

Kapitel 7

Modularisierung

Modularisierung in der Küche

- Die Herstellung von Apfelkuchen ist die eigentliche Aufgabe. Das ist das **Hauptprogramm**.
- Die Herstellung von Hefeteig ist eine Teilaufgabe im Rahmen der Herstellung eines Apfelkuchens. Das ist eine **Funktion** oder ein **Unterprogramm**.
- Das Starten der Aktivität »Hefeteig erstellen« aus der Zubereitungsvorschrift von Apfelkuchen bezeichnen wir als einen Aufruf des Unterprogramms aus dem Hauptprogramm. Wir sprechen von einem **Unterprogrammaufruf** oder einem **Funktionsaufruf**.
- Zwischen Haupt- und Unterprogramm müssen beim Aufruf ganz bestimmte Informationen fließen, z. B. darüber, wie viel Hefeteig zu erstellen ist und ob dem Teig Zucker zugesetzt werden soll. Über den Austausch dieser Informationen muss zwischen Haupt- und Unterprogramm eine präzise Vereinbarung bestehen. Das Hauptprogramm muss wissen, welche Informationen das Unterprogramm benötigt und welche Ergebnisse es produziert. Eine solche Vereinbarung nennen wir eine **Schnittstelle**.
- Eine im Rahmen der Schnittstelle vereinbarte Einzelinformation, wie z.B. »Zuckerzugabe in Gramm«, nennen wir einen **Parameter**. Alle Parameter zusammen beschreiben die Schnittstelle. Ein Parameter, durch den Informationen vom Hauptprogramm zum Unterprogramm fließen, bezeichnen wir als **Eingabeparameter**. Einen Parameter, durch den Informationen vom Unterprogramm zum Hauptprogramm zurückfließen, bezeichnen wir als **Rückgabeparameter**.
- Konkrete, durch die Parameter der Schnittstelle fließende Daten (z. B. 100 Gramm Zuckerzugabe) bezeichnen wir als **Parameterwerte**. Entsprechend der Flussrichtung bezeichnen wir die Parameterwerte auch als **Eingabewerte** oder **Rückgabewerte**.

Rezept für Apfelkuchen

Zutaten:

600 Gramm Hefeteig
1,5 Kilo Äpfel
...

Zubereitung:

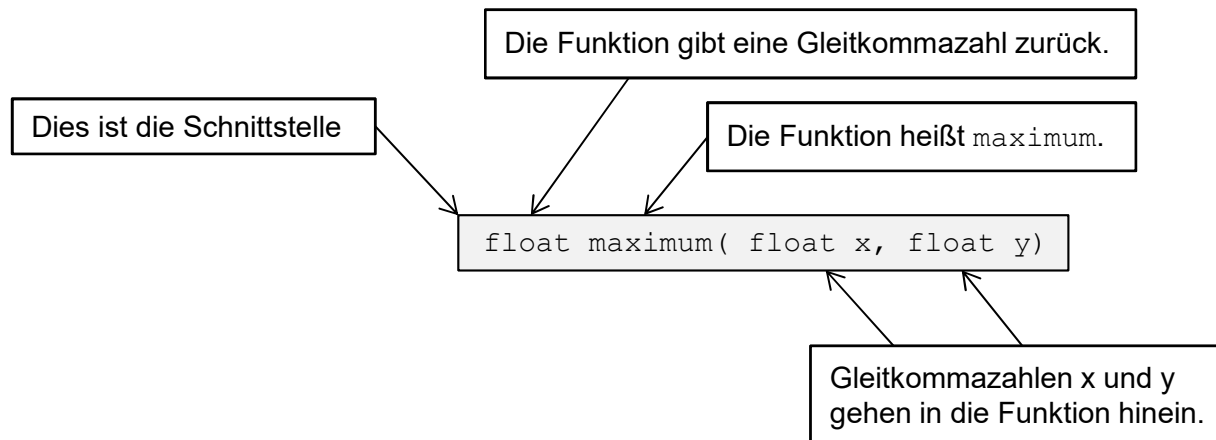
Bereiten Sie den Hefeteig nach Rezept zu und rollen diesen dann auf einer bemehlten Arbeitsfläche quadratisch aus. Geben Sie den Teig dann auf ein mit Backpapier ausgelegtes Blech und ziehen den Rand an jeder Seite hoch. Der Teig kann dann mit einem Küchentuch abgedeckt noch ein wenig stehen bleiben. In der Zwischenzeit die Äpfel schälen, entkernen und in schmale Spalten schneiden. ...

Funktionschnittstelle

An der **Schnittstelle** einer Funktion werden alle Informationen festgelegt, die zwischen Haupt- und Unterprogramm fließen.

Wenn wir eine Funktion erstellen wollen, die das Maximum von zwei Gleitkommazahlen bestimmen soll, so fließen an der Schnittstelle die folgenden Informationen:

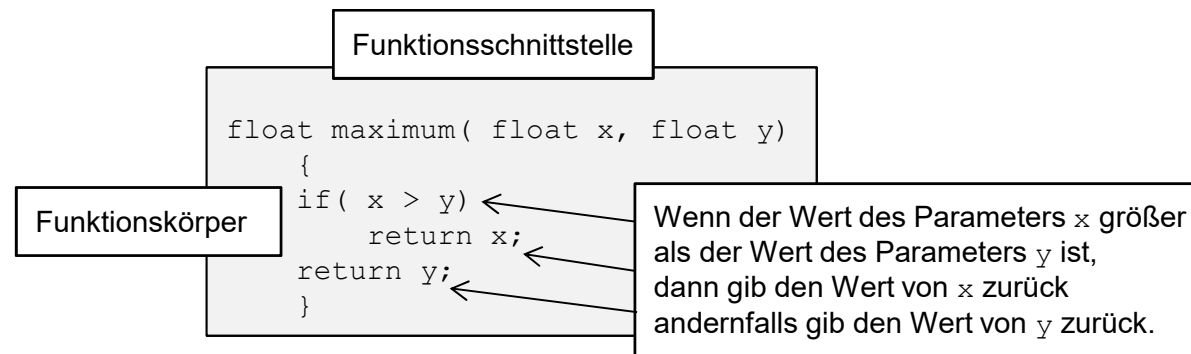
- In die Funktion hinein fließen zwei `float`-Zahlen (Eingabeparameter), von denen die größere zu bestimmen ist.
- Aus der Funktion heraus fließt eine `float`-Zahl (Rückgabeparameter), nämlich die größere der beiden hinein geflossenen Zahlen.



Wie die Funktion ihre Aufgabe erledigt, interessiert an der Schnittstelle nicht. Es geht nur um die Informationen, die sie benötigt, um ihre Aufgabe zu erledigen und die Informationen, die sie nach Erledigung der Aufgabe zurückgibt.

Funktionskörper

Im **Funktionskörper** wird festgelegt, wie die Funktion ihre Aufgabe erledigt. Der Funktionskörper enthält den Quellcode, der den Ablauf der Funktion festlegt:

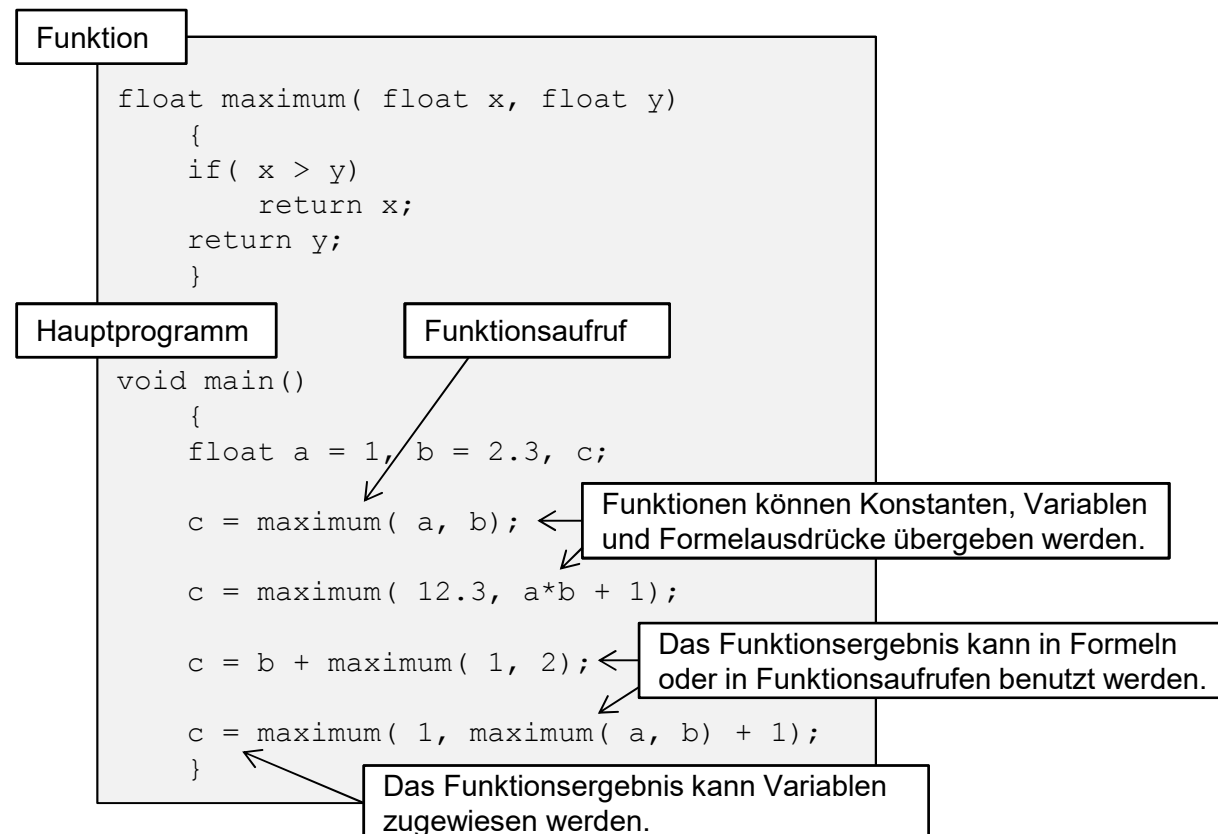


Eine `return`-Anweisung beendet die Funktion und gibt den Wert des hinter `return` stehenden Ausdrucks an das rufende Programm zurück.

Der Ergebnistyp des Ausdrucks (hier `float`) muss zum Rückgabetyp der Funktionsschnittstelle passen.

Funktionsaufruf

Funktionen können aus dem Hauptprogramm oder aus anderen Funktionen gerufen werden. Die aufrufende Funktion übergibt dazu die erforderlichen Parameter und erhält den von der Funktion ermittelten Rückgabewert:



Anzahl und Typen der beim Aufruf übergebenen Parameter und die Verwendung des Rückgabetyps müssen mit der Schnittstellenvereinbarung übereinstimmen.

Unterschiedliche Parameter und Returnwerte

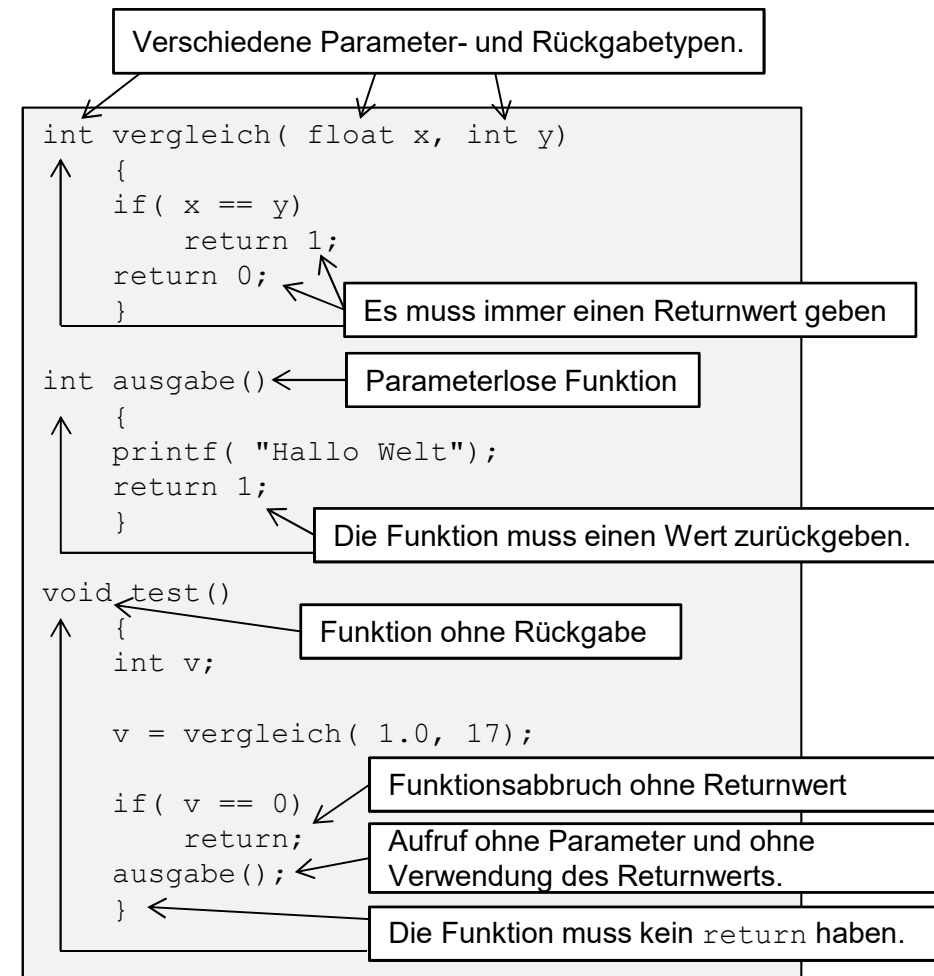
Eine Funktion kann unterschiedliche Parametertypen haben oder sogar parameterlos sein. Eine Funktion kann auch ohne Rückgabewert sein und erhält in diesem Fall den Rückgabotyp `void`:

Hat eine Funktion einen Returntyp, muss überall dort, wo der Kontrollfluss der Funktion endet, ein zum Typ passenderer Wert zurückgegeben werden.

