendet, die bis zur Mitte des Strings läuft. Jedes Zeichen der ersten Stringhälfte wird mit dem korrespondierenden Zeichen von hinten verglichen. Da die Groß-/Kleinschreibung keine Rolle spielen soll, verwenden wir die `tolower()`-Funktion aus dem `ctype.h`-Header.

Stimmen zwei miteinander verglichenen Zeichen nicht überein, wird die Funktion direkt beendet und der Wert `false` an den Aufrufer zurückgegeben. Falls bis zum Ende des Schleifendurchlaufs keine Unstimmigkeiten aufgetreten sind, gibt die Funktion `true` zurück.

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h> // für strlen()
#include <ctype.h>  // für tolower()

bool isPalindrome(char string[]) {
    int len = (int)strlen(string);
    int middle = len / 2;

    for (int i = 0; i < middle; i++) {
        if (tolower(string[i]) != tolower(string[len - i - 1]))
            return false;
    }
    return true;
}
```

5. Verkettung von Strings unter Verwendung dynamischer Speicherzuweisung

Da die Funktion `stringAppend()` dynamisch Speicher auf dem Heap für den neuen String reservieren soll, müssen zunächst die Längen der beiden übergebenen Strings mithilfe der `strlen()`-Funktion ermittelt werden. Da `strlen()` das abschließende Nullendezeichen eines Strings nicht in seiner Längenberechnung berücksichtigt, wird in der Berechnung der Gesamtlänge ein zusätzliches Byte für dieses Endezeichen eingerechnet:

```
size_t lengthTotal = length1 + length2 + 1;
```

Als nächstes nutzen wir die `malloc()`-Funktion aus dem `stdlib.h`-Header, um den Speicher dynamisch auf dem Heap zu reservieren.

Daraufhin kopieren wir zeichenweise die Zeichen aus den übergebenen Strings in den neu allokierten Speicherbereich; zuerst für den ersten String, dann für den zweiten. Beim zweiten Kopiervorgang müssen wir darauf achten, dass wir ab dem richtigen Index anfangen, die Zeichen zu kopieren.

Abschließend dürfen wir nicht vergessen, dass Nullendezeichen an den neuen String anzuhängen, um den neuen String korrekt zu terminieren und den Pointer auf den Speicherbereich des neuen Strings an den Aufrufer zurückliefern.

Bei einer Lösung wie dieser ist es wichtig, daran zu denken, dass der Speicher für diesen neuen String später mit `free()` freigegeben wird, um Speicherlecks zu vermeiden.

```
char *stringAppend(char *firstString, char *secondString) {
    size_t length1 = strlen(firstString);
    size_t length2 = strlen(secondString);
    size_t lengthTotal = length1 + length2 + 1;

    char *concatenatedString = malloc(lengthTotal * sizeof(char));

    for (size_t i = 0; i < length1; i++)
        concatenatedString[i] = firstString[i];
```

```
    for (size_t i = 0; i < length2; i++)
        concatenatedString[length1 + i] = secondString[i];

    concatenatedString[lengthTotal - 1] = '\0';

    return concatenatedString;
}
```

6. Finde das am häufigsten vorkommende Zeichen in einem String

Um das am häufigsten vorkommende Zeichen in einem String zu ermitteln, ist es notwendig, den String mehrfach zu durchlaufen. Dies erfordert zwei ineinander verschachtelte Schleifen: In der äußeren Schleife wird jeweils ein Zeichen aus dem String entnommen und in der Variablen `currentChar` gespeichert. Anschließend zählen wir in der inneren Schleife die Häufigkeit dieses Zeichens im gesamten String, wofür die Variable `currentOccurrence` verwendet wird. Hierfür reicht es, wenn die innere Schleife von der aktuellen Position der äußeren Schleife bis zum Ende des Strings läuft.

Wichtig ist, dass wir bei jedem neuen Durchlauf der äußeren Schleife die Variablen `currentChar` und `currentOccurrence` neu initialisieren. Falls das aktuell betrachtete Zeichen (`currentChar`) identisch mit dem bis dahin am häufigsten gefundenen Zeichen (`maxChar`) ist, können wir dieses Zeichen überspringen und direkt zum nächsten fortschreiten, um redundante Zählungen zu vermeiden.

In der inneren Schleife vergleichen wir jedes Zeichen des Strings mit `currentChar`. Stimmen sie überein, erhöhen wir `currentOccurrence`. Nach Abschluss der inneren Schleife prüfen wir, ob `currentOccurrence` größer als die bisherige maximale Häufigkeit (`maxOccurrence`) ist. Ist dies der Fall, aktualisieren wir die Variablen `maxChar` und `maxOccurrence` entsprechend.

