

Stacks und Queues für temporäre Daten

Datenstruktur können die **Performance** verbessern.

Und: sie können **einfacheren Code** erlauben:

Stacks und Queues sind:

- Arrays mit Einschränkungen (die sie einfacher machen)
- Container zur **Verwaltung transienter, temporärer Daten** wie Druckaufträge oder Zwischenergebnisse (Informationen, die nach ihrer Verarbeitung bedeutungslos sind)



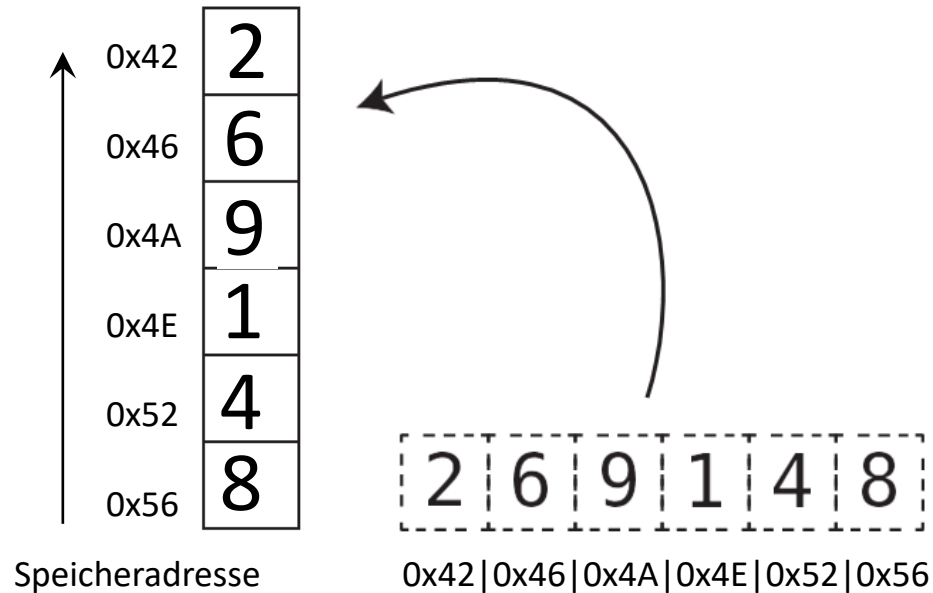
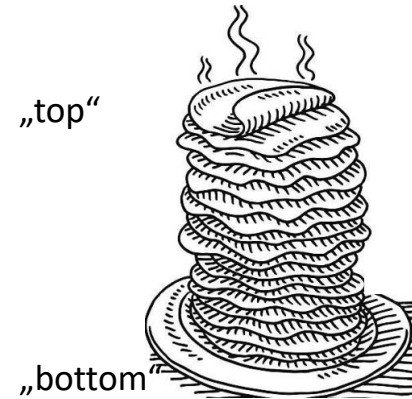
Bsp.: Essenbestellung im Restaurant

Mit der Lieferung ist die Bestellung hinfällig und überflüssig

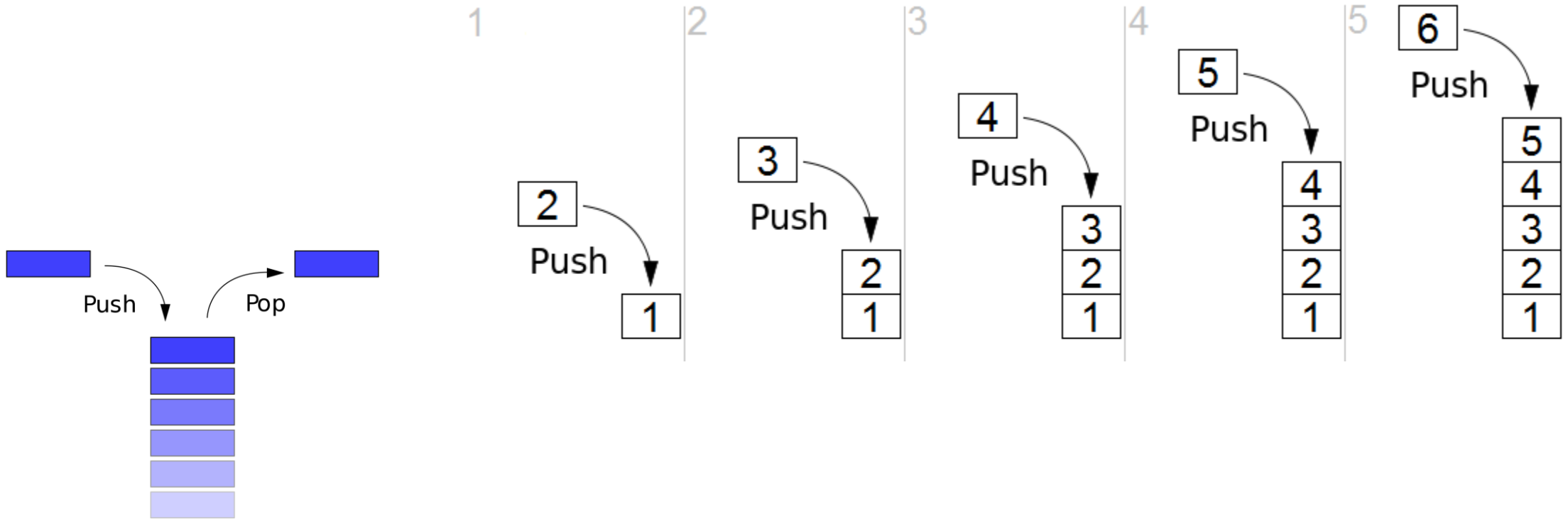
Stacks und Queues verwalten diese Daten – mit unterschiedlicher Unterstützung der Reihenfolge