Hat eine Funktion keinen Returntyp, kann trotzdem eine `return`-Anweisung (ohne Returnwert) verwendet werden, um den Kontrollfluss zu beenden und unmittelbar zum rufenden Programm zurückzukehren.

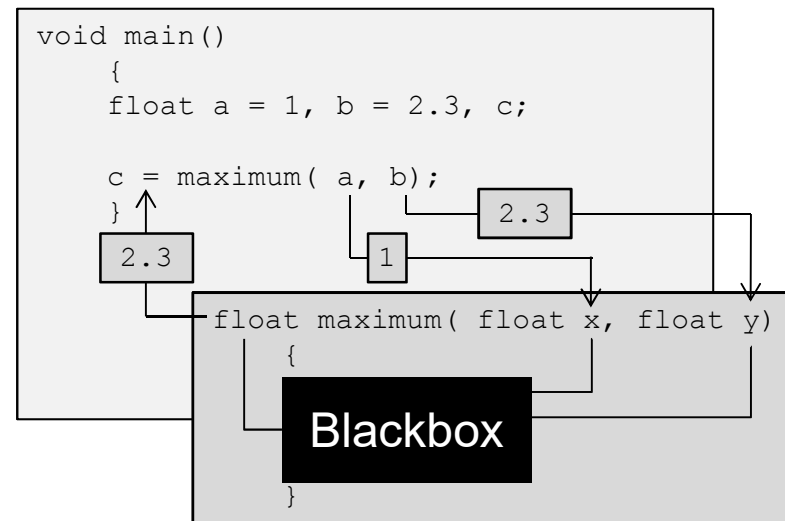
An der Schnittstelle vereinbarte Parameter müssen vom aufrufenden Programm mit dem korrekten Typ übergeben werden.

Der Rückgabewert muss vom aufrufenden Programm nicht entgegen genommen und auch nicht unbedingt verwendet werden.



Das Blackbox-Prinzip

Eine Funktion ist eine "doppelte Blackbox", in die von außen niemand hineinsehen, aus der aber auch von innen niemand heraussehen kann. Von innen und außen sieht man nur die Schnittstelle.

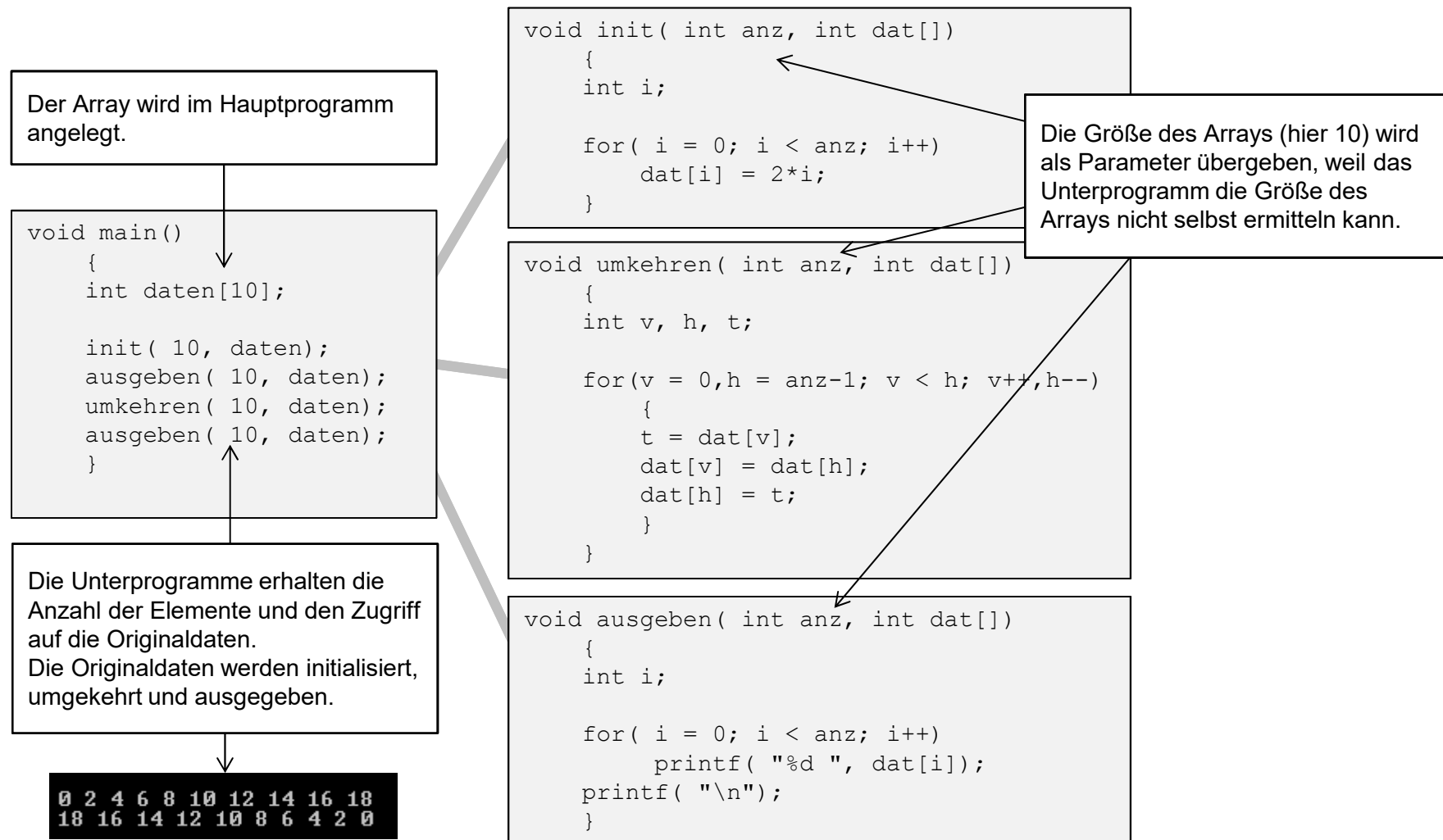


Das aufrufende Programm kennt keine internen Variablen der Funktion. Umgekehrt kennt auch die Funktion keine Variablen des Hauptprogramms. Eine zufällige Gleichheit von Variablennamen ändert daran nichts. Weder das Hauptprogramm kann auf Variablen des Unterprogramms noch das Unterprogramm kann auf Variablen des Hauptprogramms zugreifen.

An der Schnittstelle werden Kopien der Parameterwerte übergeben. Änderungen dieser Werte haben keine Auswirkungen auf das rufende Programm.

Arrays als Funktionsparameter

Wird ein Array an eine Funktion übergeben, so erhält die Funktion Zugriff auf die Originaldaten des Hauptprogramms. Es entsteht keine Kopie des gesamten Arrays sondern nur eine Kopie der Zugriffsinformation (Dazu mehr im Abschnitt über Zeiger und Adressen).



Strings als Funktionsparameter

Strings sind Arrays und werden daher an der Funktionsschnittstelle wie Arrays behandelt:

```
void main()
{
    int l, v;

    l = stringlaenge( "qwert");
    printf( "Laenge: %d\n", l);

    v = stringvergleich( "qwert", "qwerz");
    if( v == 1)
        printf( "gleich\n");
    else
        printf( "ungleich\n");
}
```

Laenge: 5
ungleich

```
int stringlaenge( char s[])
{
    int i;

    for( i = 0; s[i] != 0; i++)
        ;
    return i;
}
```

```
int stringvergleich( char s1[], char s2[])
{
    int i;

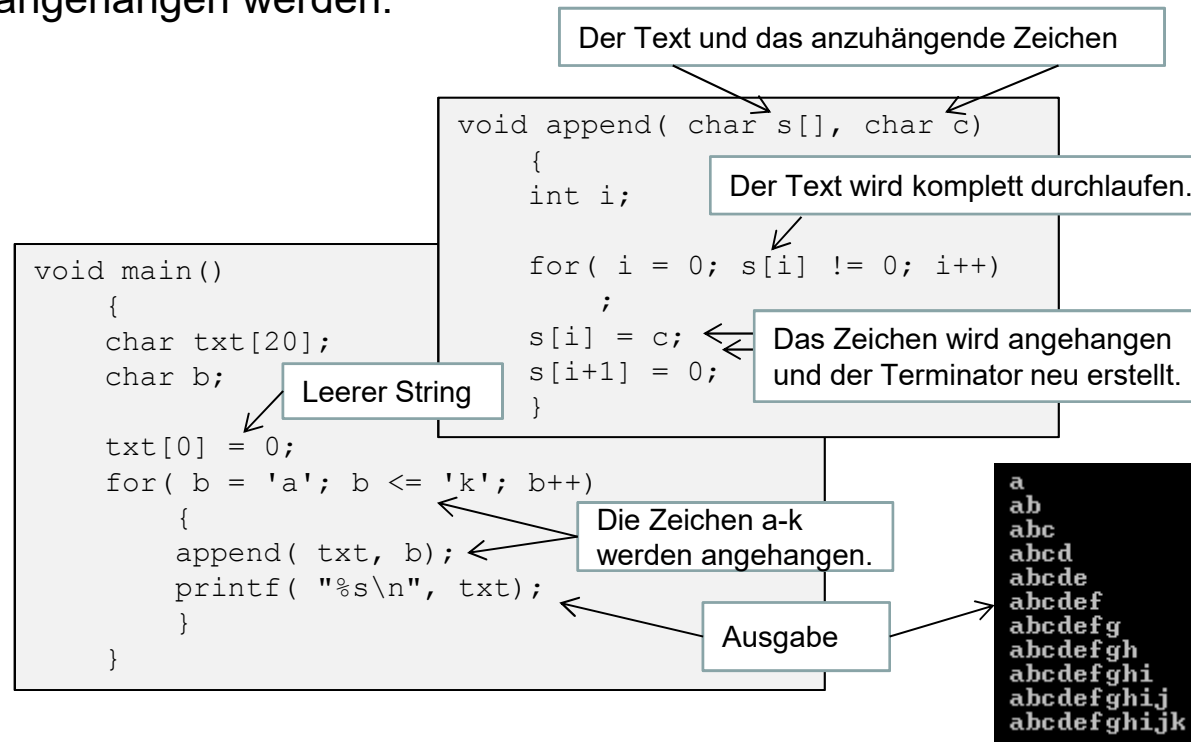
    for( i = 0; (s1[i]!=0)&&(s1[i]==s2[i]); i++)
        ;
    return s1[i] == s2[i];
}
```

Wegen des Terminatorzeichens kann das Unterprogramm das Ende des Strings ermitteln. Es muss daher keine Längeninformation mitgegeben werden. Verfügt das rufende Programm über die Länge des Strings, so kann diese mitgegeben werden, damit die Länge im Unterprogramm nicht erneut berechnet werden muß.

Wird der String im Unterprogramm verändert, sollte die Puffergröße als Parameter mitgegeben werden, um Pufferüberschreitungen (Buffer Overflow) vermeiden zu können.

