

ALGORITHMEN UND DATENSTRUKTUREN

ÜBUNG 5: FUNKTIONEN & PULSIERENDER SPEICHER

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 11. November 2021

WER BIN ICH?

- ▶ **Eric** Kunze
- ▶ `eric.kunze@tu-dresden.de`
- ▶ Telegram: `@oakoneric` bzw. `t.me/oakoneric`
- ▶ Fragen, Wünsche, Vorschläge, ...
- ▶ Website mit Material:
`https://oakoneric.github.io/aud21`
keine Garantie für Vollständigkeit/Korrektheit
- ▶ Anwesenheitsliste & Abmeldung bei Nichterscheinen
(weiterhin) per Mail oder Telegram

Deklaration einer Funktion:

```
<return_type> <function_name> (<argument_list>...);
```

- ▶ `return` Keyword zur Rückgabe eines Wertes
- ▶ `void` als `return_type`, wenn keine Rückgabe erfolgen soll

Funktionsaufruf:

```
<function_name>(<argument1>, <argument2>...);
```

Beispiel:

```
int square (int x) {  
    return x * x;  
}
```

```
int main () {  
    int y = square(4);  
}
```

DIE ACKERMANN-FUNKTION

“Die Ackermannfunktion ist eine 1926 von Wilhelm Ackermann gefundene, extrem schnell wachsende mathematische Funktion, mit deren Hilfe in der theoretischen Informatik Grenzen von Computer- und Berechnungsmodellen aufgezeigt werden können.”

Quelle: <https://de.wikipedia.org/wiki/Ackermannfunktion>

Definition von $\text{ack} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$$\text{ack}(0, y) = y + 1 \quad (y \geq 0)$$

$$\text{ack}(x, 0) = \text{ack}(x - 1, 1) \quad (x > 0)$$

$$\text{ack}(x, y) = \text{ack}(x - 1, \text{ack}(x, y - 1)) \quad (x, y > 0)$$

DIE ACKERMANN-FUNKTION

Definition von ack : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

$$\text{ack}(0, y) = y + 1 \quad (y \geq 0)$$

$$\text{ack}(x, 0) = \text{ack}(x - 1, 1) \quad (x > 0)$$

$$\text{ack}(x, y) = \text{ack}(x - 1, \text{ack}(x, y - 1)) \quad (x, y > 0)$$

einige Werte

$x \setminus y$	0	1	2	3	4	...	m
0	1	2	3	4	5	...	$m + 1$
1	2	3	4	5	6	...	$m + 2$
2	3	5	7	9	11	...	$2m + 3$
3	5	13	29	61	125	...	$8 * 2^m - 3$
4	13	65533	$2^{65536} - 3$	$\underbrace{2^{2^{\dots^2}}}_{m+3} - 3$

AUFGABE 1

```
1 #include <stdio.h>
2 int ack(int x, int y){
3     if ((x == 0) && (y >= 0))           return y + 1;
4     else if ((x > 0) && (y == 0))       return ack(x-1, 1);
5     else if ((x > 0) && (y > 0)){
6         return ack(x-1, ack(x, y-1)); }
7 }
8 int main() {
9     int x = 0, y = 0, a;
10    printf("\nAckermannfunktion\n");
11    printf("x = ");    scanf("%d", &x);
12    printf("y = ");    scanf("%d", &y);
13    a = ack(x,y);
14    printf("ack(%i,%i)=%i.\n", x, y, a);
15    return 0;
16 }
```

Deklaration (als EBNF): `ElementType Ident { [Number] } ;`

- ▶ `ElementType ...` Typ eines Eintrags
- ▶ `Number ...` Anzahl der Einträge
- ▶ `Ident ...` Bezeichner des Arrays

Deklaration (als EBNF): `ElementType Ident { [Number] } ;`

- ▶ `ElementType ...` Typ eines Eintrags
- ▶ `Number ...` Anzahl der Einträge
- ▶ `Ident ...` Bezeichner des Arrays

Beispiele:

- ▶ `Liste: int liste[5];`
- ▶ `Matrix: int matrix[3][4];`

Deklaration (als EBNF): `ElementType Ident { [Number] } ;`

- ▶ `ElementType ...` Typ eines Eintrags
- ▶ `Number ...` Anzahl der Einträge
- ▶ `Ident ...` Bezeichner des Arrays