Stacks („Stapelspeicher“) funktionieren wie Arrays – mit folgenden 3 Restriktionen:

- Daten können nur am Ende („top“) des Stack hinzugefügt werden
- Daten können nur am Ende („top“) des Stack entnommen/ gelöscht werden
- Nur das letzte (oberste) Element des Stack kann gelesen werden



Stack in Aktion: LIFO (Last In, First Out)

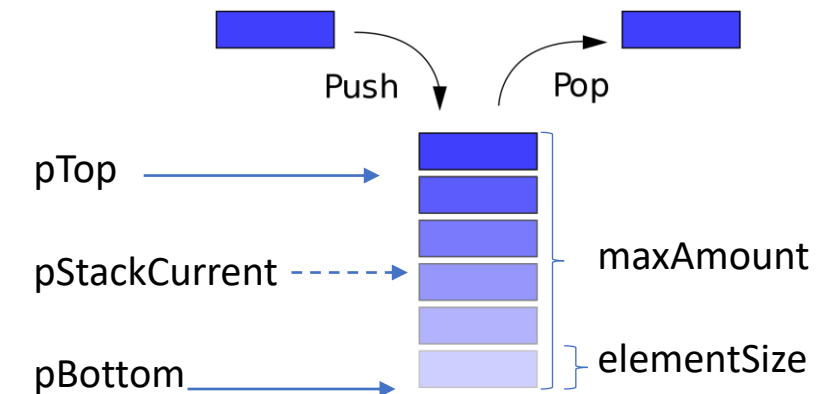


Stack: abstrakter Datentyp

Nur wenige Programmiersprachen bieten ein Stackarray als Basisdatentyp an – muss meist auf Basis des Arrays selbst implementiert werden.

```
typedef void stackElement_t;
```

```
typedef struct  
{  
    stackElement_t *pBottom;  
    stackElement_t *pTop;  
    stackElement_t *pStackCurrent;  
    size_t      elementSize;  
    size_t      maxAmount;  
} myStack_t; // Type of meta data
```





Stack: abstrakter Datentyp

```
myStack_t *StackNew (size_t i_elementSize, size_t i_maxAmount)
{
    myStack_t *pStack = malloc (sizeof (myStack_t)); // Create stack meta
    if (NULL == pStack) return NULL;                // data structure

    pStack->pTop= malloc (i_elementSize *i_maxAmount); // full actual stack
    if (NULL == pStack->pTop)
    {
        free (pStack);
        return NULL;
    }
    pStack->pBottom = pStack->pTop + (i_elementSize * i_maxAmount);
    pStack->pStackCurrent = pStack->pBottom;
    pStack->elementSize = i_elementSize;
    pStack->maxAmount = i_maxAmount;
    return pStack;
}
```

Stack: abstrakter Datentyp

```
int StackDestroy (myStack_t *io_pStack)
{
    if (NULL == io_pStack) return EXIT_FAILURE; // no stack

    free (io_pStack->pTop);                      // free stack memory

    io_pStack->pBottom = NULL;                    // invalidate pointers
    io_pStack -> pTop=NULL;

    free (io_pStack);                            // free meta data memory

    return EXIT_SUCCESS;
}
```