Stringmanipulationen

Strings können (wie Arrays) in Funktionen verändert werden. Zum Beispiel können Zeichen an einen String angehängen werden:



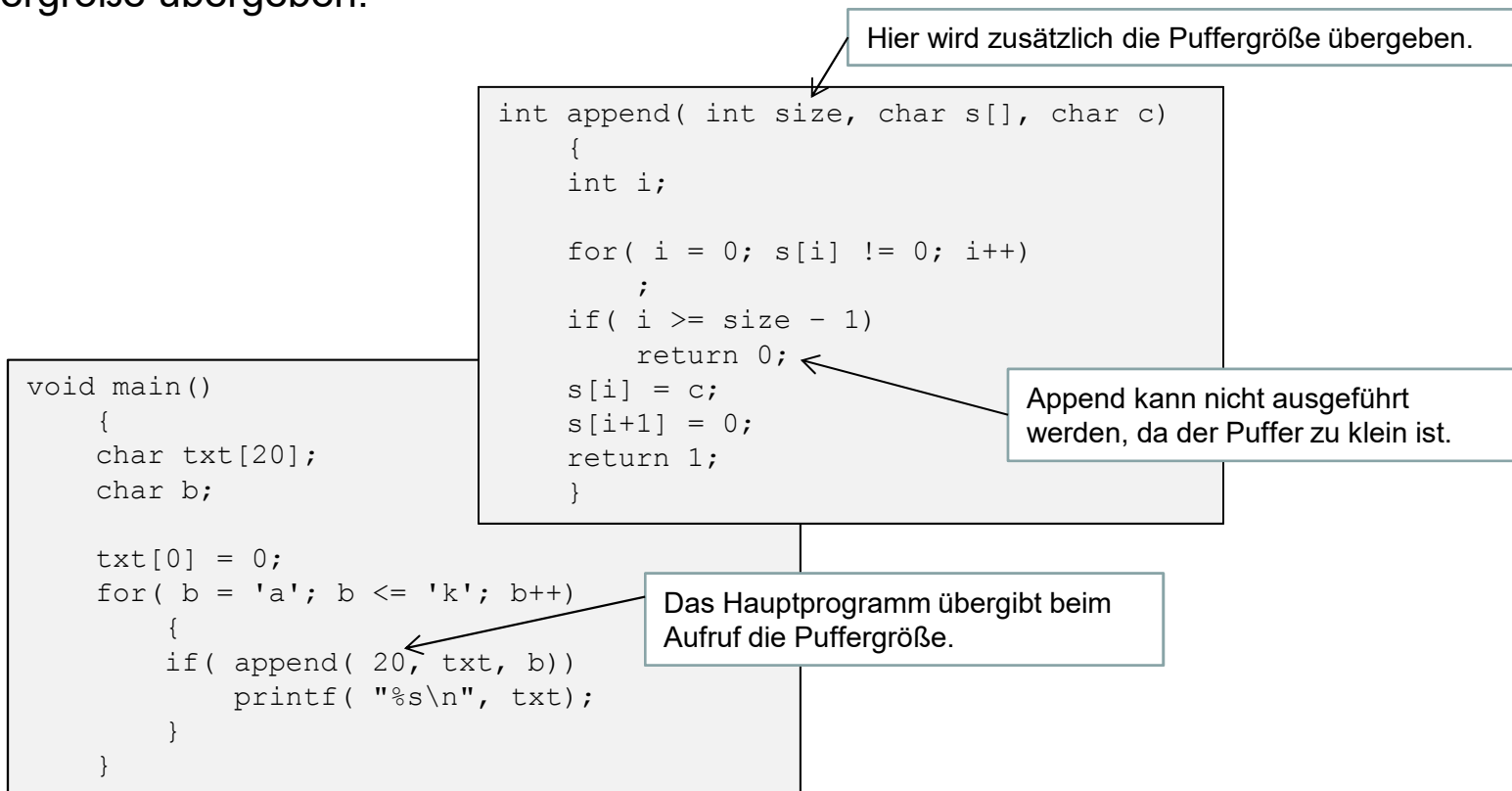
Vorsicht bei Veränderungen von Strings im Unterprogramm.

Der Programmierer ist dafür verantwortlich, dass keine Arraygrenzen überschritten werden und der String stets korrekt terminiert ist.

Wird im obigen Beispiel `append` mehr als 19 mal gerufen, stürzt das Programm ab. Fehlende Sicherungen gegen Pufferüberschreitungen sind eine Hauptfehlerquelle in C-Programmen.

Stringmanipulationen 2

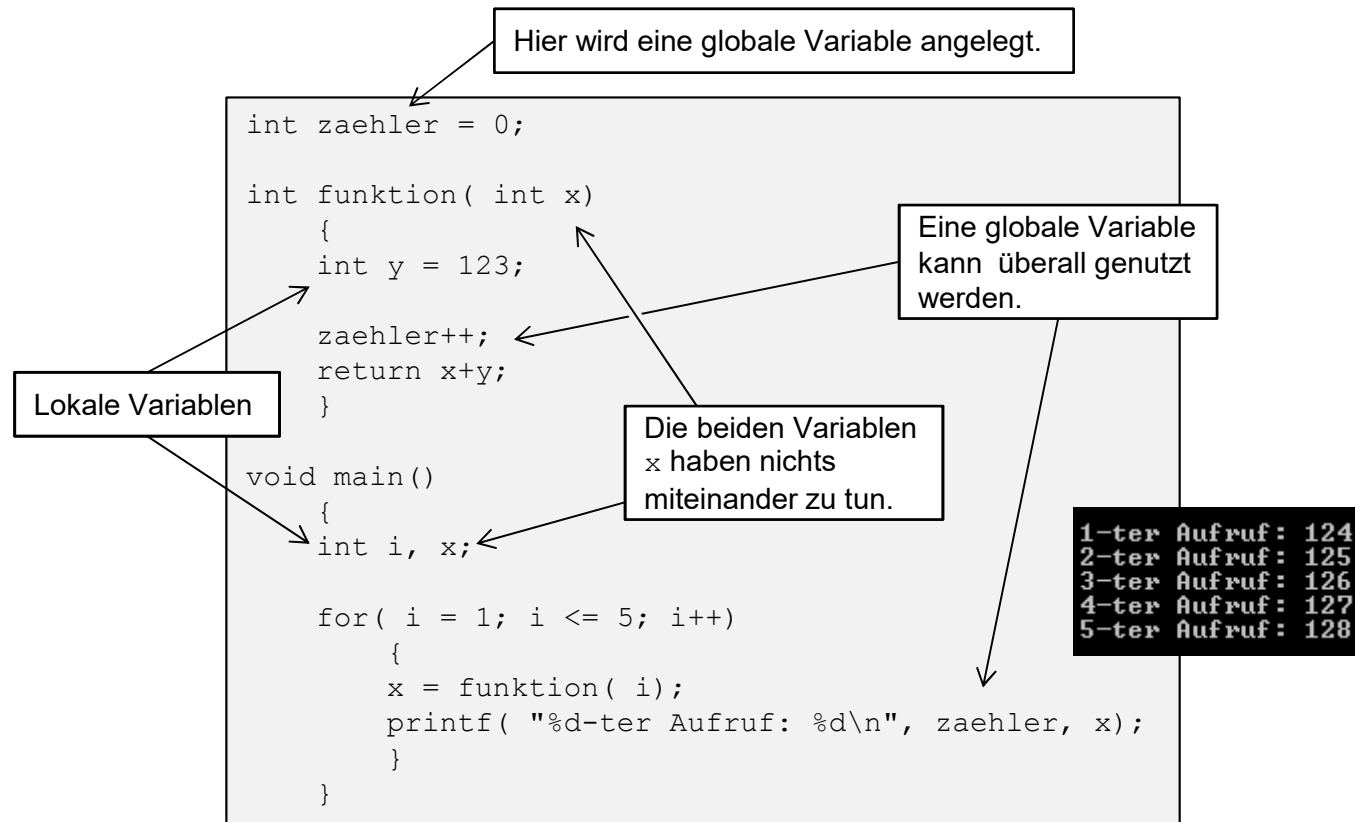
Um Pufferüberschreitungen abfangen zu können, sollte man Funktionen, die Strings verändern, die Puffergröße übergeben:



Nach wie vor kann `append` nur 19 mal mit Erfolg gerufen werden, aber das Programm stürzt nicht mehr ab und meldet zurück, wenn der String nicht mehr vergrößert werden kann.

Globale Variablen

Variablen außerhalb von Funktionen sind global und können funktionsübergreifend genutzt werden:

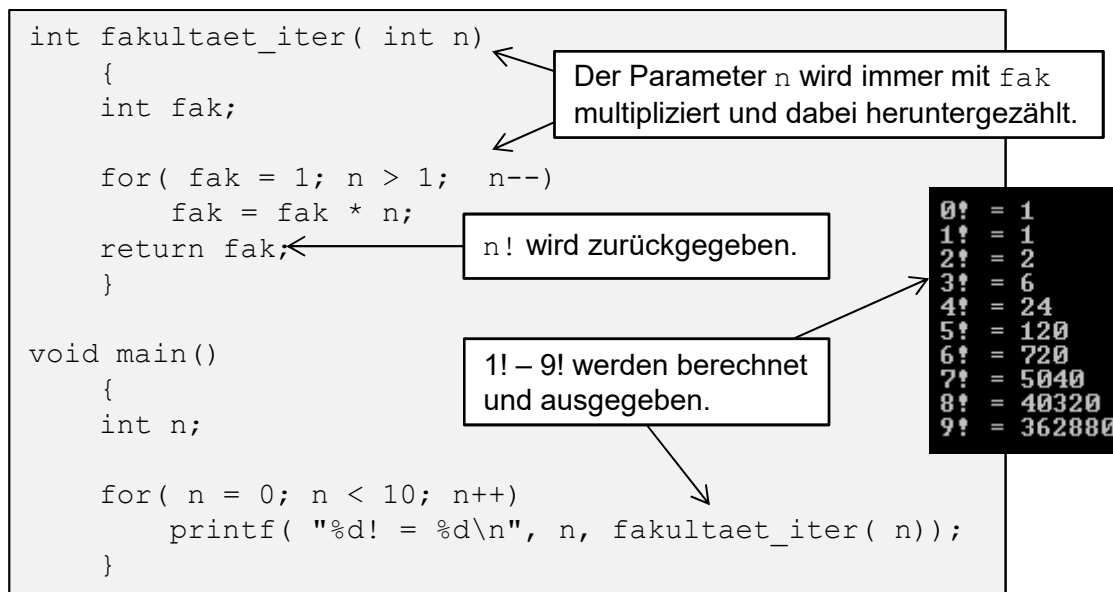


Globale Variablen sollten nur sparsam verwendet werden, da sie die Schnittstellen unterlaufen (Seiteneffekt). Verwenden Sie eine globale Variable nur für ein Datum, das programmweit überall mit der gleichen Bedeutung verwendet wird.

Beispiel - Die Fakultätsfunktion

Am Beispiel der Fakultätsfunktion wollen wir eines der wichtigsten Programmierprinzipien kennenlernen. Dazu implementieren wir diese Funktion zunächst in naheliegender Weise.

Die Fakultät $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ kann durch iterativ in einer Schleife berechnet werden:



Rekursion

Funktionen können sich selbst unmittelbar oder mittelbar (über eine andere Funktion) aufrufen. Man nennt dies **Rekursion**. Rekursion ist eine einfache und elegante Lösungsstrategie, wenn man ein Problem auf ein oder mehrere strukturell gleiche, aber "kleinere" Probleme zurückführen kann, bis man am Ende auf ein leicht zu lösendes Problem stößt.

Rekursive Definition der Fakultätsfunktion:

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{falls } n > 1 \end{cases}$$

In dieser Definition führt man die Berechnung von $n!$ auf die Berechnung der kleineren Fakultät $(n-1)!$ zurück, bis man am Ende auf ein leicht zu lösendes Problem (Berechnung von $1!$) stößt.

Die Definition führt unmittelbar zu einer **rekursiven Implementierung** der Fakultätsfunktion:

```
int fakultaet_rek( int n)
{
    if( n <= 1)
        return 1;
    return n*fakultaet_rek( n-1);
}
```

$n! = 1$ falls $n \leq 1$.

Selbstauf
(unmittelbare Rekursion)

$n! = n \cdot (n-1)! \text{ falls } n > 1$.