```
void printMaxChars(char *string) {
    size_t length = strlen(string);
    char maxChar = '\0';
    char currentChar;
    int maxOccurrence = 0;
    int currentOccurrence = 0;

    for (size_t i = 0; i < length; i++) {
        currentChar = string[i];
        currentOccurrence = 0;
        if (currentChar == maxChar)
            continue;
```

```
        for (size_t j = i; j < length; j++) {
            if (string[j] == currentChar)
                currentOccurrence++;
        }
        if (currentOccurrence > maxOccurrence) {
            maxOccurrence = currentOccurrence;
            maxChar = currentChar;
        }
    }

    printf("Zeichen %c kommt %d mal vor.\n", maxChar,
    ↪  maxOccurrence);
}
```

7. Zähle die Anzahl an Wörtern in einem String

Bei dieser Aufgabenstellung spielen zwei Dinge eine wesentliche Rolle: es ist wichtig zu verstehen, woran wir erkennen, was ein Wort ausmacht; zum anderen müssen wir eine Strategie entwickeln, um durch den String zu laufen, um nach Wörtern zu suchen.

Direkt vor einem eigenständigen Wort muss ein Leerraumzeichen stehen, denn wenn wir nach dem Wort »Mutter« suchen, soll »Tagesmutter« nicht mitgezählt werden. Die einzige Ausnahme dieser Regel wäre das allererste Zeichen im String. Außerdem darf das Zeichen hinter dem gesuchten Wort wiederum kein alphabetisches Zeichen sein, denn wenn wir nach dem Wort »Mutter« suchen, sollte »Muttertag« wiederum nicht mitgezählt werden.

Bezüglich der Strategie müssen wir auf jeden Fall zeichenweise durch den gesamten String laufen. Das tun wir in Zeile 11 mittels der while-Schleife, die solange läuft, bis das nachfolgende Zeichen dem String-Endezeichen entspricht. Für den aktuell zu untersuchenden Index des Strings verwenden wir die Variable `stringPos`, die entsprechend sukzessive erhöht werden muss.

Jetzt wenden wir die Suchkriterien an: zuerst prüfen wir, ob das Zeichen am aktuell verwendeten Index überhaupt ein alphabetisches Zeichen ist. Falls nicht, können wir direkt den Index erhöhen und den nächsten Schleifendurchlauf starten (Zeilen 14-17).

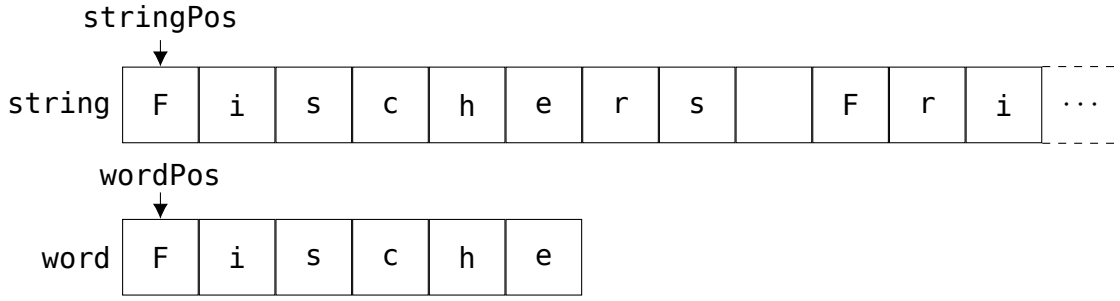
Hatten wir am aktuellen Index ein alphabetisches Zeichen, dann geht es darum, zu prüfen, ob das Zeichen davor ein Leerraumzeichen war (mit der Ausnahme des allerersten Zeichens im String) (Zeilen 19-22).

War auch diese Prüfung erfolgreich, stehen wir offensichtlich an einem Wortanfang und es geht darum die nächsten Zeichen des Strings mit denen unseres Suchworts zu vergleichen.

Hierfür verwenden wir eine for-Schleife, die maximal so viele Durchläufe durchläuft, wie das Suchwort Zeichen enthält (`lenWord`). Wir vergleichen jetzt jedes Zeichen des Suchworts am Index `wordPos` mit dem Zeichen des Strings am Index `stringPos`. Da wir die Groß-/Kleinschreibung nicht berücksichtigen wollen, wenden wir auf beide Zeichen die `tolower()`-Funktion an.

Falls die beiden Zeichen nicht übereinstimmen, entspricht das gerade untersuchte Wort im String nicht unserem Suchwort. Wir setzen daher unseren booleschen Wert `wordFound` auf `false` und brechen aus der inneren Schleife aus.

Andernfalls müssten alle Zeichen des untersuchten Worts dem unseres Suchtworts entsprochen haben. Dann geht es darum, zu prüfen, ob das untersuchte Wort auch



abgeschlossen ist, wofür hinter dem untersuchten Wort ein Leerraumzeichen stehen müsste. Dafür prüfen wir in Zeile 22, ob an der nun folgenden Stelle ein alphabetisches Zeichen steht.

Sollte dies zutreffen, so haben wir das Suchwort leider nicht gefunden und müssen einen Schritt weiter im Index und die Suche erneut mit einem neuen Schleifendurchlauf starten.

Andernfalls haben wir das Suchwort tatsächlich gefunden und können jetzt den Zähler `wordCount` um Eins erhöhen (Zeile 37-38).

Abschließend müssen wir für die while-Schleife noch dafür sorgen, dass der Index für den nächsten Schleifendurchlauf um Eins erhöht wird.