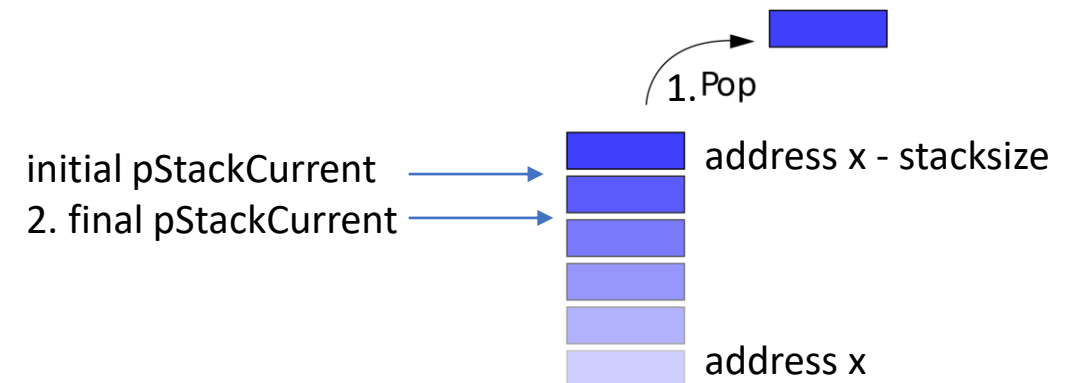
Stack: abstrakter Datentyp: Push & Pop

```
int Pop (myStack_t *io_pStack, stackElement_t *io_pValue) // returns topmost value
{
    if ((NULL != io_pStack) && (io_pStack->pStackCurrent != io_pStack->pBottom))
    { // SP not NULL AND stack not empty

        // copy topmost value to io-parameter-pointer
        memcpy (io_pValue, io_pStack->pStackCurrent, io_pStack->elementSize);

        // increase current stackpointer
        io_pStack->pStackCurrent += io_pStack->elementSize;

        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```

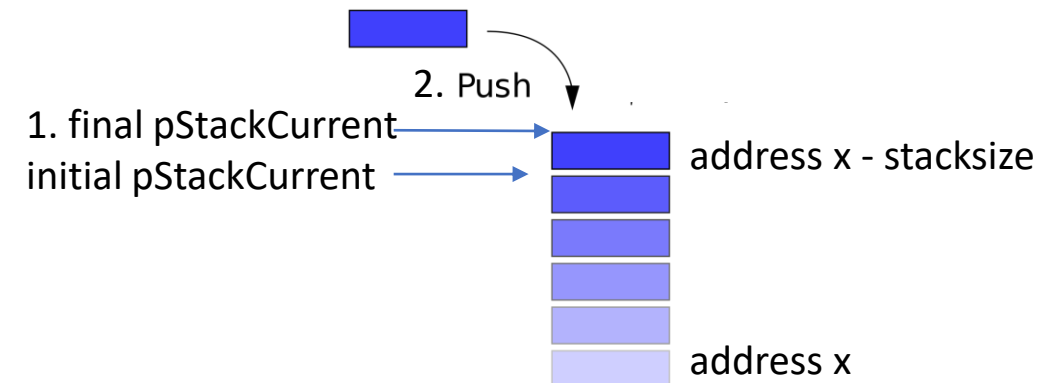


Stack: abstrakter Datentyp: Push & Pop

```
int Push (myStack_t *io_pStack, stackElement_t *io_pValue) // places value on top
{
    if ((NULL != io_pStack) && // SP not NULL AND stack not full
        (io_pStack->pStackCurrent >= ((io_pStack->pBottom)-(io_pStack->elementSize*io_pStack->maxAmount))))
    {
        // decrease stackpointer
        io_pStack->pStackCurrent -= io_pStack->elementSize;

        // copy value to current stackpointer (topmost)
        memcpy (io_pStack->pStackCurrent, io_pValue, io_pStack->elementSize);

        return EXIT_SUCCESS;
    }
    else return EXIT_FAILURE;
}
```



Stack als abstrakter Datentyp

Angebote Funktionen (Interface) erlauben die Benutzung des Arrays nur in limitierter Weise (Stack: Push, Pop).

Kein Direktzugriff über die Indizes des Arrays!

- Datenstruktur Stack ist nicht die gleiche wie ein Array
- Ein Stack kann auch auf Basis einer verketteten Liste aufgebaut werden!

-> Die Datenstruktur *Stack* ist ein klassisches Beispiel für einen **abstrakten Datentyp** (unterstützt Zugriffsfunktionen auf Basis einer anderen Datenstruktur)

Stacks in Aktion: lint (lexikale Analyse)

Bsp.: Ein Programm, das die syntaktische Korrektheit von JavaScript-Programmzeilen prüft, und zwar auf korrekte öffnende und schließende Klammern (), { } und [] hin

Es können 3 fehlerhafte Situationen auftreten:

```
(var x = 2;           // Syntax Error Typ #1: öffnende ohne schließende Klammer  
  
var x = 2;)          // Syntax Error Typ #2: schließende ohne vorausgehende öffnende Klammer  
  
