Übung 6

HfTL Hochschule für Telekommunikation Leipzig University of Applied Sciences

Erarbeitet von: M.Eng. Michael Finsterbusch
Modulverantwortlicher: Prof. Dr. rer. nat. Matthias Krause
Stand: 3. Oktober 2017

Ziel der Übung ist das Kennenlernen und Vertiefen von:

- Rekursion
- File I/O
- Arrays
- Schleifen
- Kontrollstrukturen

Abgabe:
- http://praktomat.hft-leipzig.de unter „Tutorial: DKMI/DAI-17 C-Progr.“
- sämtliche Abgabemodalitäten sind im Praktomat hinterlegt

Aufgaben

Gegeben ist die Header-Datei *uebung6.h*, in der Funktionsdeklarationen enthalten sind, sowie die Datei *uebung6.c*, in der die Funktionen implementiert werden. Kopieren Sie diese in Ihr Arbeitsverzeichnis. Die Funktionen sollen entsprechen den folgenden Vorgaben implementiert werden. Um die korrekte Funktionsweise Ihrer Implementation zu testen, verwenden Sie die Funktionen in der `main()`-Funktion, die in der Datei *main.c* implementiert werden soll.

1. Implementieren Sie die Funktion `fak()` in der Datei *uebung6.c* zur Berechnung der Fakultät. Realisieren Sie die Funktion rekursiv:

$$n! = \begin{cases} 1, & \text{wenn } n = 0 \\ n \cdot (n-1)!, & \text{wenn } n > 0 \end{cases}$$

2. Implementieren Sie die Funktion `fak_it()` in der Datei *uebung6.c* zur Berechnung der Fakultät. Realisieren Sie die Funktion iterativ:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{k=1}^{n} k$$

3. Implementieren Sie die Funktion `fib()` in der Datei *uebung6.c* zur Berechnung der Fibonacci-Folge. Die Fibonacci-Folge ist durch das rekursive Bildungsgesetz:

   `fib(n) = fib(n-1) + fib(n-2)` für n $>$ 2 mit den Anfangswerten fib(1) = fib(2) = 1 definiert.

4. Implementieren Sie die Funktion `maze_print()` mit der ein Labyrinth (engl. maze) auf die Konsole ausgegeben werden soll. Der Aufbau des Labyrinths ist in einem zweidimensionalen Array enthalten. Es besteht aus Wänden, Gängen, einem Startpunkt und einem Ziel. Das Array hat eine feste Größe von 100x100, aber die Dimensionen eines gegebenen Labyrinths können auch kleiner als 100x100 sein. Die Indizes des Arrays (`labyrinth[x][y]`) können als Koordinaten im Labyrinth interpretiert werden. Dabei ist der Punkt (0,0) in der linken oberen Ecke und der Punkt (99,99) rechts unten. Die verschiedenen Zustände die ein Punkt im Labyrinth annehmen kann sind:

```
enum maze_field_type {
    MAZE_WALL = 0,        /* Wand */
    MAZE_WALKWAY,         /* Gang */
    MAZE_START,           /* Start */
    MAZE_FINISH,          /* Ziel */
    [...]
};
```

In der Ausgabe soll jede Koordinate im Labyrinth durch ein Zeichen dargestellt werden. Die Wand (`MAZE_WALL`) mit '`#`', ein Gang (`MAZE_WALKWAY`) mittels Leerzeichen (' '), Start (`MAZE_START`) durch ein '`S`' und das Ziel (`MAZE_FINISH`) mit '`F`'. Ein einfaches 5x11 Labyrinth könnte zum Beispiel so aussehen:

```
###########
##  ###    ##
S     # #   F
## #    ####
###########
```

5. Implementieren Sie die Funktion (`maze_from_file()`), mit der ein Labyrinth aus einer Datei ausgelesen werden kann. Die erste Zeile der Datei enthält die Dimension des Labyrinths. Die folgenden Zeilen enthalten das Layout des Labyrinths:

```
5x11
WWWWWWWWWWW
WW  WWW    WW
S     W W   F
WW  W    WWWW
WWWWWWWWWWW
```

Hinweis: verwenden Sie die Funktion `fgets()`, um die Datei Zeilenweise einzulesen und zu verarbeiten.