An der Schnittstelle ist diese Funktion identisch mit der nicht rekursiven Variante und kann daher exakt gleich verwendet werden:

```
void main()
{
    int n;

    for( n = 0; n < 10; n++)
        printf( "%d! = %d\n", n, fakultaet_rek( n));
}
```

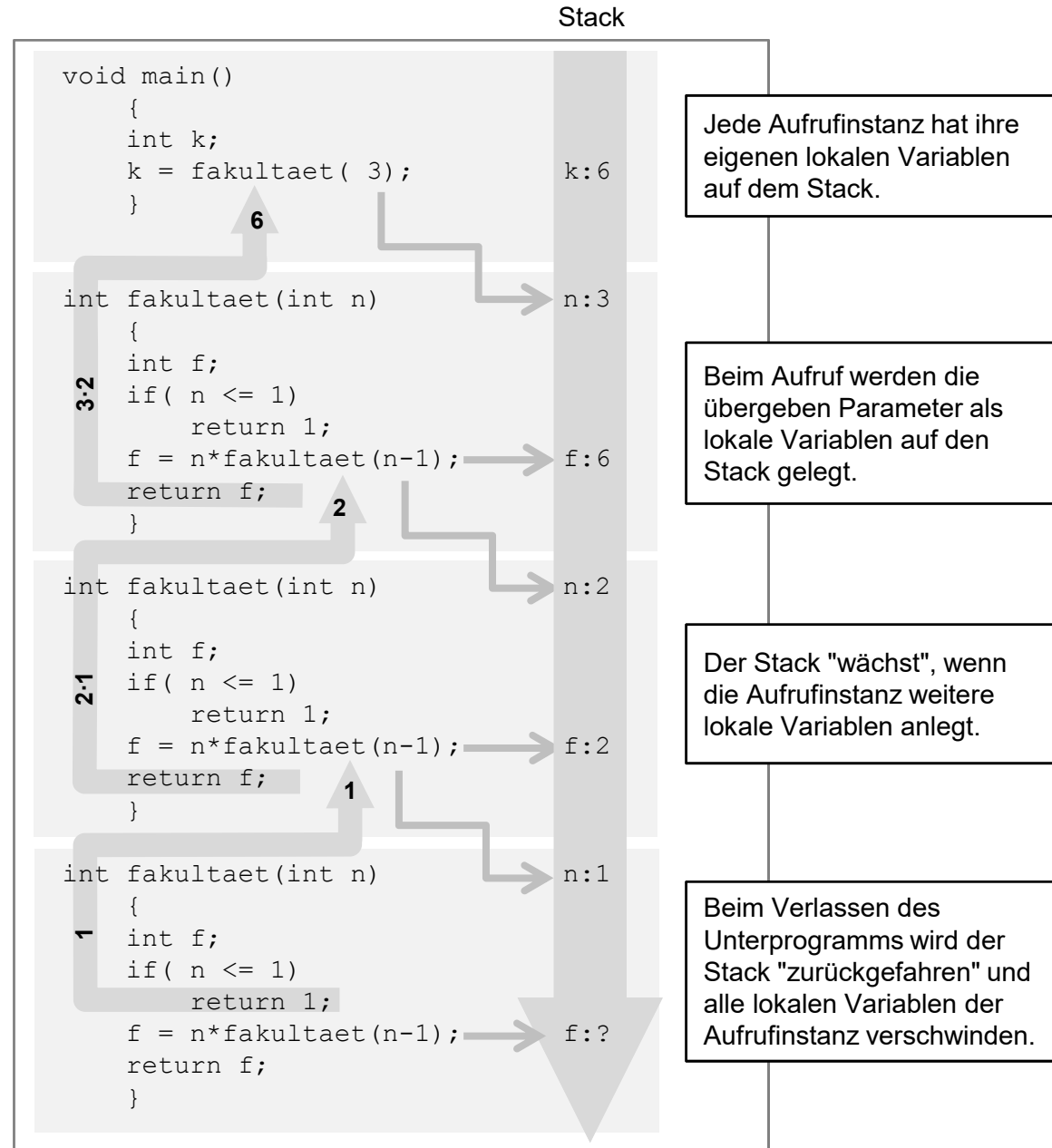
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Rekursion und der Stack

Der Stack ist eine Speicherstruktur in der das Laufzeitsystem temporäre Daten ablegt.

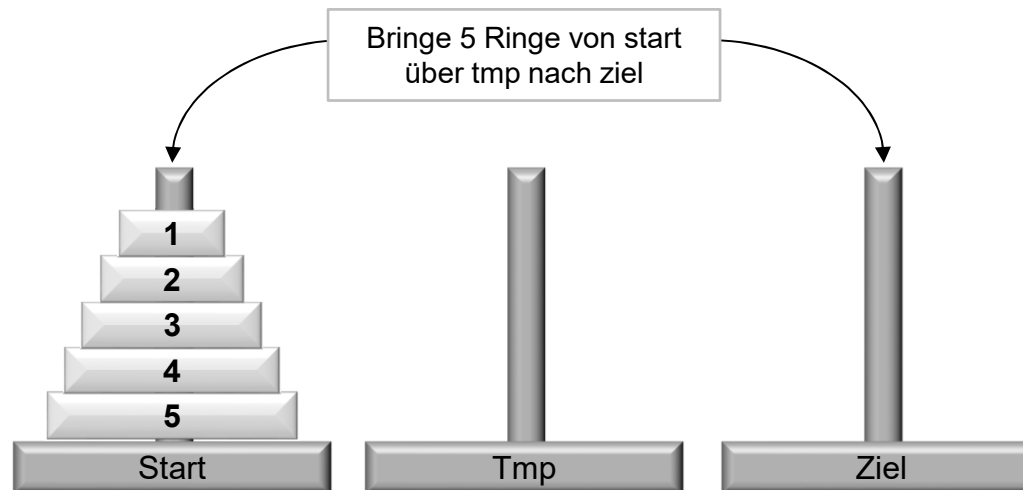
Jede Funktion, besser gesagt, jede Aufrufinstanz einer Funktion hat ihre eigene Laufzeitumgebung auf dem Stack und ist dadurch von den Aufrufinstanzen anderer Funktionen oder der eigenen Funktion abgeschottet.

Beim Rücksprung werden die lokalen Daten der Aufrufinstanz wieder beseitigt.



Die Türme von Hanoi

Aufgabe: Transportiere n Ringe unterschiedlicher Größe von einem Stab zu einem anderen, wobei ein Hilfsstab benutzt werden kann.

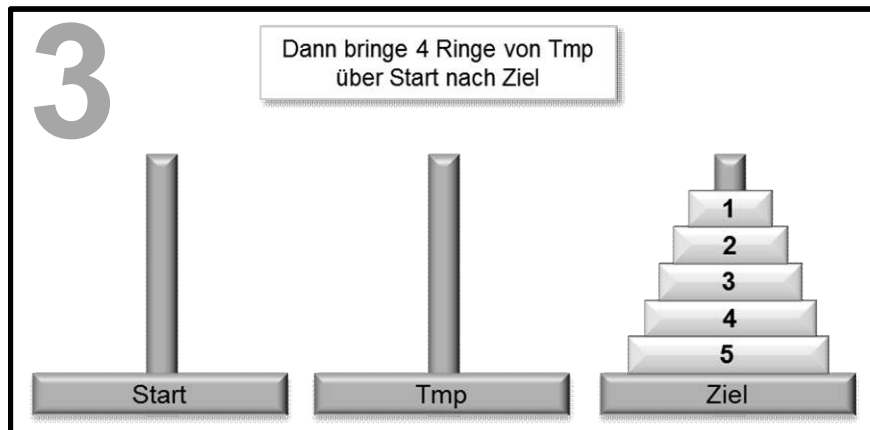
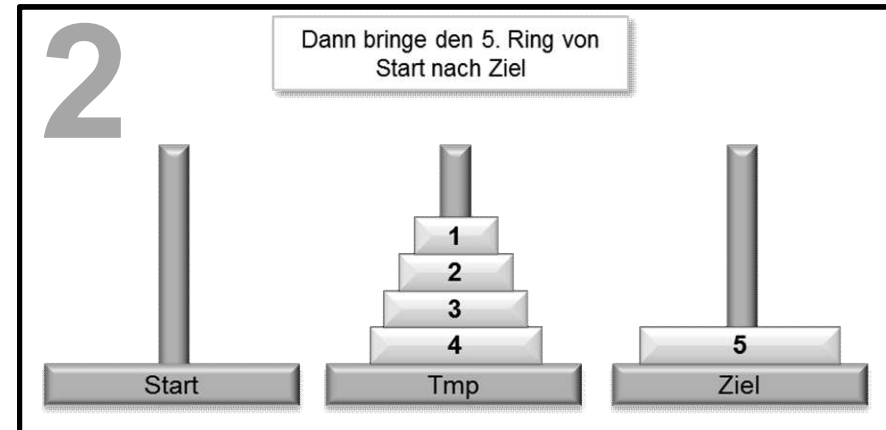
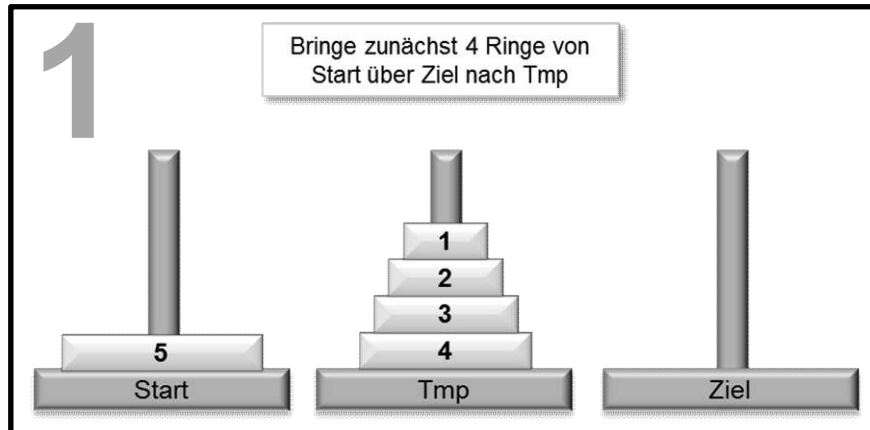


Regeln:

- Es darf ein Hilfsstab (Tmp) zur Zwischenablage benutzt werden.
- Es darf in einem Schritt immer nur ein Ring bewegt werden.
- Es darf nie ein kleinerer Ring unter einem größeren Ring liegen.

Rekursive Lösung der Türme von Hanoi

Wir zeigen, dass man 5 Ringe bewegen kann, wenn man annimmt, dass man 4 Ringe bewegen kann:



Wir wissen nicht, wie die Ringe zu bewegen sind, aber wir wissen, dass wir 5 Ringe bewegen können, wenn wir vier Ringe bewegen können.

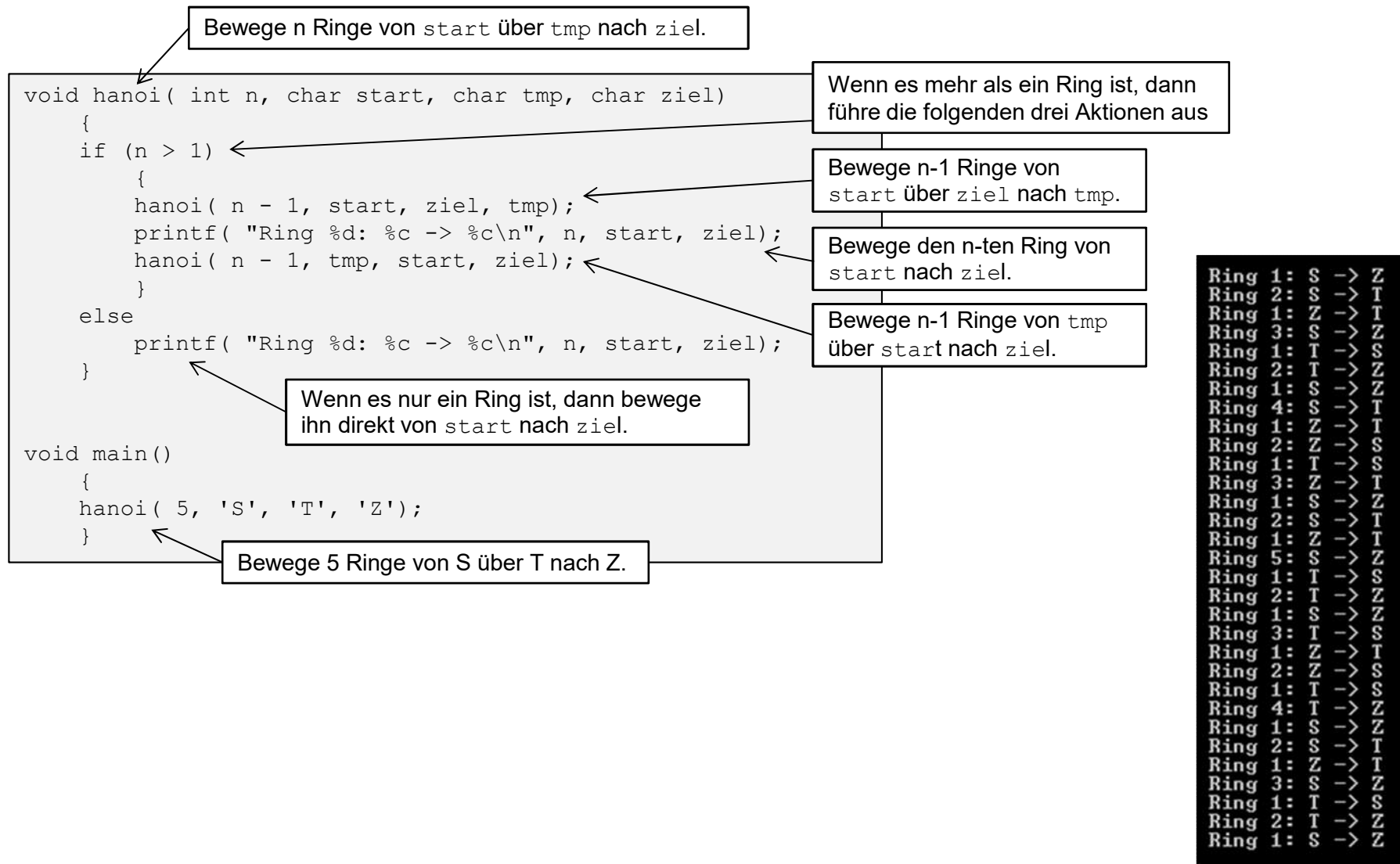
Allgemein: Wenn wir n Ringe bewegen können, dann können wir auch $n+1$ Ringe bewegen.