(var x = [1, 2, 3]);  // Syntax Error Typ #3: schließende Klammer hat anderen Typ als öffnende Klammer
```

Stacks in Aktion: lint (lexikale Analyse)

Start:

- mit leerem Stack
- wir lesen zeilenweise jedes Zeichen von links nach rechts zur Prüfung ein



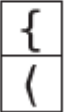
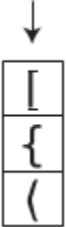
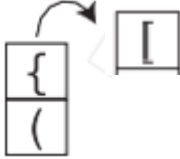
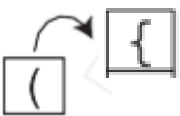
Stacks in Aktion: lint (lexikale Analyse)

Regeln: Das nächste Zeichen ist:

1. keine Klammer (rund, eckig oder geschweift): ignorieren und weiter.
2. öffnende Klammer: auf den Stack pushen.
3. schliessende Klammer: oberstes Element vom Stack holen (Pop):
 - keine Elemente mehr im Stack: schliessende Klammer ohne vorausgehende öffnende! (Syntax Error Typ #2)
 - Daten im Stack, aber die schliessende Klammer passt nicht zum obersten Element: Syntax Error Typ #3
 - schliessende Klammer paßt zum obersten Element auf dem Stack: öffnende Klammer erfolgreich geschlossen.
4. Zeilenende:
 - Stack enthält noch Element(e): öffnende ohne schliessende Klammer (Syntax Error Typ #1)
 - Stack ist leer: kein Fehler

Stacks in Aktion: lint (lexikale Analyse)

Beispiel: `(var x = {y: [1, 2, 3]})`

- Wir lesen die Zeile von links nach rechts, die erste Klammer geht auf den Stack 
- Wir ignorieren die Nicht-Klammer-Zeichen `var x =` 
- Wir pushen die öffnende Klammer auf den Stack: 
- Wir ignorieren `y:`
- Wir pushen die nächste öffnende Klammer auf den Stack: 
- Wir ignorieren `1, 2, 3`
- Wir treffen auf eine schließende Klammer. Wir entnehmen das oberste Element des Stack und kontrollieren auf Korrespondenz. Gegeben. 
- Wir treffen auf eine schließende Klammer. Wir entnehmen das oberste Element des Stack und kontrollieren auf Korrespondenz. Gegeben. 
- Wir treffen auf eine schließende Klammer. Wir entnehmen das oberste Element des Stack und kontrollieren auf Korrespondenz. Gegeben.
- Wir sind am Zeilenende und prüfen den Füllstand des Stack. Stack ist leer – kein Fehler!

Stacks in Aktion: lint (lexikale Analyse)

Die Implementierung iteriert über jedes Zeichen der Zeile, und befüllt den Stack maximal mit allen Zeichen.

Die Zeile `(var x = { y: [1, 2, 3]) }` würde den Fehler *“Ungültige schließende Klammer) an Position 25”* erzeugen (Schritt 8).

Die Zeile `(var x = { y: [1, 2, 3] }` würde den Fehler *“(hat keine schließende Klammer”* erzeugen (Schritt 10).

Stacks sind ideal zur Bearbeitung von Daten in umgekehrter Reihenfolge (LIFO).

Weiteres Bsp: „undo“-Funktion in einem Textverarbeitungsprogramm



Die Queue

- ist eine andere Datenstruktur, um temporäre Daten zu verwalten.
- ist ebenfalls ein abstrakter Datentyp
- folgt prinzipiell den gleichen Regeln wie der Stack, aber mit anderer Entnahmeregel (FIFO – First In, First Out).

Bsp.: Leute an einer Kinokasse.

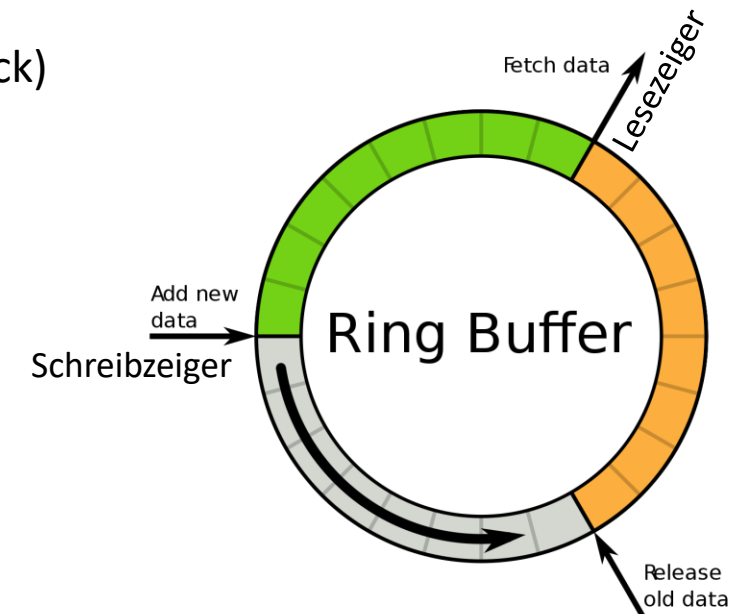


Anderer Satz von Restriktionen:

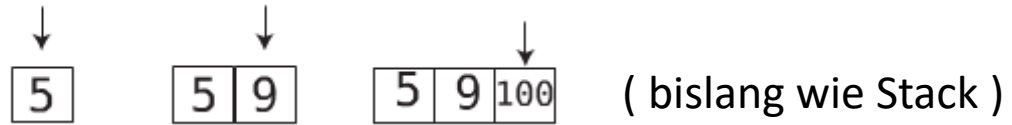
- Daten können nur am Ende hinzugefügt werden (identisch zum Stack)
- Daten können nur am Anfang entnommen/ gelöscht werden (Gegenteil vom Stack)
- Nur das erste Element kann gelesen werden (Gegenteil vom Stack)

Auch Queues lassen sich als Array (mittels Ringbuffer) realisieren, wenn ihre maximale Größe vorab festgelegt werden kann!

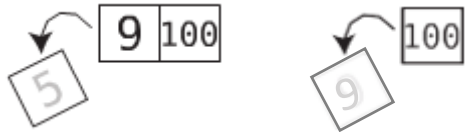
Schreibzeiger startet, Zeiger dürfen sich nie überholen!



Queue in Aktion



Entnahme:



Queue: Implementierung