```
1  #include <stdio.h>
2  #include <string.h> // für strlen()
3  #include <ctype.h>  // für tolower() und isalpha()
4  #include <stdbool.h>
5
6  int countWords(char string[], char word[]) {
7      int lenWord = (int)strlen(word);
8      int wordCount = 0;
9      int stringPos = 0;
10
11     while (string[stringPos + 1] != '\0') {
12         bool validWord = true;
13
14         if (!isalpha(string[stringPos])) {
15             stringPos++;
16             continue;
17         }
18
19         if (stringPos != 0 && string[stringPos - 1] != ' ') {
20             stringPos++;
21             continue;
22         }
```

```
23
24          for (int wordPos = 0; wordPos < lenWord; wordPos++) {
25              if (tolower(string[stringPos]) !=
                ↪  tolower(word[wordPos])) {
26                  validWord = false;
27                  break;
28              }
29              stringPos++;
30          }
31
32          if (isalpha(string[stringPos])) {
33              stringPos++;
34              continue;
35          }
36
37          if (validWord)
38              wordCount++;
39
40          stringPos++;
41      }
42
43      return wordCount;
44  }
```

8. Finde den Durchschnitt von Gruppen von Zahlen in einer Datei

Im Folgenden werden Ihnen zwei mögliche Lösungen vorgestellt. Eine, bei der wir die Zeile komplett in einen Puffer einlesen und dann mit der `strtok()`-Funktion in einzelne »*Tokens*« zerlegen; und eine, bei der wir keinen separaten Puffer anlegen und direkt mit der `fscanf()`-Funktion Ganzzahlen aus der Datei einlesen.

Variante mit `strtok`

Um die Gruppen an Zahlen aus der Datei auszulesen, müssen wir zuerst die Datei lesend mittels `fopen()` öffnen und prüfen, ob das Öffnen erfolgreich war. Darauffolgend können wir eine Zeile aus der Datei einlesen. Falls wir davon ausgehen, dass es nur eine Zeile ist und diese nicht länger als maximal 256 Zeichen ist, reicht ein einzelner Aufruf der `fgets()`-Funktion, um die Zeile in den temporären Puffer `buf` einzulesen.

Jetzt können wir den Inhalt der Zeile interpretieren. Wenn wir wissen, dass die einzelnen Zahlen mittels Leerzeichen voneinander getrennt sind, können wir die `strtok()`-Funktion aus dem `string.h`-Header verwenden, um immer einzelne Tokens einzulesen.

Der erste Aufruf von `strtok()` muss auf den Beginn der Zeile zeigen. Das Token (also hier die erste Zahl) kann dann durch den Pointer `p` ausgelesen und mittels der `atoi()`-Funktion in eine Ganzzahl umgewandelt werden. Folgende Aufrufe von `strtok()` müssen dann jedoch immer als erstes Argument `NULL` nutzen, um weiter im String voranzuschreiten.

Wenn wir wissen, wie viele Zahlen eine Gruppe umfasst, können wir in einer darauffolgenden for-Schleife entsprechend viele Zahlen einlesen, aufsummieren und schließlich den Durchschnitt berechnen.

Die größte Schwierigkeit könnte darin bestehen, dies kontinuierlich so lange für Gruppen von Zahlen durchzuführen, bis das Zeilenende erreicht ist. Wenn wir `strtok()` einsetzen, können wir uns zunutze machen, dass die Funktion am Zeilenende `NULL` zurückliefert und wir dementsprechend den Aufruf in einer **`while`**-Schleife anwenden können.