Ein Labyrinth kann aus folgenden Symbolen aufgebaut sein:

'W': `MAZE_WALL`

' ': `MAZE_WALKWAY`

'S': `MAZE_START`

'F': `MAZE_FINISH`

Alle weiteren Symbole sind auch als `MAZE_WALL` zu werten. Das Labyrinth hat eine maximale Größe von 100x100. Lesen Sie das Labyrinth ein und speichern es in dem übergebenen Array. Verwenden Sie eine `switch`-Anweisung, um die verschiedenen Symbole zu unterscheiden.

Die Funktion soll folgende Fehlerwerte zurückgeben:

- **0** Bei Erfolg
- **1** Fehler beim Öffnen der Datei
- **2** Fehler beim Lesen der Datei

3 Labyrinth zu groß

4 Sonstige Fehler

6. Implementieren Sie die Funktion `maze_start_pos()` zum Bestimmen des Startpunktes in einem Labyrinth. Das Labyrinth ist ein zweidimensionales Array, dass wie in den vorangegangenen Aufgaben definiert und initialisiert ist. Ermitteln Sie die Koordinaten und geben Sie diese mit Hilfe der Struktur `struct Point` zurück. Ist in dem Labyrinth kein Startpunkt definiert, sollen die Koordinaten den Wert (-1,-1) enthalten.

7. Implementieren Sie die Funktion `maze()`, um einen Weg durch ein gegebenes Labyrinth zu finden.

   Der Weg durch ein Labyrinth kann mit der Hilfe eines *Backtracking*-Algorithmus ermittelt werden. Bei diesem wird mittels Versuch und Irrtum das Labyrinth durchlaufen. Es wird der erst beste Weg gewählt, führt dieser zum Ziel, ist das Problem gelöst, andernfalls geht man einfach zurück und nimmt einen anderen Weg.

   In der zweidimensionalen Darstellung des Labyrinths gib es vier Richtungen in die man sich bewegen kann: Oben, Unten, Links und Rechts. In diese Richtungen kann man sich nur bewegen, wenn der Weg frei, d.h. vom Typ `MAZE_WALKWAY`, ist. Ist es möglich einen Schritt nach oben zu gehen, wird `labyrinth[x-1][y]=MAZE_STEP` gesetzt. Anschließend wird die Funktion `maze()` rekursiv aufgerufen, um den nächsten Schritt auszuführen. Die Funktion `maze()` gibt als Rückgabewert `0` zurück, wenn das Ziel erreicht ist. Endet der Weg in einer Sackgasse, wird `1` zurückgegeben, der falsche Weg wird mit `labyrinth[x-1][y]=MAZE_WRONG` markiert und es muss versucht werden nach Unten, Links oder nach Rechts zu gehen. Im Programmablaufplan in Abbildung 1 ist dargestellt, wie geprüft wird, ob der Weg nach oben frei ist und wie ein Schritt nach oben ausgeführt wird. Dies muss mit entsprechenden Anpassungen auch für die anderen drei Bewegungsrichtungen implementiert werden. Aus diesen vier Teilen ergibt sich dann die Funktion `maze()`, siehe Abbildung 2.

   Gibt die Funktion `maze()`, aus der `main()` Funktion gerufen, den Wert `1` zurück, gibt es keinen Weg durch das Labyrinth. Ist der Rückgabewert `0`, ist im Array `labyrinth` der Weg durch das Labyrinth markiert und es sind auch alle gewählten Wege vermerkt, die nicht zum Ziel führten.

8. Erweitern Sie die Funktion `maze_print()` aus Aufgabe 4, sodass `MAZE_STEP` mittels '`+`' und `MAZE_WRONG` mit '`?`' dargestellt wird. Beispiel:

```
1  ###########
2  ##?#######+##
3  S++++#+#++F
4  ##?#+++#####
5  ###########
```