Man kann einen Ring bewegen.

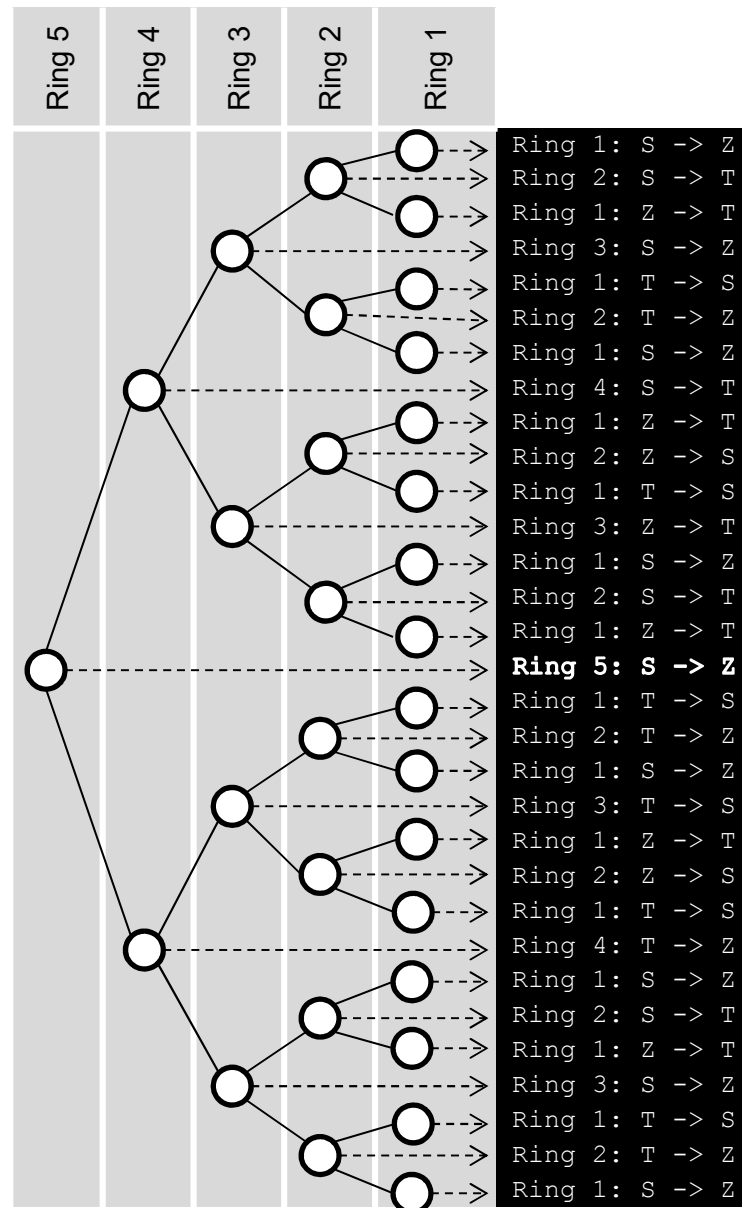
Wenn man n Ringe bewegen kann, dann kann man auch $n+1$ Ringe bewegen.

→ ...Vollständige Induktion... → Man kann beliebig viele Ringe bewegen

Die Türme von Hanoi als rekursive Funktion

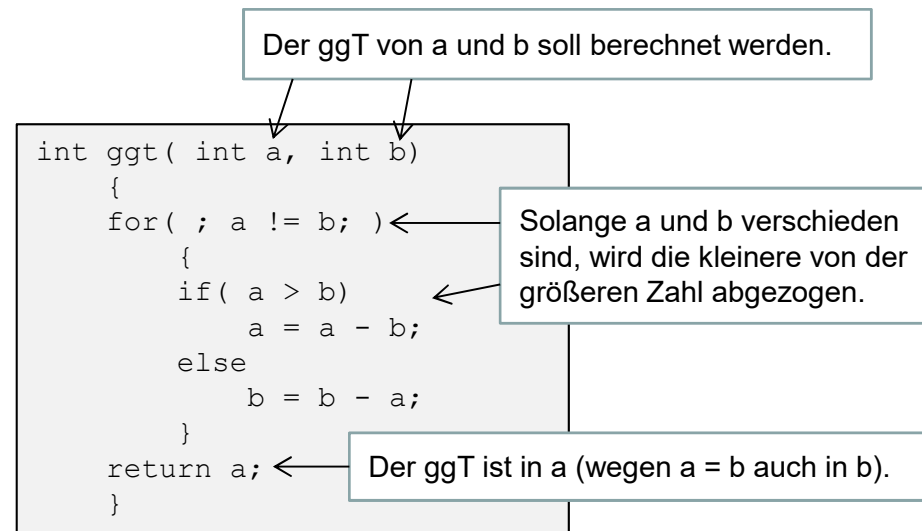


Rekursive Lösungssuche in der Funktion hanoi:



Beispiel - Berechnung des größten gemeinsamen Teilers

Den größten gemeinsamen Teiler (ggT) von zwei Zahlen erhält man, indem man solange die kleinere Zahl von der größeren abzieht bis beide Zahlen gleich sind.



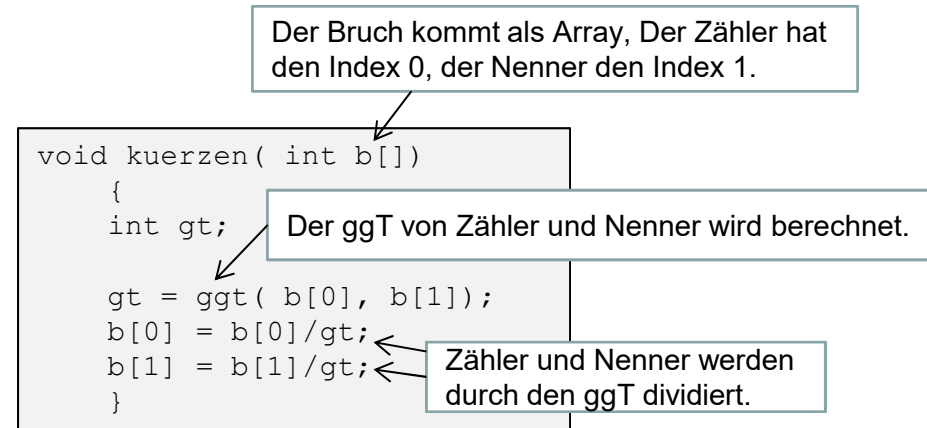
Mit dieser Hilfsfunktion wollen wir ein Programm zur Bruchrechnung erstellen.

Bruchrechnung – Teil 1

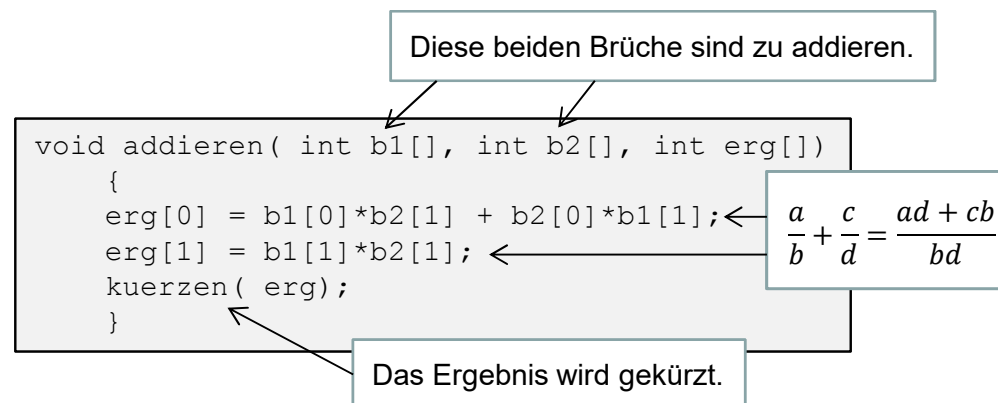
Einen Bruch speichern wir mit Zähler und Nenner in einem Array mit zwei Elementen:

```
int bruch[2]; // bruch[0] ist der Zähler, bruch[1] ist der Nenner
```

Brüche werden gekürzt, indem Zähler und Nenner durch ihren ggT geteilt werden:

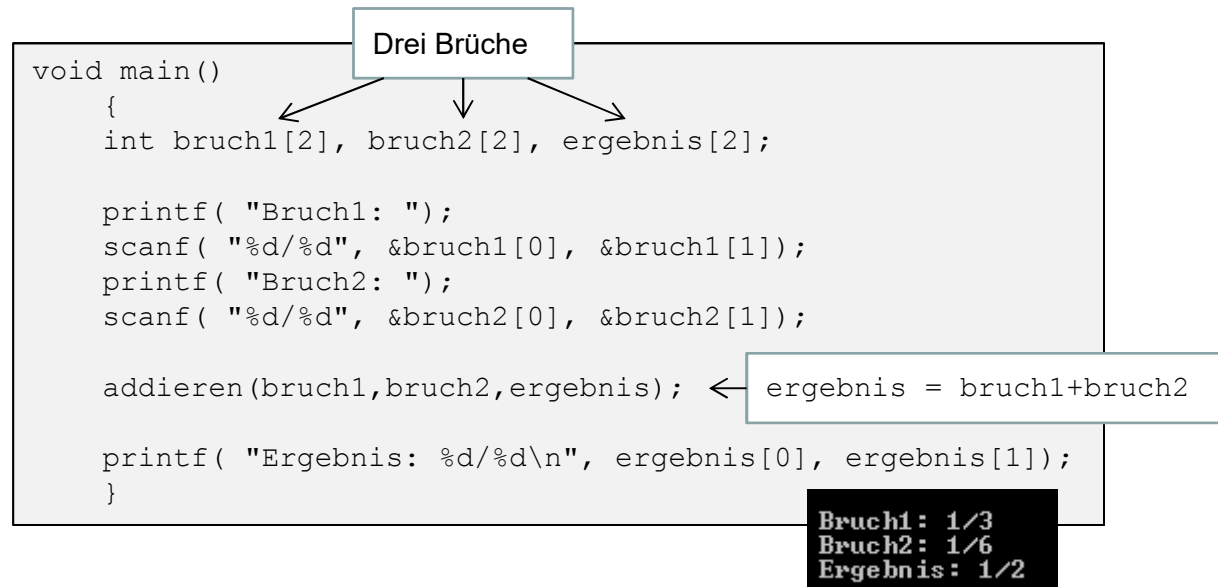


Brüche werden nach der Formel $\frac{a}{b} + \frac{c}{d} = \frac{ad+cb}{bd}$ addiert, wobei anschließend gekürzt werden sollte:



Bruchrechnung – Teil 2

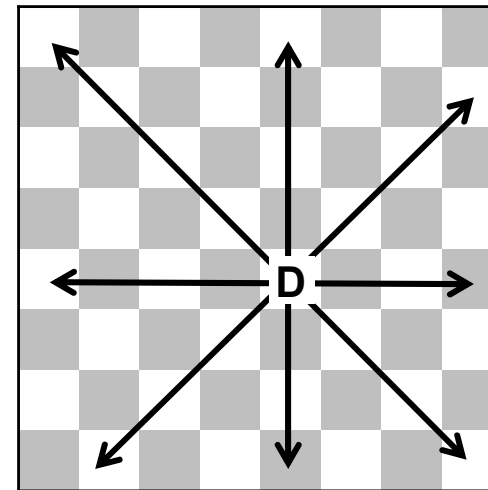
Mit den Funktionen kürzen und addieren kann ein kleines Programm zur Bruchrechnung geschrieben werden:



Damenproblem - Teil 1

Positioniere n Damen auf einem $n \times n$ Schachbrett so, dass keine Dame eine andere schlagen kann. Dies bedeutet:

- höchstens (genau) eine Dame in jeder Zeile
- höchstens (genau) eine Dame in jeder Spalte
- höchstens eine Dame in jeder Diagonalen

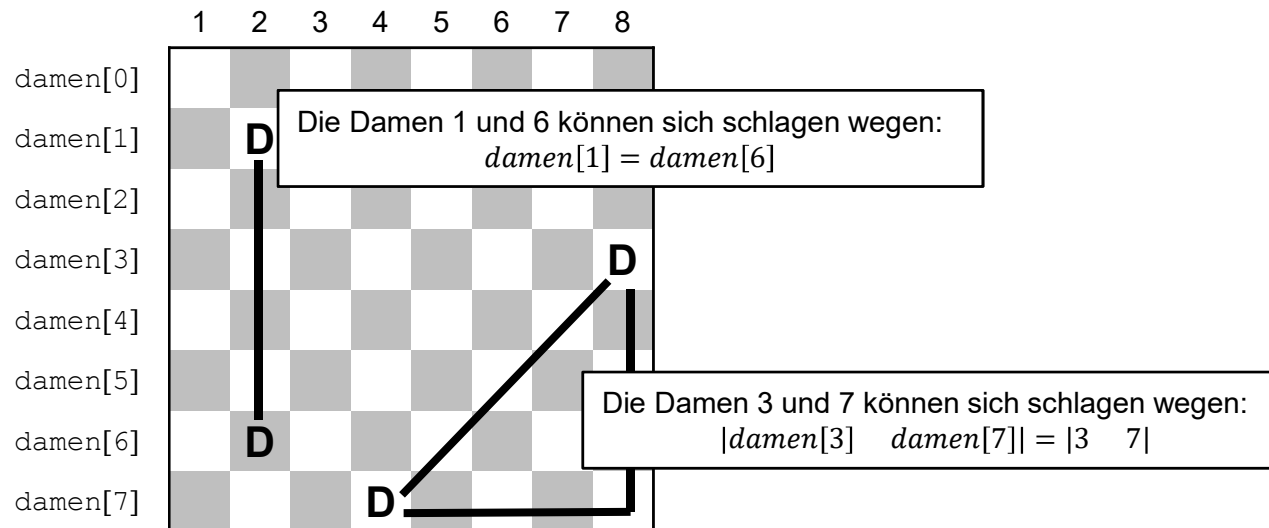


Modellierung durch einen Array `damen[8]`, wobei der Wert im Array die Spalte angibt, in der die Dame steht:

	1	2	3	4	5	6	7	8
<code>damen[0]</code>	D							
<code>damen[1]</code>					D			
<code>damen[2]</code>							D	
<code>damen[3]</code>						D		
<code>damen[4]</code>			D					
<code>damen[5]</code>							D	
<code>damen[6]</code>		D						
<code>damen[7]</code>				D				

Damenproblem – Teil 2

Prüfung, ob eine Dame von einer anderen geschlagen werden kann:



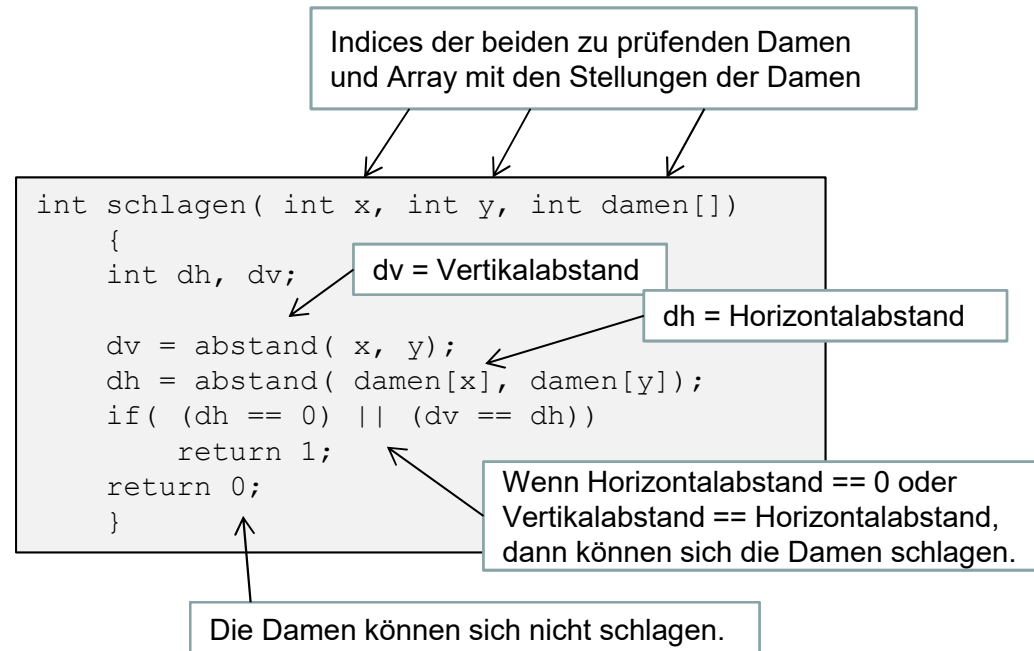
Eine Dame kann eine andere schlagen, wenn beide Damen in der gleichen Spalte stehen oder wenn der "Horizontalabstand" gleich dem "Vertikalabstand" ist.

Wir benötigen eine Abstandsfunktion:

```
int abstand( int x, int y)
{
    if( x >= y)
        return x - y;
    return y - x;
}
```

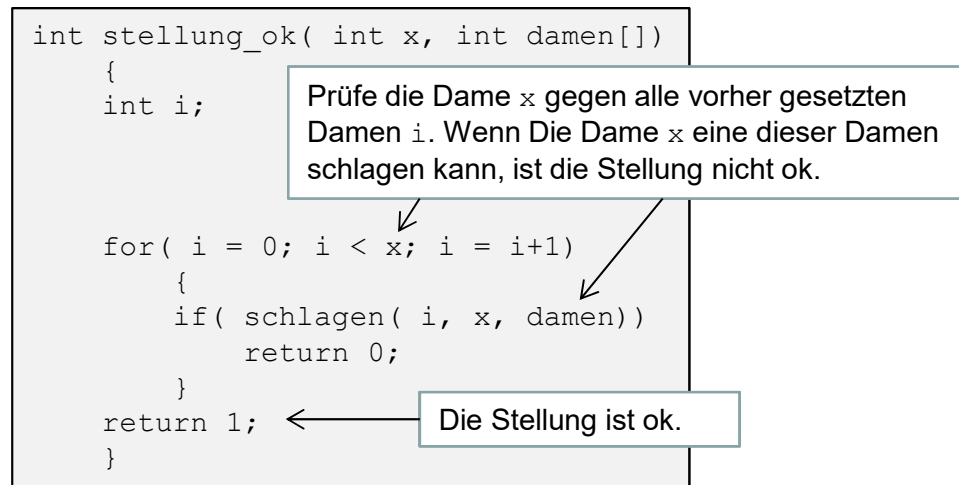

Damenproblem – Teil 3

Prüfung, ob eine Dame von einer anderen geschlagen werden kann:



Damenproblem – Teil 4

Prüfe, ob eine Dame von einer zuvor gesetzten (oberhalb stehenden) geschlagen werden kann:



Damenproblem – Teil 5

Lösungsausgabe

```
int laufendenummer = 0;

void print_loesung( int anz, int damen[])
{
    int i;

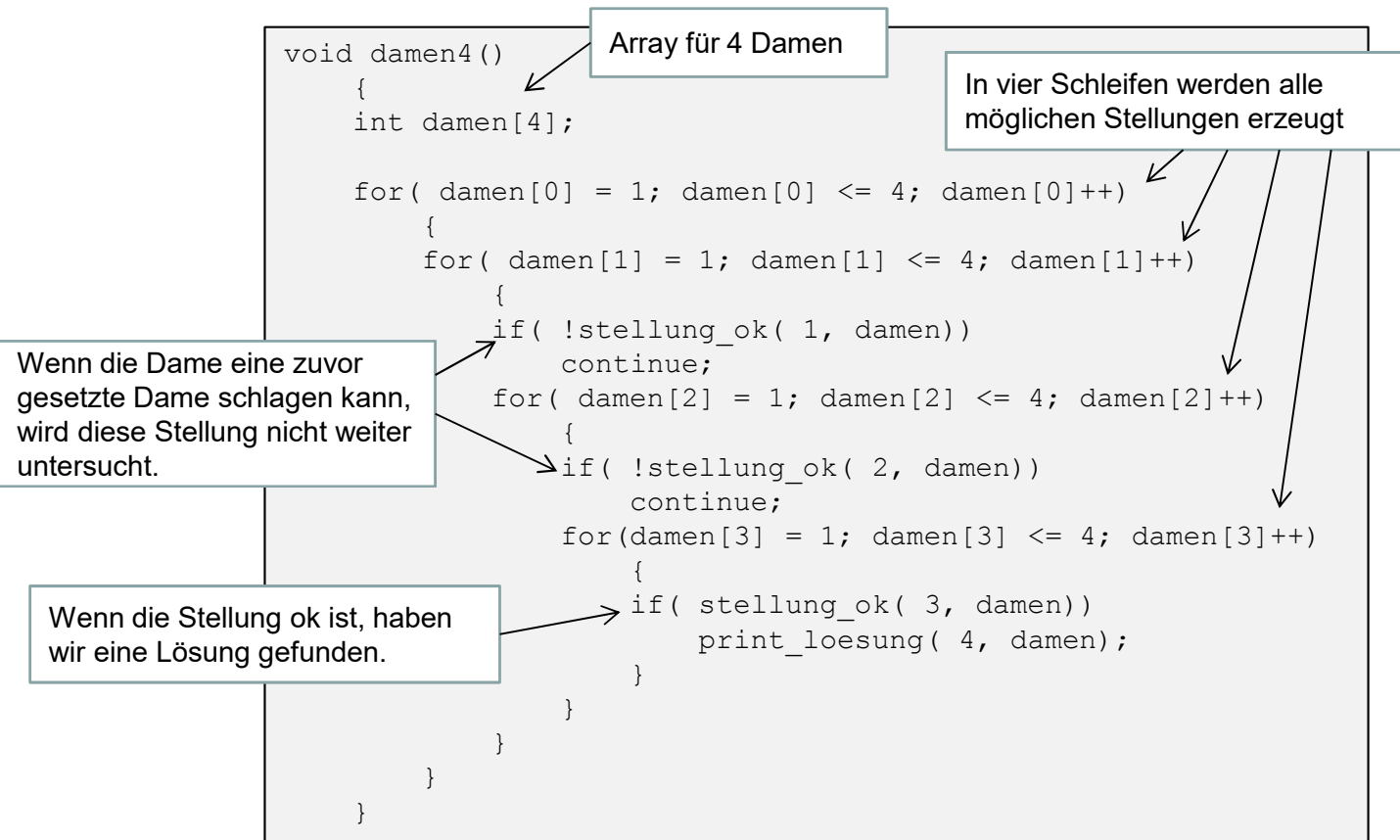
    laufendenummer++;
    printf( "%2d. Loesung: ", laufendenummer);
    for( i = 0; i < anz; i = i + 1)
        printf( " %d", damen[i]);
    printf( "\n");
}
```

← In einer globalen Variablen
zählen wir die Lösungen.

← Lösungsausgabe.