```c
#include <stdio.h>
#include <string.h> // für strtok()
```

```c
#include <stdlib.h> // für atoi()

int main(void) {
    char *filename = "08_group_of_numbers.txt";
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Fehler beim Öffnen der Datei\n");
        return -1;
    }

    char buf[256];
    if (fgets(buf, 255, fp) == NULL) {
        printf("Fehler beim Einlesen der Zeile");
        return -2;
    }

    int groupLength;
    int groupSum;
    double groupAverage;
    char *p;
    p = strtok(buf, " ");
    while (p != NULL) {
        groupLength = atoi(p);
        groupSum = 0;
        groupAverage = 0;
        for (int i = 0; i < groupLength; i++) {
            p = strtok(NULL, " ");
            if (p == NULL) {
                printf("Unerwartetes Ende der Zeile\n");
                return -3;
            }
            groupSum += atoi(p);
        }
        groupAverage = (double)groupSum / groupLength;
        printf("Gruppe mit %d Elementen: %.2lf\n", groupLength,
        ↪  groupAverage);

        // Bereite die nächste Gruppe vor
        p = strtok(NULL, " ");
    }
```

```
    fclose(fp);
}
```

Statt eine statische Puffergröße von 256 Zeichen festzulegen, könnten wir zuerst auslesen, wie lang die Zeile in der Datei ist und daraufhin entsprechend viel dynamischen Speicher allokieren:

```
// Ermittle die Länge der Zeile
fseek(fp, 0, SEEK_END);
long length = ftell(fp);
fseek(fp, 0, SEEK_SET);

// Allokiere Speicher für die Zeile
char *buf = (char *)malloc((unsigned long)length + 1);
if (buf == NULL) {
    printf("Speicherzuweisungsfehler\n");
    fclose(fp);
    return -1;
}

if (fgets(buf, (int)length + 1, fp) == NULL) {
    printf("Fehler beim Einlesen der Zeile\n");
    free(buf);
    fclose(fp);
    return -2;
}

// [...]

free(buf);
fclose(fp);
```

Variante mit `fscanf`

Die alternative Lösung nutzt die `fscanf()`-Funktion, um die Zahlen direkt aus der Datei zu lesen. Zuerst wird `fscanf()` in einer while-Schleife verwendet, um die Länge jeder Gruppe einzulesen. Innerhalb dieser Schleife wird eine for-Schleife mit einer Anzahl von Durchläufen gleich der Gruppenlänge verwendet. In jedem Durchlauf dieser for-Schleife rufen wir `fscanf()` erneut auf, um den nächsten Ganzzahlwert aus der Datei einzulesen. Bei jedem Aufruf von `fscanf()` bewegt sich der interne »*File*

Position Pointer« weiter, sodass kontinuierlich durch die Datei gelesen wird. Dieser Ansatz eliminiert die Notwendigkeit, die gesamte Zeile zuerst in einen Puffer zu lesen und dann zu zerlegen

```c
#include <stdio.h>
#include <stdlib.h> // für malloc und free

int main(void) {
    char *filename = "08_group_of_numbers.txt";
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Fehler beim Öffnen der Datei\n");
        return -1;
    }

    int groupLength;
    int number;
    int groupSum;
    double groupAverage;

    while (fscanf(fp, "%d", &groupLength) == 1) {
        groupSum = 0;
        groupAverage = 0.0;
        for (int i = 0; i < groupLength; i++) {
            if (fscanf(fp, "%d", &number) != 1) {
                printf("Fehler beim Lesen der Zahlen\n");
                fclose(fp);
                return -2;
            }
            groupSum += number;
        }
        groupAverage = (double)groupSum / groupLength;
        printf("Gruppe mit %d Elementen: %.2lf\n", groupLength,
        ↪  groupAverage);
    }