Beispiele:

- ▶ `Liste: int liste[5];`
- ▶ `Matrix: int matrix[3][4];`

Initialisierungen:

- ▶ `int liste[5] = {2,7,0,-4,1};`
- ▶ `int matrix[3][4] = { {1,2,3,4}, {5,6,7,8}, {2,3,4,5} };`

Deklaration (als EBNF): `ElementType Ident { [Number] } ;`

- ▶ `ElementType ...` Typ eines Eintrags
- ▶ `Number ...` Anzahl der Einträge
- ▶ `Ident ...` Bezeichner des Arrays

Beispiele:

- ▶ `Liste: int liste[5];`
- ▶ `Matrix: int matrix[3][4];`

Initialisierungen:

- ▶ `int liste[5] = {2,7,0,-4,1};`
- ▶ `int matrix[3][4] = { {1,2,3,4}, {5,6,7,8}, {2,3,4,5} };`

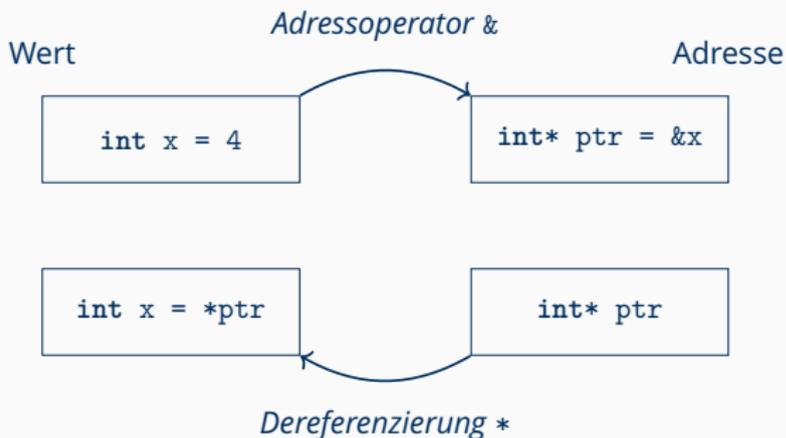
Zuweisungen: *Indizierung beginnt bei 0*

- ▶ `liste[2] = 16; ~> [2,7,16,-4,1]`
- ▶ `matrix[1][3] = 0; ~> $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 0 \\ 2 & 3 & 4 & 5 \end{pmatrix}$`

POINTER

Pointer-Type: `<base_type>* <pointer_name>`

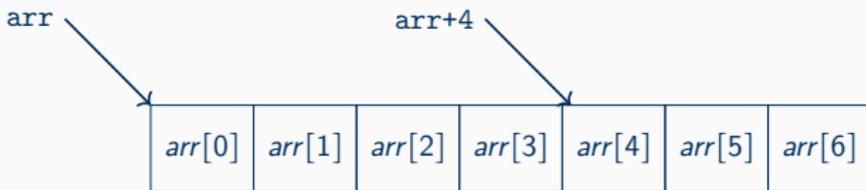
Wert einer Variable **vs.** Speicheradresse einer Variable



Nutzen: „Verlängerung“ Sichtbarkeit von Variablen (Veränderung innerhalb von Funktionen „speichern“), dynamische Datentypen

ARRAY-POINTER-DUALITÄT

Ein Array ist ein Pointer, der auf die erste Komponente des Arrays zeigt.



Pointerarithmetik: `ptr + i` zeigt auf die *i*-te Speicherzelle nach `ptr` (analog `ptr - i`, `ptr++` usw.)

Der Elementzugriff `arr[i]` ist gleichbedeutend mit `*(arr + i)`.

