

Übungspaket 22

Rekursive Funktionsaufrufe

Übungsziele:

1. Technische Voraussetzungen für rekursive Funktionsaufrufe
2. Umsetzung mathematisch definierter Rekursionen in entsprechende C-Programme
3. Bearbeitung rekursiver Beispiele

Skript:

Kapitel: [44](#) und [48](#) sowie insbesondere Übungspaket [19](#)

Semester:

Wintersemester 2025/26

Betreuer:

Benjamin, Thomas und Ralf

Synopsis:

Funktionen haben wir nun endlich im vorherigen Übungspaket geübt. Jetzt gibt es aber noch Funktionen, die sich selbst aufrufen. Dies wird als Rekursion bezeichnet und bereitet vielen Programmieranfängern größere Schwierigkeiten, obwohl die Rekursion technisch gesehen nichts besonderes ist. Um hier Abhilfe zu schaffen, werden wir uns in diesem Übungspaket von verschiedenen Seiten der Rekursion nähern.

Teil I: Stoffwiederholung

Aufgabe 1: Rekursion am Beispiel der Fakultät

In wohl den meisten Lehrbüchern wird die Rekursion am Beispiel der Fakultät eingeführt; so haben auch wir es in unserem Skript gemacht. Wiederhole die beiden möglichen Definitionen der Fakultät:

Iterativ:

Rekursiv:

Aufgabe 2: Technische Umsetzung der Rekursion

Was ist die wesentliche Voraussetzung für die technische Umsetzung der Rekursion in einer Programmiersprache?

Worin liegt die eigentliche Schwierigkeit der meisten Studenten?

Rekursive Funktionen weisen neben der Tatsache, dass sie sich selbst (direkt oder indirekt) aufrufen, ein ganz besonderes Charakteristikum auf. Welches ist das?

Teil II: Quiz

Regeln für die heutige Quizrunde

Diesmal geht es in den Quizfragen um das Verstehen rekursiv definierter Funktionen. Dies fällt, wie schon mehrmals gesagt, den meisten Programmieranfängern recht schwer. Eine gute Möglichkeit hier einen Schritt voranzukommen, besteht darin, dass man rekursive Funktionen mit ein paar Freunden und/oder Kommilitonen (bei einem Bier) spielt. Die Regeln sind wie folgt:

1. Jeder Mitspieler bekommt ein Blatt Papier, auf dem er seinen eigenen Stack-Frame verwaltet. Was alles steht doch gleich nochmal im Stack-Frame?
2. Einer der Mitspieler nimmt sich die Aufgabe und füllt seinen Stack-Frame aus, soweit die Informationen vorliegen.
3. Nach dem Ausfüllen des Stack-Frames beginnt der Spieler mit dem Abarbeiten des Algorithmus. Sollte er dabei auf einen rekursiven Funktionsaufruf stoßen, unterbricht er seine Arbeit und lässt diesen von einem Mitspieler bearbeiten.
4. Der neue Mitspieler erledigt alle oberen Punkte, bis er endgültig mit seiner Arbeit fertig ist. Zum Schluss übermittelt er das berechnete Ergebnis an seinen Auftraggeber, der anschließend die Kontrolle erhält.

Aufgabe 1: Berechnung der Fakultät

Funktion: Die Funktion `fakultaet()` sei wie folgt definiert:

```
1 int fakultaet( int n )
2   {
3       if ( n < 2 )
4           return 1;
5       else return n * fakultaet(n - 1);
6   }
```

Aufgabe: Welche Resultate werden für gegebene n produziert?

n	1	2	3	4	5
$n!$

Funktionalität: Was macht `fakultaet()`?

Aufgabe 2: Ausgabe I

Funktion: Die Funktion `prtb2()` sei wie folgt definiert:

```
1 void prtb2( int i )
2     {
3         if ( i >= 2 )
4             prtb2( i / 2 );
5         printf( "%d", i % 2 );
6     }
```

Aufgabe: Welche Ausgaben werden für gegebene n produziert?

n	1	2	3	4	5	6
<code>prtb2(n)</code>

Funktionalität: Was macht `prtb2(n)`?

Aufgabe 3: Ausgabe II

Funktion: Die Funktion `display()` sei wie folgt definiert:

```
1 void display( int i )
2     {
3         if ( i != 0 )
4         {
5             printf( "%d", i % 2 );
6             display( -(i/2) );
7         }
8     }
```

Aufgabe: Welche Ausgaben werden für gegebene n produziert?

n	15	63	255
<code>display(n)</code>

Funktionalität: Was macht `display(n)`?

Aufgabe 4: Ausgabe III

Funktion: Die Funktion `triline()` sei wie folgt definiert:

```
1 void triline( int i, int nr, int max )
2     {
3         if ( i == 0 && nr < max - 1 )
4             triline( 0, nr + 1, max );
5         if ( i < max - nr - 1 || i > max + nr - 1 )
6             printf( " " );
7         else printf( "%d", nr );
8         if ( i == 2*(max - 1) )
9             printf( "\n" );
10        else triline( i + 1, nr, max );
11    }
```

Aufgaben:

Aufruf	<code>triline(0,0,1)</code>	<code>triline(0,0,2)</code>	<code>triline(0,0,3)</code>
Ausgabe
	
		

Funktionalität: Was macht `triline()`?

Teil III: Fehlersuche

Aufgabe 1: Fehler bei der Summenberechnung

Eine alte mathematische Aufgabe ist die Berechnung der Summe $s = \sum_{i=1}^n$, für die der kleine Gauß eine schöne Formel gefunden hat. Starmathematiker DR. G. AUSS schlägt folgende Lösung vor:

```
1 int summe( int n )
2     {
3         return n + summe( n - 1 );
4     }
```

Doch was ist hier falsch? Beschreibe den Fehler und korrigiere das Programm entsprechend.

Teil IV: Anwendungen

In jeder der folgenden Aufgaben ist eine Funktion zu implementieren, die eine gestellte Aufgabe *rekursiv* lösen soll. Ihr solltet eure erarbeiteten Lösungen natürlich eintippen und mittels eines einfachen Hauptprogramms auch testen.

Aufgabe 1: Fibonacci-Zahlen

Schreibe eine Funktion, die die Fibonacci-Zahlen von 0 bis 10 berechnet. Die Fibonacci-Zahlen F_i sind wie folgt definiert: $F_0 = 0$, $F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$

Aufgabe 2: Ausgabe eines int-Arrays

Gegeben sei ein `int`-Array der Größe `size`. Schreibe eine Funktion `print_array()`, die die einzelnen Elemente rekursiv ausgibt. Beispiel: Sollte ein Array der Größe 4 die Elemente 0, 1, 815 und 4711 enthalten, sollte die Ausgaben 0, 1, 815 und 4711 lauten.

Aufgabe 3: Gespiegelte Ausgabe eines int-Arrays

Schreibe eine zweite Funktion `print_array_reverse()`, die die Array-Elemente von hinten nach vorne ausgibt. Bei obigem Beispiel soll die Ausgabe 4711, 815, 1 und 0 lauten.

Aufgabe 4: Ab und auf

Schreibe eine Funktion, die die Zahlen hinab bis 0 und wieder zurück ausgibt. Beispiel: Das Argument 3 soll dazu führen, dass die Zahlen 3 2 1 0 1 2 3 ausgegeben werden.

Aufgabe 5: Auf und ab

Schreibe eine Funktion, die die Zahlen hinauf bis zu einem gegebenen Zahlenwert und wieder zurück ausgibt. Eine beispielhafte Ausgabe lautet: 0 1 2 3 2 1 0