    fclose(fp);
    return 0;
}
```

9. Finde die jüngste Person

Die Funktion `findYoungestPerson()` nimmt zwei Parameter entgegen: einmal einen Pointer auf ein Array von Personen und einmal einen Integer, der die Größe des Arrays angibt. Machen Sie sich klar, dass das erste Funktionsargument auch wie folgt hätte deklariert werden können:

`Person **persons`

Die Funktion gibt einen Pointer auf eine `Person`-Struktur zurück – nämlich genau diejenige mit dem niedrigsten Alter.

Innerhalb der Funktion findet zuerst eine Plausibilitätsprüfung der übergebenen Größe (`size`) statt. Falls die Größe kleiner oder gleich 0 ist, gibt die Funktion NULL zurück, was bedeutet, dass kein gültiges Ergebnis gefunden werden kann.

Der Pointer `youngest` wird initialisiert und auf das erste Element des Arrays gesetzt. Dies dient als Startpunkt für den Vergleich, um die jüngste Person zu finden. Die Schleife durchläuft jedes Element des Arrays. Für jedes Element wird überprüft, ob das Alter (`age`) des aktuellen Elements (`persons[i]`) geringer ist als das Alter der derzeit jüngsten Person (`youngest`). Wenn dies der Fall ist, wird `youngest` aktualisiert, um auf das aktuelle Element zu zeigen, da dieses Element eine jüngere Person repräsentiert.

Nachdem das gesamte Array durchlaufen wurde, gibt die Funktion den Pointer `youngest` zurück. Dieser zeigt auf die Struktur der jüngsten Person im Array.

```
Person *findYoungestPerson(Person *persons[], int size) {
    if (size <= 0) {
        return NULL;
    }

    Person *youngest = persons[0];
    for (int i = 0; i < size; i++) {
        if (persons[i]->age < youngest->age) {
            youngest = persons[i];
        }
    }

    return youngest;
}
```

10. Erstelle eine verkettete Liste

Datenstruktur `Node`

Die Datenstruktur `Node` soll zwei Elemente enthalten: einmal einen Integer namens `data` und einmal einen Pointer auf einen anderen Knoten (vom Typ `Node`).

Um diese Struktur als einen Typ festzulegen, verwenden wir **`typedef`**. Hier müssen wir lediglich darauf achten, dass der neue Typ noch nicht innerhalb der Deklaration verwendet werden kann, so dass das Schlüsselwort **`struct`** noch innerhalb der Deklaration vor `Node *next` aufgeführt werden muss.

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

Datenstruktur `List`

Die Datenstruktur `List` soll nur ein Element enthalten, nämlich einen Pointer auf einen Knoten. Dieser Pointer ist sozusagen der `head`-Pointer, der immer auf den Anfang der Liste zeigt.

```
typedef struct {
    Node *head;
    int size;
} List;
```

Funktion `initList()`

Die `initiList()`-Funktion bekommt einen Pointer auf eine Liste übergeben und initialisiert die Werte `head` und `size` entsprechend auf die Werte `NULL` und `0`.

```
void initList(List *list) {
    list->head = NULL;
    list->size = 0;
}
```

Funktion addNode()

Die `addNode()`-Funktion allokiert zu Beginn dynamisch Speicher vom Heap für den neu einzufügenden Knoten. Über den von `malloc()` zurückgegebenen Pointer können die beiden Elemente dieses Knotens belegt werden. Das Element `newNode->data` erhält den Datumswert, der an die Funktion übergeben wurde; das Element `newNode->next` wird auf `NULL` gesetzt, da dieser neue Knoten ja ans Ende der Liste gelegt werden soll.

Jetzt geht es darum, zu ermitteln, ob dies der erste Knoten in der Liste ist oder nicht. Falls es sich um den ersten Knoten handelt, können wir einfach den `head`-Pointer aus der Liste auf diesen neuen Knoten zeigen lassen.

Andernfalls müssen wir bis ans Ende der Liste laufen und den `next`-Pointer des letzten Knotens auf den neuen Knoten `newNode` zeigen lassen. Schließlich noch den Zähler `size` der Liste um Eins erhöhen.

```
void addNode(List *list, int data) {
    Node *newNode = malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Fehler bei der Speicherallokation\n");
        return;
    }
    newNode->data = data;
    newNode->next = NULL;

    if (list->head == NULL) {
        list->head = newNode;
    } else {
        Node *current = list->head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }

    list->size++;
}
```

Funktion `printList()`

Die `printList()`-Funktion bekommt einen Pointer auf eine Liste übergeben und durchläuft alle Knoten der Liste bis zum Ende. Für jeden Knoten gibt sie entsprechend den Index und den Wert (`current->data`) auf dem Bildschirm aus.

```
void printList(const List *list) {
    Node *current = list->head;
    int i = 1;
    while (current != NULL) {
        printf("Element %d: data = %d\n", i, current->data);
        current = current->next;
        i++;
    }
}
```

Funktion `freeList()`

Die `freeList()`-Funktion bekommt einen Pointer auf eine Liste übergeben und durchläuft alle Knoten der Liste bis zum Ende. Für jeden Knoten sichert sie sich temporär den Zeiger auf diesen Knoten in einer temporären Variable, bevor sie den Pointer auf den nächsten Knoten ändert und den Speicher für den zu löschenden Knoten freigibt.

Wenn dies für alle Knoten in der List durchgeführt wurde, kann die Liste wieder auf die Initialwerte zurückgesetzt werden.