AUFGABE 1

```
1 void palindrom(char str[], int l, int *korrekt) {
2     int i = 0;
3     l = l - 1;
4     *korrekt = 1;
5
6     while (i < l && *korrekt) {
7         *korrekt = str[i] == str[l];
8         i = i + 1;
9         l = l - 1;
10    }
11 }
12
13 int main () {
14     char[...] str;
15     int korr;
16     int len;
17     ...
18     palindrom(str, len, &korr);
19 }
```

AUFGABE 2

```
1 #include <stdio.h>
2
3 void swoop(int a, int b) {
4     /* label 1 */
5     a = b;
6     b = a;
7     /* label 2 */
8 }
```

```
9 int main() {
10     int x = 3, y = 6;
11     /* label 3 */
12     swoop(x, y); /*$1*/
13     /* label 4 */
14     printf("x = %d, y = %d", x, y);
15     return 0;
16 }
```

AUFGABE 2 — TEIL (A)

Label	RM	1	2	3	4
label3	—	x 3	y 6		
label1	1			a 3	b 6
label2	1			a 6	b 6
label4	—	x 3	y 6		

AUFGABE 2 — TEIL (B)

```
1 #include <stdio.h>
2 void swap(int *a, int *a){
3     int tmp;
4     tmp = *a;
5     *a = *b;
6     *b = tmp;
7 }
8 int main() {
9     int x = 4, y = 6;
10    printf("x = %d, y = %d \n", x, y);
11    swap(&x, &y);
12    printf("x = %d, y = %d \n", x, y);
13    return 0;
14 }
```

Gültigkeitsbereiche von Objekten:

- ▶ Eine Funktion ist ab ihrer Deklaration bis zum Programmende sichtbar. Vorwärtsdeklarationen beachten!
- ▶ Ihre formalen Parameter jedoch nur innerhalb der Funktionsdefinition!
- ▶ Gibt es gleichlautende formale Parameter in verschiedenen Funktionen, müssen diese in der Tabelle natürlich unterschieden werden (z.B. durch „x in f“).
- ▶ Vorsicht bei Namenskonflikten: lokale Variablen überschreiben die Sichtbarkeit globaler Variablen.

Speicherprotokoll:

- ▶ Für jeden Funktionsaufruf werden erst die Parameter, dann die lokalen Variablen in Reihenfolge ihres Auftretens in der Umgebung notiert. Globale Variablen stehen ganz vorn.
- ▶ Variablennamen werden nur notiert, wenn die Variablen sichtbar sind. Globale Variablennamen werden immer notiert.
- ▶ Der Wert von nicht sichtbaren Variablen muss nur notiert werden wenn er sich ändert.
- ▶ Uninitialisierte Variablen werden mit Inhalt „?“ notiert.

AUFGABE 3

```
1 #include <stdio.h>
2 int a;
3
4 void g(int a, int *b);
5
6 void f(int *i, int j) {
7     /*label1*/
8     if (*i + j < a) {
9         *i = *i + 1;
10        f(i, j);           /*$1*/
11    }
12    /*label2*/
13 }
14
15 void g(int a, int *b) {
16     int i = 2;
17     /*label3*/
18     while (a != 1) {
19         f(&i, a);         /*$2*/
20         a = a / 2;
21         *b = *b + 1;
22         /*label4*/
23     }
24 }
25
26 int main() {
27     int x = 0;
28     scanf("%i", &a);
29     /*label5*/
30     g(a, &x);           /*$3*/
31     /*label6*/
32     return 0;
33 }
```

Gültigkeitsbereiche

Objektname	Gültigkeitsbereich
a	2 - 14 und 25 - 33
g	4 - 33
a, b in g	15 - 24
i in g	16 - 24
f	6 - 33
i, j in f	6 - 13
main	26 - 33
x in main	27 - 33

AUFGABE 3 — TEIL (B)

Label	RM	1	2	3	4	5	6	7	8	9	10	11	12
label5	-	a 7	x 0										
label3	3			a 7	b 2	i 2							
label1	2:3	a 7					i 5	j 7					
label2	2:3	a 7					i 5	j 7					
label4	3		1	a 3	b 2	i 2							
label1	2:3	a 7					i 5	j 3					
label1	1:2:3	a 7				3			i 5	j 3			
label1	1:1:2:3	a 7				4					i 5	j 3	
label2	1:1:2:3	a 7									i 5	j 3	
label2	1:2:3	a 7							i 5	j 3			
label2	2:3	a 7					i 5	j 3					
label4	3		2	a 1	b 2	i 4							
label6	-	a 7	x 2										