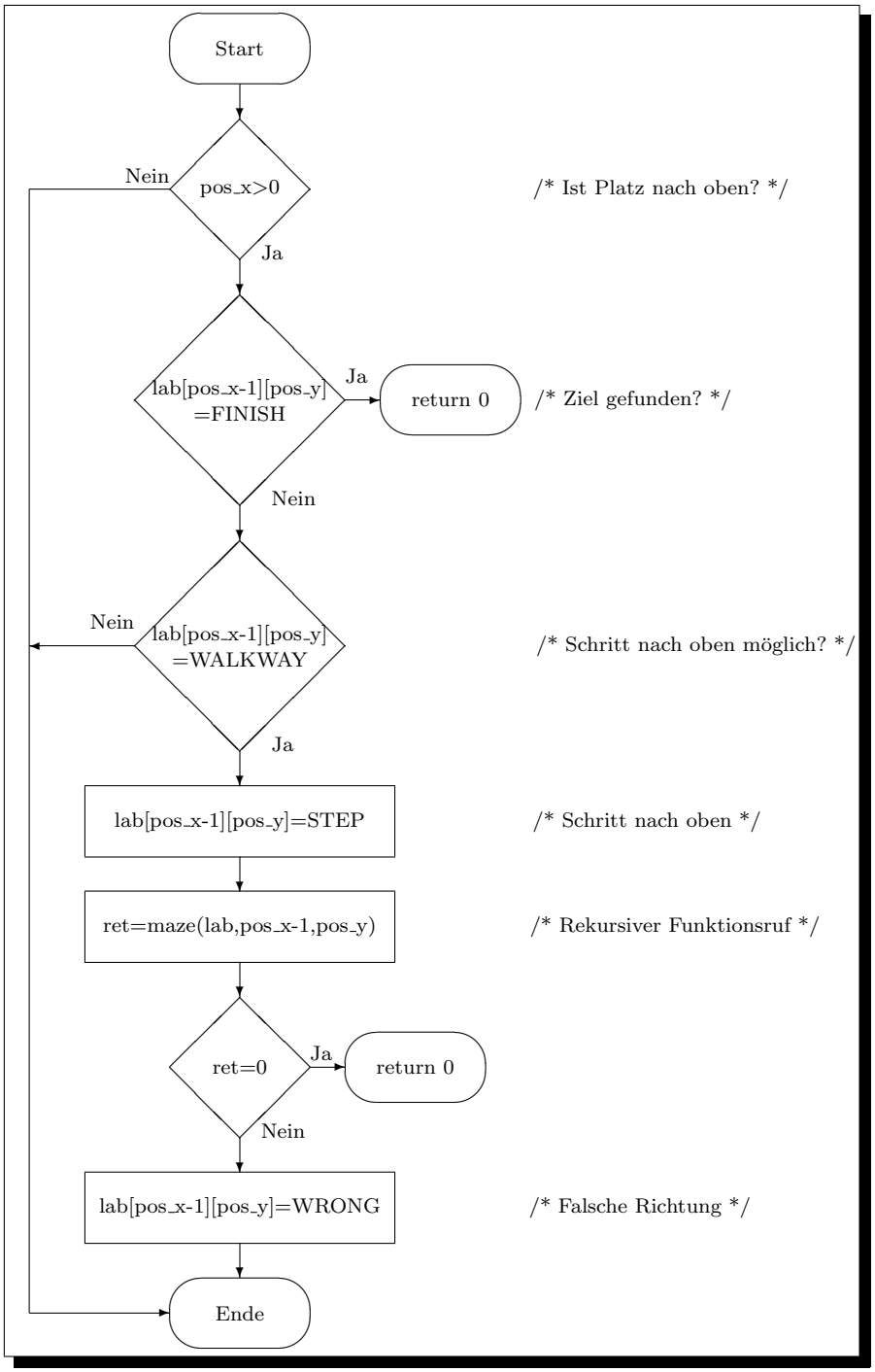

Abbildung 1: Programmablaufplan für den Funktionsteil `up`

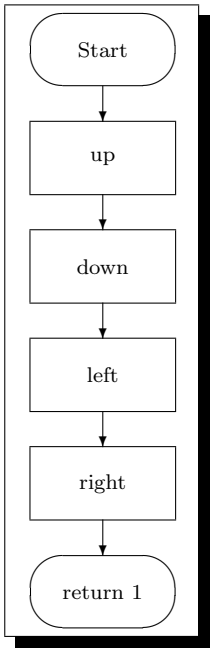

Abbildung 2: Programmablaufplan für die Funktion `maze()` bestehend aus den Funktionsteilen `up`, `down`, `left` und `right`.