```
void freeList(List *list) {
    Node *current = list->head;
    while (current != NULL) {
        Node *temp = current;
        current = current->next;
        free(temp);
    }
    list->head = NULL;
    list->size = 0;
}
```

Funktion `removeNodesWithValue()`

Die `removeNodesWithValue()`-Funktion soll zwei Eingabeparameter besitzen: einmal einen Pointer auf die Liste, die verändert werden soll und einmal einen Integer, der angibt, welche Knoten mit dem entsprechenden Wert in ihrem `data`-Element gelöscht werden sollen.

Wir durchlaufen die Liste wieder vom Anfang bis zum Ende mittels einer while-Schleife. Innerhalb der while-Schleife prüfen wir für jeden Knoten, ob dessen `data`-Element dem an die Funktion übergebenen Wert entspricht. Ist das der Fall müssen wir den Knoten aus der Liste löschen.

Wir nutzen in dieser Lösung einen Hilfs-Pointer `prev`, der auf den Knoten vor `current` zeigen soll. Wenn `prev` `NULL` ist, bedeutet dies, dass der zu entfernende Knoten der Kopf der Liste ist. In diesem Fall wird der Kopf der Liste auf den nächsten Knoten nach `current` gesetzt. Wenn `prev` nicht `NULL` ist, wird der `next`-Pointer von `prev` auf den Knoten nach `current` gesetzt, wodurch `current` aus der Kette der Knoten entfernt wird.

Anschließend wird der `temp` verwendet, um den Pointer auf den zu entfernenden Knoten zu speichern; `current` kann dann zum nächsten Knoten bewegt werden, wonach der Speicher des entfernten Knotens freigegeben werden kann. Anschließend wird der Zähler `size` entsprechend um Eins dekrementiert.

Für den Fall, dass das `data`-Element nicht den gesuchten Wert hat, wird `prev` auf `current` gesetzt und `current` bewegt sich zum nächsten Knoten.

```
void removeNodesWithValue(List *list, int value) {
    Node *current = list->head;
    Node *prev = NULL;

    while (current != NULL) {
        if (current->data == value) {
            if (prev == NULL) {
                list->head = current->next;
            } else {
                prev->next = current->next;
            }
            Node *temp = current;
            current = current->next;
            free(temp);
            list->size--;
        } else {
            prev = current;
            current = current->next;
        }
```

```
    }
}
```

11. Duplikate aus einem Array entfernen

Grundsätzlich gibt es verschiedene Ansätze, um die Duplikate aus einem Array zu entfernen: wir könnten zuerst einen neuen Speicherbereich allokieren, in den wir die Unikate des ursprünglichen Arrays kopieren. Anschließend könnten wir den Speicher des neuen Arrays entsprechend der neuen Größe anpassen und den Speicher auf das alte Array freigeben.

Oder wir könnten das ursprüngliche Array direkt im Speicher verändern und die Größe des Arrays entsprechend anpassen. Das wäre eine effizientere Lösung, die im Folgenden vorgestellt wird.

Die Funktion `removeDuplicates()` akzeptiert zwei Parameter: einen *Doppel-Pointer* auf ein Array (`int **array`) und einen Pointer auf die Größe dieses Arrays (`int *size`). Sie gibt keinen Wert zurück, da sie das Array direkt im Speicher modifiziert.

Der Einsatz eines Doppel-Pointers ist hier entscheidend. Bei der Reduzierung der Größe des Arrays könnte die `realloc()`-Funktion eine neue Speicheradresse zuweisen. Mit einem einfachen Pointer (also nur `int *array`) könnten wir zwar die Elemente des Arrays manipulieren, jedoch wäre es uns nicht möglich, die Adresse des Zeigers selbst zu ändern, die außerhalb der Funktion `removeDuplicates()` definiert ist. Der Doppel-Pointer ermöglicht es uns, genau diese Adresse zu aktualisieren. So stellen wir sicher, dass nach einer erfolgreichen `realloc()`-Operation der in `main()` oder einer anderen aufrufenden Funktion verwendete Zeiger auf die korrekte, möglicherweise neue Speicheradresse des nun verkleinerten Arrays zeigt.

Zuerst wird überprüft, ob die Größe des Arrays kleiner als 2 ist. Falls dies der Fall ist, können wir direkt aus der Funktion heraus springen, da es keine Duplikate geben kann.

Die neue Größe des Arrays wird mittels `newSize = 1` mit Eins initialisiert, da das erste Element des (sortierten) Arrays immer das erste Element bleiben wird. Die for-Schleife durchläuft das Array beginnend vom zweiten Element (Index 1). Für jedes Element überprüft sie, ob es gleich dem vorherigen Element im »neuen« Array ist. Das vorherige Element im »neuen« Array bleibt dabei immer an der Position `newSize - 1`. Wenn ein Element nicht gleich seinem Vorgänger ist (d.h. kein Duplikat des vorherigen Elements), wird es in das Array an der aktuellen Position `newSize` kopiert und `newSize` wird inkrementiert.

Nachdem die for-Schleife durchlaufen wurde, wird der Speicher des Arrays auf die neue Größe angepasst. Hierfür wird die `realloc()`-Funktion aus dem `stdlib.h`-Header verwendet. `realloc()` kann dabei entweder die gleiche Adresse zurückgeben oder eine neue. Wenn `realloc()` eine neue Adresse zurückgibt, repräsentiert diese

den verkleinerten Speicherbereich und der ursprüngliche Speicher wird automatisch freigegeben.

Falls `realloc()` fehlschlägt, wird der ursprüngliche Speicherbereich hinter `*array` nicht freigegeben. Stattdessen gibt die Funktion `NULL` zurück, um zu signalisieren, dass die Größenänderung nicht erfolgreich war.

Falls `realloc()` erfolgreich ist und eine neue oder gleiche Adresse zurückgibt, aktualisieren wir den ursprünglichen Array-Zeiger (`*array`) auf diese (möglicherweise neue) Adresse. Dies ist wichtig, da sich die Adresse des Arrays geändert haben könnte. Nur im Erfolgsfall wird die Größe (`*size`) auf `newSize` aktualisiert, um die neue Anzahl der Elemente im Array widerzuspiegeln.