```
typedef void queueElement_t; // Typ eines QueueElements
```

```
typedef struct  
{  
    queueElement_t *pBottom; // start pointer  
    queueElement_t *pRead;   // read pointer  
    queueElement_t *pWrite;  // write pointer  
    bool full;                // true when full  
    bool empty;               // true when empty  
    size_t elementSize;       // size of single element  
    size_t maxAmount;         // maximum amount of elements  
  
} myQueue_t; // Type of meta data (Verwaltungsstruktur)
```

Queue: Implementierung *QueueNew()*

// Creates a queue of size *maxAmount* incl. meta data, returns pointer to meta data

```
myQueue_t * QueueNew(size_t i_maxAmount)
{
    myQueue_t * pQueue=malloc(sizeof(myQueue_t)); // allocate meta data
    if (NULL == pQueue) return NULL; // out of memory

    pQueue->pBottom = malloc(i_maxAmount * sizeof(queueElement_t)); // allocate queue
    if (NULL == pQueue->pBottom)
    {
        free(pQueue); return NULL; // out of memory
    }

    pQueue->pRead=pQueue->pWrite=pQueue->pBottom; // all pointers on start
    pQueue->full=false; pQueue->empty=true; // empty queue, not full
    pQueue->maxAmount=i_maxAmount;
    return pQueue; // return pointer to meta data
}
```

Queue: Implementierung *QueueDestroy()*

```
// free Queue via pointer to meta data
int QueueDestroy (myQueue_t *io_pQueue)
{
    if (NULL == io_pQueue) return EXIT_FAILURE; // no valid queue

    free (io_pQueue->pBottom); // free actual Queue
    io_pQueue->pBottom=NULL; // invalidate start pointer

    free(io_pQueue); // free meta data

    return EXIT_SUCCESS;
}
```

Queue: Implementierung *Empty()* und *Full()*

```
bool IsQueueEmpty (myQueue_t * i_pQueue)
{
    if (NULL != i_pQueue) return i_pQueue->empty; // returns empty-state
    else return true; // invalid, returns "empty"
}
```

```
bool IsQueueFull (myQueue_t * i_pQueue)
{
    if (NULL != i_pQueue) return i_pQueue->full; // returns full-state
    else return true; // invalid, returns "full"
}ca
```

Queue: Implementierung *Enqueue()*

```
// write element to queue
int EnQueue(myQueue_t * i_pQueue, queueElement_t i_value) // queue pointer, value to be written
{
    int back = EXIT_FAILURE;

    if ((NULL != i_pQueue) && (i_pQueue->full != true)) // Queue valid and not full
    {
        i_pQueue->empty=false;    // after writing: not (longer) empty
        *(i_pQueue->pWrite)=i_value; // write value
        back=EXIT_SUCCESS;