Damenproblem – Teil 6

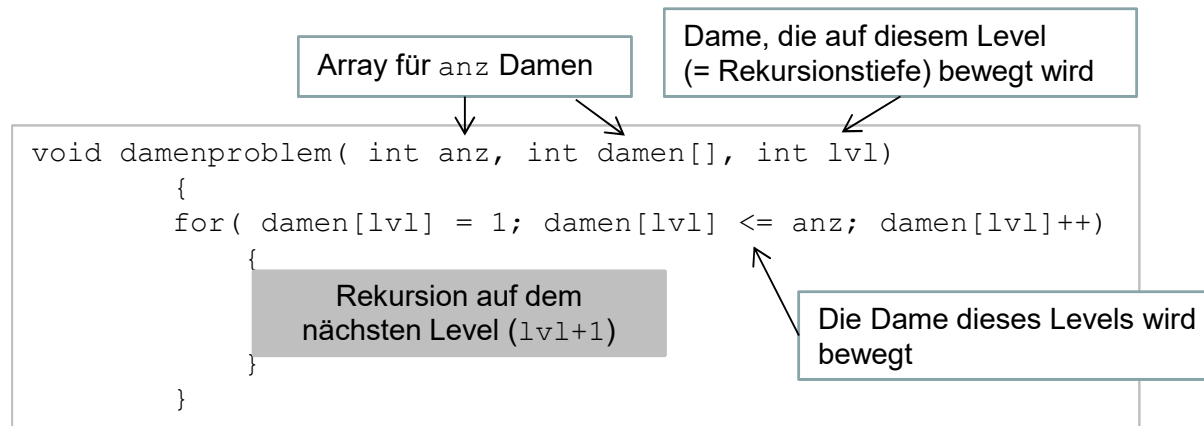
Lösung des 4-Damenproblems durch vier ineinander geschachtelte Schleifen:



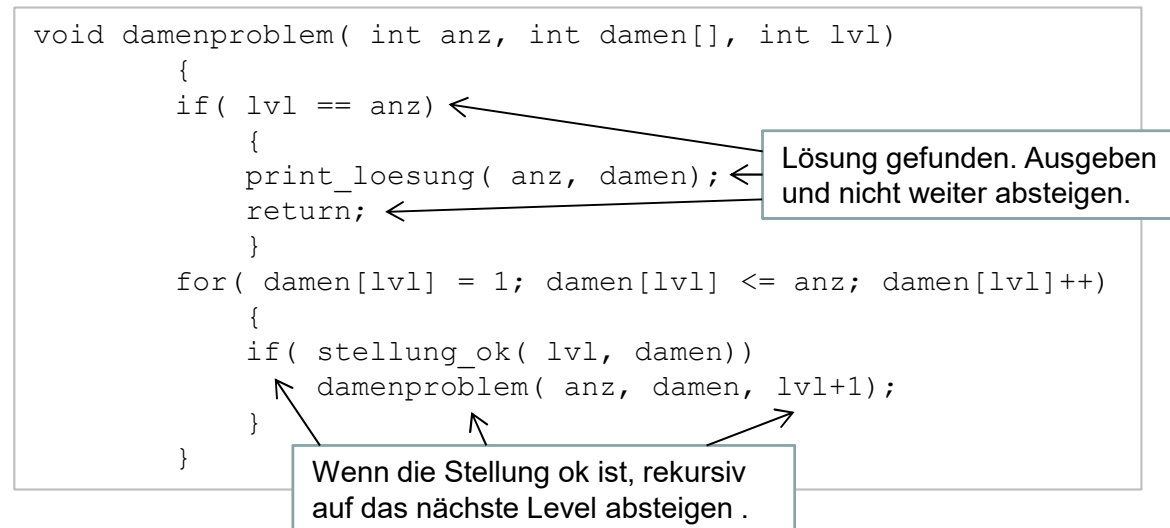
Damenproblem – Teil 7

Rekursive Lösung des allgemeinen Damenproblems

Ansatz:



Vollständige Lösung



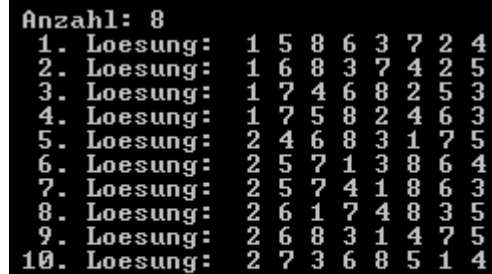
Damenproblem – Teil 8

Hauptprogramm und Ausgabe:

```
void main()
{
    int damen[20];
    int anz;

    printf( "Anzahl: ");
    scanf( "%d", &anz);

    damenproblem( anz,  damen, 0);
}
```



Anzahl: 8

1. Loesung:	1	5	8	6	3	7	2	4
2. Loesung:	1	6	8	3	7	4	2	5
3. Loesung:	1	7	4	6	8	2	5	3
4. Loesung:	1	7	5	8	2	4	6	3
5. Loesung:	2	4	6	8	3	1	7	5
6. Loesung:	2	5	7	1	3	8	6	4
7. Loesung:	2	5	7	4	1	8	6	3
8. Loesung:	2	6	1	7	4	8	3	5
9. Loesung:	2	6	8	3	1	4	7	5
10. Loesung:	2	7	3	6	8	5	1	4

Anzahl der Lösungen und Anzahl der zu untersuchenden Stellungen für 1-15 Damen:

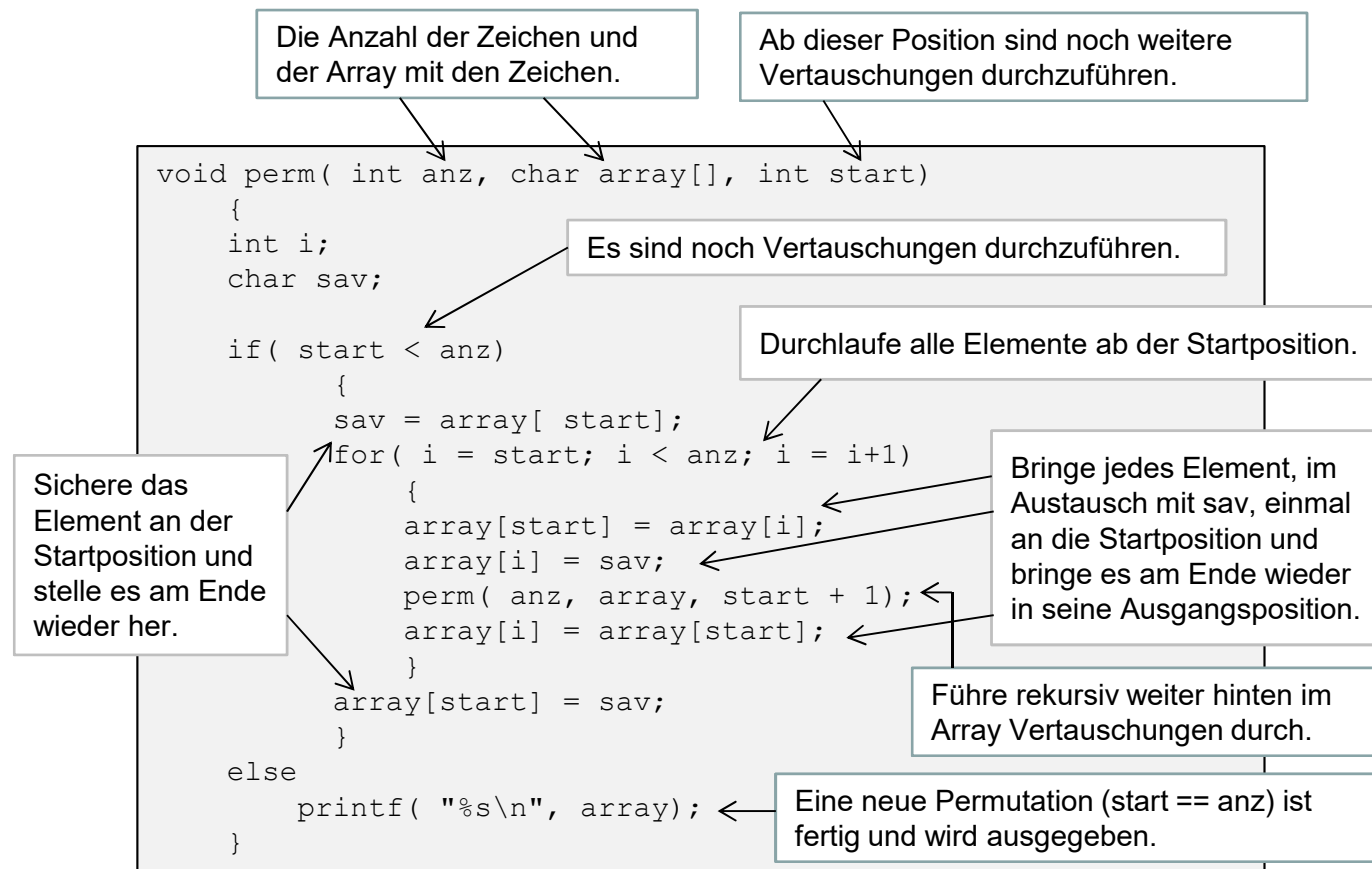
Damen	Loesungen	Stellungen
1	1	1
2	0	6
3	0	18
4	2	60
5	10	220
6	4	894
7	40	3584
8	92	15720
9	352	72378
10	724	348150
11	2680	1806706
12	14200	10103868
13	73712	59815314
14	365596	377901398
15	2279184	2532748320

Auch beim Damenproblem sind wir mit der kombinatorischen Explosion konfrontiert.

Permutationen – Teil 1

Aufgabe: Erzeuge alle möglichen Reihenfolgen (Permutationen) der Zeichen in einem String

Lösungsidee: Bringe jedes Zeichen durch Vertauschung einmal an die erste Position und erzeuge dann rekursiv alle Permutationen im Rest des Strings.



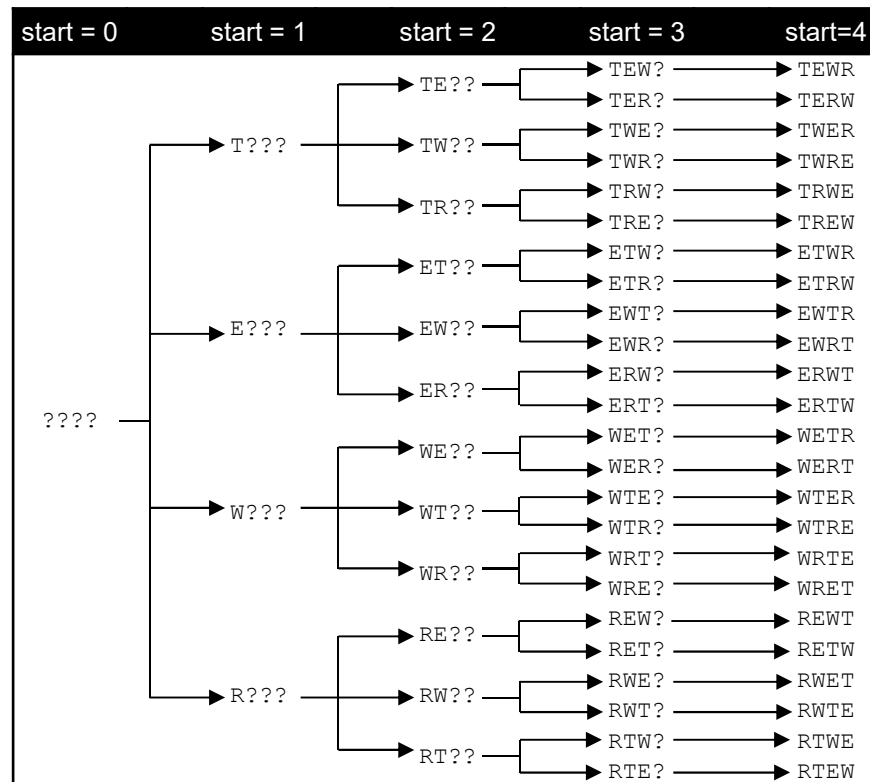
Permutationen – Teil 2

Hauptprogramm:

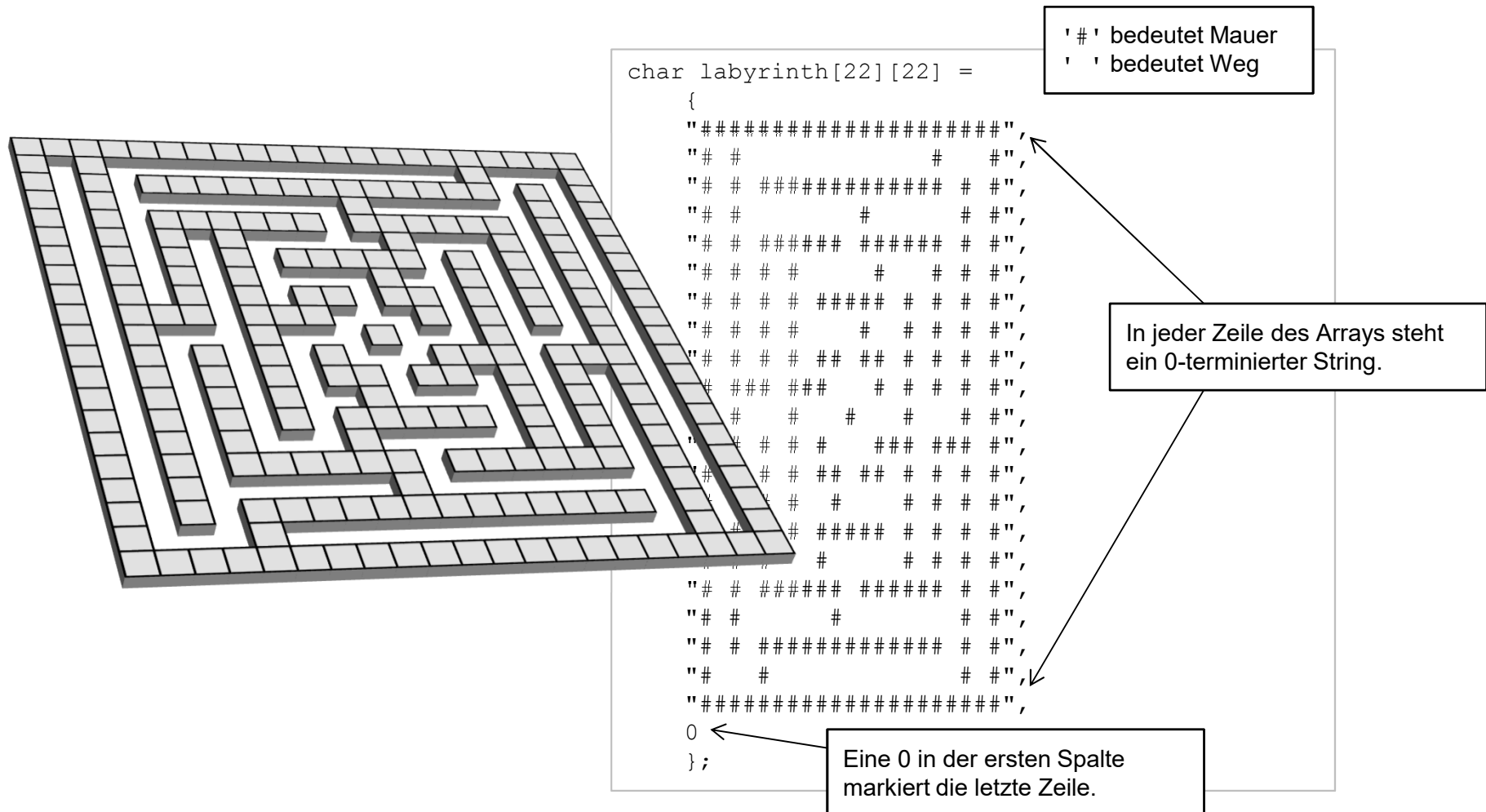
```
void main()
{
    char haufen[5] = "TEWR";

    printf( "Vorher:  %s\n", haufen);
    perm( 4, haufen, 0);
    printf( "Nachher: %s\n", haufen);
}
```