```
void removeDuplicates(int **array, int *size) {
    if (*size < 2) {
        return;
    }

    int newSize = 1;
    for (int i = 1; i < *size; i++) {
        if ((*array)[i] != (*array)[newSize - 1]) {
            (*array)[newSize] = (*array)[i];
            newSize++;
        }
    }

    int *tempArray = realloc(*array, (size_t)newSize *
    ↪  sizeof(int));
    if (tempArray != NULL) {
        // Zeiger und Größe nur aktualisieren,
        // wenn realloc() erfolgreich war
        *array = tempArray;
        *size = newSize;
    }
}
```

12. Datenstruktur sortieren

Funktion `createPlayer()`

Die Funktion `createPlayer()` soll dynamisch Speicher für einen Spieler allokieren. Dazu müssen wir mittels `malloc` entsprechend Speicher reservieren:
`Player* player = malloc(sizeof(Player));`

Sofern die Allokation erfolgreich war, können die Daten initialisiert werden. Die Initialisierung von Zeichenketten erfolgt wie gewohnt nicht einfach mittels des Zuweisungsoperators, sondern indem die Zeichen von einem Speicherbereich (Quelle) in einen anderen Speicherbereich (Ziel) kopiert werden.

Dies kann beispielsweise mittels der `strncpy()`-Funktion erfolgen. Hierbei müsste man dann allerdings sicherstellen, dass der Puffer nach dem Kopieren der Zeichenkette manuell null-terminiert wird, falls die Quelle länger als der Zielpuffer ist. Beispielsweise wie folgt:

```
strncpy(player->firstName, firstName, NAMELENGTH - 1);
player->firstName[NAMELENGTH - 1] = '\0';
```

Eine gute Alternative dazu ist die `snprintf()`-Funktion, die sicherstellt, dass die Zielzeichenkette immer null-terminiert ist, solange die Größe des Puffers größer als 0 ist.

```
Player* createPlayer(const char firstName[], const char lastName[],
↪   int age) {
    Player* player = malloc(sizeof(Player));
    if (player == NULL) {
        fprintf(stderr, "Fehler: Speicherallokation
        ↪  fehlgeschlagen\n");
        return NULL;
    }

    snprintf(player->firstName, NAME_LENGTH, "%s", firstName);
    snprintf(player->lastName, NAME_LENGTH, "%s", lastName);
    player->age = age;

    return player;
}
```

Funktion `printPlayer()`

Die Funktion `printPlayer()` empfängt einen Pointer auf ein `Player`-Objekt. Über diesen Pointer können wir auf die Strukturelemente zugreifen und die einzelnen Elemente in gewünschter Weise per `printf()` auf dem Bildschirm ausgeben.