        // pointer reached end of array?
        if (i_pQueue->pWrite == (i_pQueue->pBottom + i_pQueue->maxAmount-1))
        {
            i_pQueue->pWrite = i_pQueue->pBottom; // set pointer back to start
        }
        else i_pQueue->pWrite++;                // increase write pointer

        // read pointer reached?
        if (i_pQueue->pRead == i_pQueue->pWrite) i_pQueue->full = true; // queue full,
                                                                    // no more writes
    }
    return back;
}
```

Queue: Implementierung *Dequeue()*

```
// read next element, returns element
queueElement_t DeQueue(myQueue_t * i_pQueue)
{
    queueElement_t back;           // may be invalid

    if ((NULL != i_pQueue) && (i_pQueue->empty != true)) // Queue valid and not empty
    {
        i_pQueue->full=false;    // after reading -> not full
        back = *(i_pQueue->pRead); // read value

        // pointer reached end of array?
        if (i_pQueue->pRead == (i_pQueue->pBottom + i_pQueue->maxAmount-1))
        {
            i_pQueue->pRead = i_pQueue->pBottom; // set pointer back to start
        }
        else i_pQueue->pRead++;                // increase read pointer

        // write pointer reached?
        if (i_pQueue->pRead == i_pQueue->pWrite) i_pQueue->empty = true; // queue empty
        // no more reads
    }
    return back;
}
```



Beispiele: Druckjobs, Zwischenbuffer bei Kommunikationsprotokollen, ...

- perfekt für das Zwischenspeichern asynchroner Aufträge und Anfragen - garantieren die Bearbeitung in der Reihenfolge des Eintreffens
- geeignet für die Modellierung von Szenarien der realen Welt:
 - Verkehr an Ampelkreuzungen,
 - Startfreigabe für Flugzeuge,
 - generell Warteschlangenmodelle...

Ausblick:

Der Stack bildet auch die Basis für Rekursionen – und damit für das nächste Thema...

1. Wenn Sie eine Software für ein Call Center schreiben sollten, die Anrufer zunächst zwischenspeichert und sie dann “dem nächsten freien Bearbeiter” zuweist —würden Sie einen Stack oder eine Queue nutzen?

A: Stack

B: Queue

2. Wenn Sie Zahlen in folgender Reihenfolge auf einen Stack abgelegt hätten: 1, 2, 3, 4, 5, 6, und dann mit Pop zwei Elemente herausholen würden, welches wäre dann die nächste Zahl oben auf dem Stack?

A: 2

B: 3

C: 4

D: 5

E: 6

3. Wenn Sie Zahlen in folgender Reihenfolge in eine Queue gestellt hätten : 1, 2, 3, 4, 5, 6, und dann 2 Zahlen wieder entnommen hätten - welche Zahl wäre dann als nächste an der Reihe?

A: 2

B: 3

C: 4

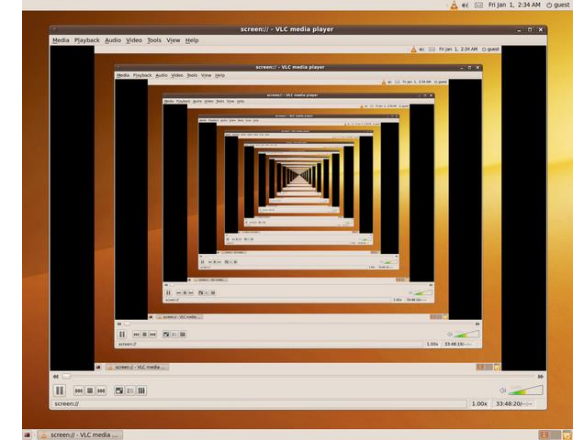
D: 5

E: 6

Rekursive Algorithmen sind ein Schlüsselkonzept der Informatik.

Was passiert beim Aufruf der Funktion `blah()`?

```
function blah()  
{  
    blah();  
}
```



Von vlc team, ubuntu, Hidro (talk) - Eigenes Werk, GPL,
<https://commons.wikimedia.org/w/index.php?curid=8879368>

Rekursion bedeutet, dass sich eine Funktion selbst aufruft (endlose Rekursion ist komplett nutzlos).

Rekursionen statt Schleifen

Stellen Sie sich vor, Sie arbeiten für die NASA und müssen einen Countdown für den Start eines Raumschiffs programmieren.

Die zu schreibende Funktion soll eine Zahl als Input bekommen — wie z.B. 10— und die Zahlen von 10 bis 0 anzeigen.

Aufruf: `countdown (10);`

```
void Countdown (int start)
{
    for (int i=start; i>=0; i--)    printf ("%d\n", i);
}
```

Rekursionen statt Schleifen

Erster Versuch mit Rekursion:

Aufruf: `countdown (10);`