Vorgehensweise und Ausgabe:



```
Vorher:  TEWR
TEWR
TERW
TWER
TWRE
TRWE
TREW
ETWR
ETRW
ETRW
EWTR
EWRT
ERWT
ERTW
WETR
WERT
WTER
WTRE
WRTE
WRET
REWT
RETW
RWET
RWTE
RTEW
Nachher: TEWR
```

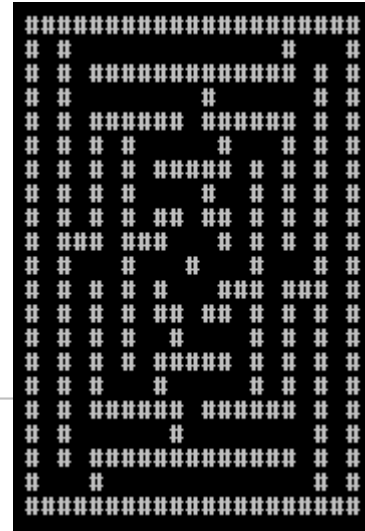



Labyrinth – Hilfsfunktion zur Ausgabe des Labyrinths

Die Ausgabe wird beendet, wenn in der ersten Spalte der betrachteten Zeile eine 0 steht.

```
void ausgabe()  
{  
    int zeile;  
  
    for( zeile = 0; labyrinth[zeile][0] != 0; zeile++)  
        printf( "%s\n", labyrinth[zeile]);  
}
```

Ausgabe einer Zeile des Labyrinths.



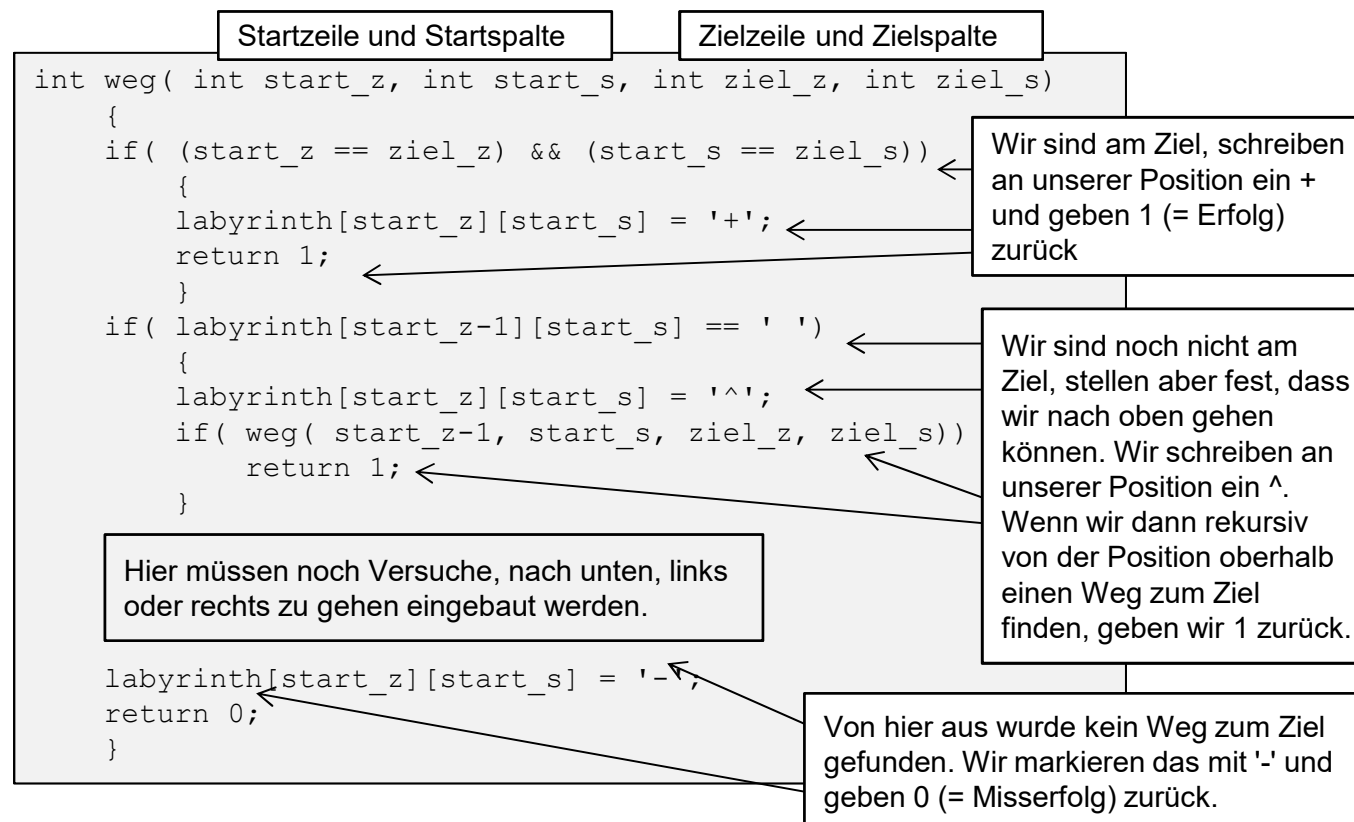
Labyrinth – Lösungsstrategie zur rekursiven Wegsuche

Betreten werden immer nur Felder auf denen ein Leerzeichen steht.

Es wird versucht zunächst nach oben zu gehen und von dort aus rekursiv einen Weg zum Ziel zu finden. Klappt das nicht, geht man zurück und versucht es unten. Klappt das auch nicht, versucht man es noch links und rechts. Sind alle Möglichkeiten erschöpft, gibt es von diesem Punkt aus keinen Weg zum Ziel.

Sobald das Ziel erstmalig erreicht ist, wird die Suche abgebrochen. Damit ist ein Weg, aber nicht unbedingt der kürzeste Weg, gefunden.

Das Programm markiert den eingeschlagenen Weg mit den Zeichen ^, v, < und >. Punkte, die als nicht zielführend erkannt wurden, werden mit – markiert, damit sie nicht erneut untersucht werden, wenn sie noch einmal erreicht werden.



Labyrinth – Das komplette Programm

```
int weg( int start_z, int start_s, int ziel_z, int ziel_s)
{
    if( (start_z == ziel_z) && (start_s == ziel_s)) ←
    {
        labyrinth[start_z][start_s] = '+';
        return 1;
    }
    if( labyrinth[start_z-1][start_s] == ' ')
    {
        labyrinth[start_z][start_s] = '^';
        if( weg( start_z-1, start_s, ziel_z, ziel_s))
            return 1;
    }
    if( labyrinth[start_z+1][start_s] == ' ')
    {
        labyrinth[start_z][start_s] = 'v';
        if( weg( start_z+1, start_s, ziel_z, ziel_s))
            return 1;
    }
    if( labyrinth[start_z][start_s-1] == ' ')
    {
        labyrinth[start_z][start_s] = '<';
        if( weg( start_z, start_s-1, ziel_z, ziel_s))
            return 1;
    }
    if( labyrinth[start_z][start_s+1] == ' ')
    {
        labyrinth[start_z][start_s] = '>';
        if( weg( start_z, start_s+1, ziel_z, ziel_s))
            return 1;
    }
    labyrinth[start_z][start_s] = '-';
    return 0;
}
```

Zusätzliche Versuche, nach unten, links oder rechts zu gehen.

Labyrinth – Hauptprogramm und Ausgabe

```

void main()
{
    int start_z, start_s, ziel_z, ziel_s;

    ausgabe();

    printf( "\nStart (Zeile Spalte): ");
    scanf( "%d %d", &start_z, &start_s);
    printf( "Ziel (Zeile Spalte): ");
    scanf( "%d %d", &ziel_z, &ziel_s);

    if( weg( start_z, start_s, ziel_z, ziel_s))
        ausgabe();
    else
        printf( "Kein Weg gefunden!\n");
}

```

Eingabe von Start und Ziel.

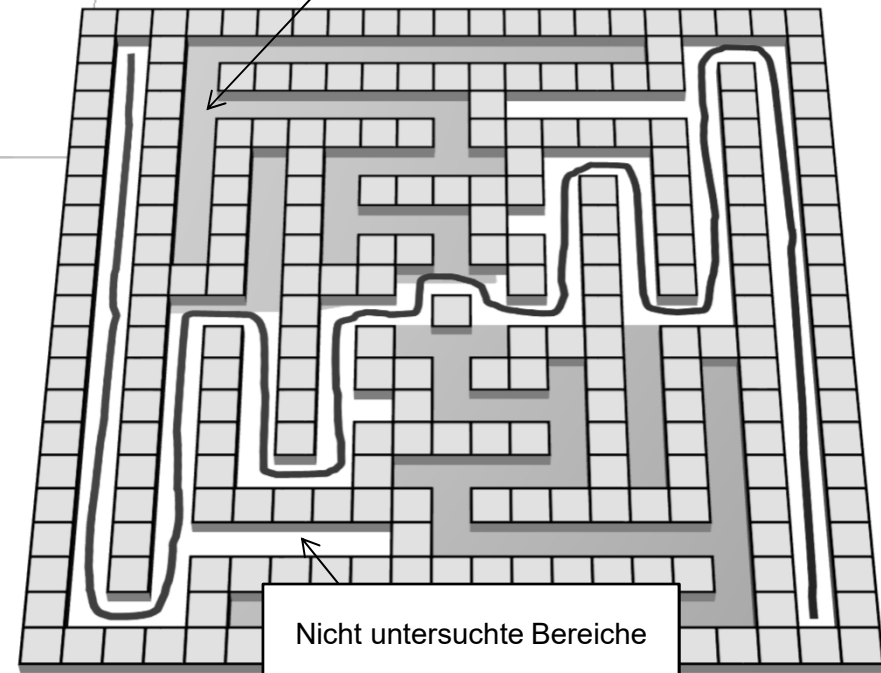
Wegsuche.

Nicht zum Ziel führende
Bereiche

```

Start <Zeile Spalte>: 1 1
Ziel <Zeile Spalte>: 19 19
#####
#v#-----#>>v#
#v#-#####^#v#
#v#-----# ^#v#
#v#-#####^#v#
#v#-#-#-----#>>v#^#v#
#v#-#-#-#####^#v#^#v#
#v#-#-#-----# ^#v#^#v#
#v#-#-#-#-#-#-#^#v#^#v#
#v#-#-#-#-#-#-#^#v#^#v#
#v#>>v#>>^#>>^#>>^#v#
#v#^#v#^#-----#-#-#-#v#
#v#^#v#^#-#-#-#-#-#v#
#v#^#v#^#-#-#-#-#-#v#
#v#^#v#^#-#-#-#-#-#v#
#v#^#v#>>^#-#-#-#-#v#
#v#^#v#-#-#-#-#-#v#
#v#^#-#-#-#-#-#v#
#>>^#-----#+#
#####

```



Nicht untersuchte Bereiche