```
void printPlayer(Player *player) {
    printf("%s %s, %d Jahre\n", player->firstName,
     ↪  player->lastName, player->age);
}
```

Funktion `comparePlayerByLastName()`

Die Funktion `comparePlayerByLastName()` ist so konzipiert, dass sie durch eine der Sortierfunktionen (bspw. `qsort`) zum Vergleich von zwei Elementen genutzt werden kann. Grundsätzlich ist es möglich, eine beliebige Vergleichsfunktion zu erstellen, die die Vornamen, Nachnamen oder das Alter aufsteigend oder absteigend miteinander vergleicht.

Um einen Vergleich nach Nachnamen durchzuführen, müssen wir das entsprechende Strukturelement der beiden übergebenen Objekte A und B auswerten. Wie in der Aufgabenstellung beschrieben, handelt es sich bei den beiden Pointern `pa` und `pb` jeweils um einen Pointer auf ein Array-Element des `teams`-Arrays, also jeweils um einen Pointer auf einen Pointer auf ein `Player`-Objekt.

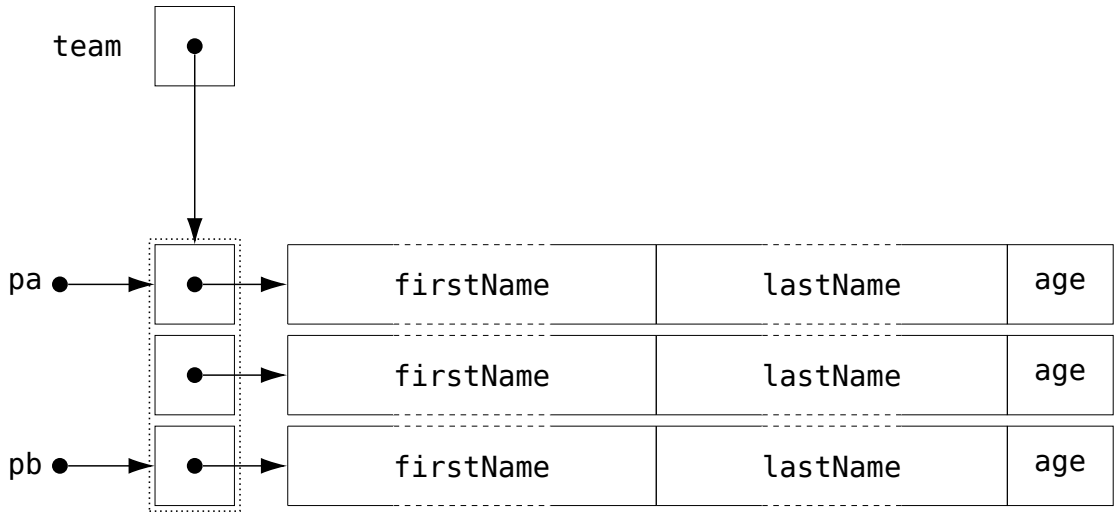


Abbildung 12.1.: *Darstellung der beiden Pointer `pa` und `pb` auf jeweils ein Array-Element des `teams`-Array*

Aus diesem Grund müssen wir diese Pointer als `Player**` interpretieren, so dass wir durch eine Dereferenzierung einen Pointer auf ein `Player`-Objekt erhalten:
`const Player *A = *(const Player**)pa;`

Sobald wir Pointer auf die beiden zu vergleichenden `Player`-Objekte haben, können wir deren Nachnamen mittels der `strcmp()`-Funktion lexikografisch vergleichen.

```
int comparePlayerByLastName(const void *pa, const void *pb) {
    const Player *A = *(const Player**)pa;
    const Player *B = *(const Player**)pb;

    return strcmp(A->lastName, B->lastName);
}
```

Funktion `freeTeam()`

Die Funktion `freeTeam()` soll dazu dienen, den dynamisch allokierten Speicher in geordneter Weise für ein ganzes Team wieder freizugeben. Im Gegensatz zur Allokation des Speichers müssen wir jetzt in ungekehrter Reihenfolge vorgehen, d. h. zuerst müssen wir den Speicher jedes einzelnen `Player`-Objekts freigeben und im Anschluss den Speicher des Arrays, in dem die Pointer auf die `Player`-Objekte abgelegt wurden.

```
void freeTeam(Player **team, int numPlayers) {
    for (int i = 0; i < numPlayers; i++) {
        free(team[i]);
    }
    free(team);
}
```

Diese Reihenfolge muss eingehalten werden, denn hätten wir zuerst den Speicher des Arrays freigegeben, hätten wir die Pointer auf die einzelnen `Player`-Objekte verloren.