```
void Countdown (int number)
{
    printf ("%d\n", number);
    Countdown (number-1);
}
```

Perfekt?

Nur dass Sie Rekursion verwenden *können*, heißt nicht, daß Sie unbedingt Rekursion verwenden sollten!

Rekursionen statt Schleifen

Weiter mit dem Beispiel:

Unsere Lösung ist nicht perfekt, sie gibt endlos negative Zahlen aus...

Wir brauchen eine Möglichkeit, den Countdown bei 0 zu beenden und davor zu schützen, endlos zu laufen!

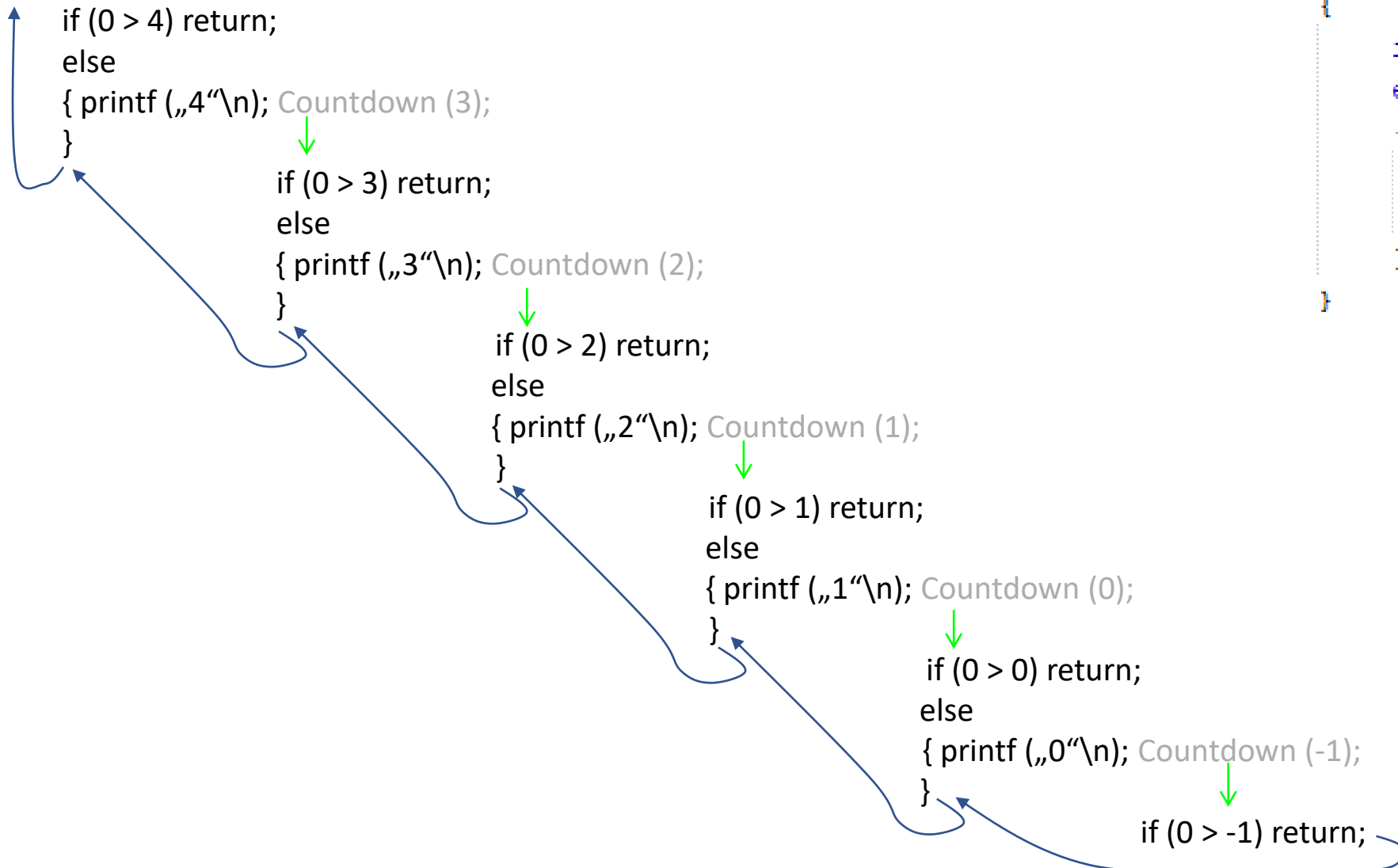
```
void Countdown (int number)
{
    printf ("%d\n", number);
    Countdown (number-1);
}
```

Zweiter Versuch:

```
void Countdown (int number)
{
    if (0 > number) return; // Abbruchkriterium - nötig wie bei jeder Schleife
    else
    {
        printf ("%d\n", number);
        Countdown (number-1);
    }
    // ← hier wird nach return fortgesetzt!
}
```

Rekursionen statt Schleifen

Countdown (4):



```
void Countdown (int number)
{
    if (0 > number) return; //
    else
    {
        printf ("%d\n", number);
        Countdown (number-1);
    } // ← hier
}
```


Rekursiven Code lesen

Zwei Fähigkeiten werden benötigt für das Verständnis rekursiven Codes:

- rekursiven Code lesen
- rekursiven Code schreiben

Bsp. für Lesen: Fakultäten berechnen (n!)

Die Fakultät von 3 ist:

$$3 * 2 * 1 = 6$$

Die Fakultät von 5 ist:

$$5 * 4 * 3 * 2 * 1 = 120$$

```
int Factorial (int number)
{
    if (1 == number) return 1;
    else return number * Factorial(number - 1);
}
```

Rekursiven Code lesen

Empfehlung für das Lesen von rekursivem Code:

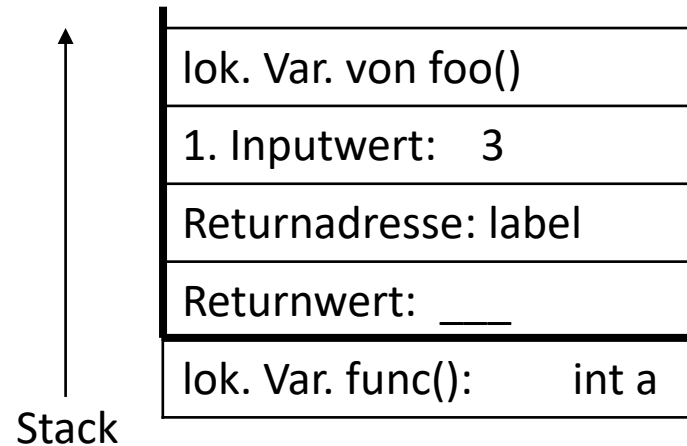
1. Das Abbruchkriterium identifizieren.
2. Für das Abbruchkriterium durch die Funktion gehen.
3. Den Fall vor dem Abbruchkriterium identifizieren.
Das ist der Fall vor dem letzten Fall (number=2).
4. Für diesen vorletzten Fall durch die Funktion gehen.
5. Diesen Prozess wiederholen durch *Identifizieren des Falles vor dem gerade analysierten*, und für diesen Fall durch die Funktion gehen.

```
int Factorial (int number)
{
    if (1 == number) return 1;
    else return number * Factorial(number - 1);
}
```

Rekursion aus Sicht des Computers

Nutzung des Stack für die Rückkehradressen und Aufrufparameter:

```
void func()  
{ int a;  
  a=foo (3);  
  label: ...
```



```
int Factorial (int number)  
{  
    if (1 == number) return 1;  
    else return number * Factorial(number - 1);  
}
```

Factorial(3) wird zuerst aufgerufen. Bevor sie ausgeführt wird...

Factorial(2) wird aufgerufen. Bevor sie ausgeführt wird...

Factorial(1) wird aufgerufen.

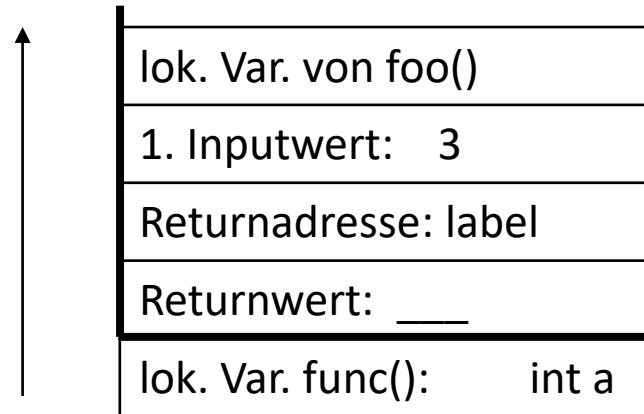
Factorial(1) kehrt zuerst *zurück*.

Factorial(2) wird abgearbeitet auf Basis des Ergebnisses von Factorial(1).

Zuletzt, wird Factorial(3) abgearbeitet auf der Basis des Resultats von Factorial(2).

blah() rief sich selbst endlos auf. Was bedeutet das für den Callstack?

Rekursion aus Sicht des Computers



Im Fall endloser Rekursion legt der Rechner (push) die gleiche Rückkehradresse + Parameter + lokale Variable wieder und wieder auf den Stack.

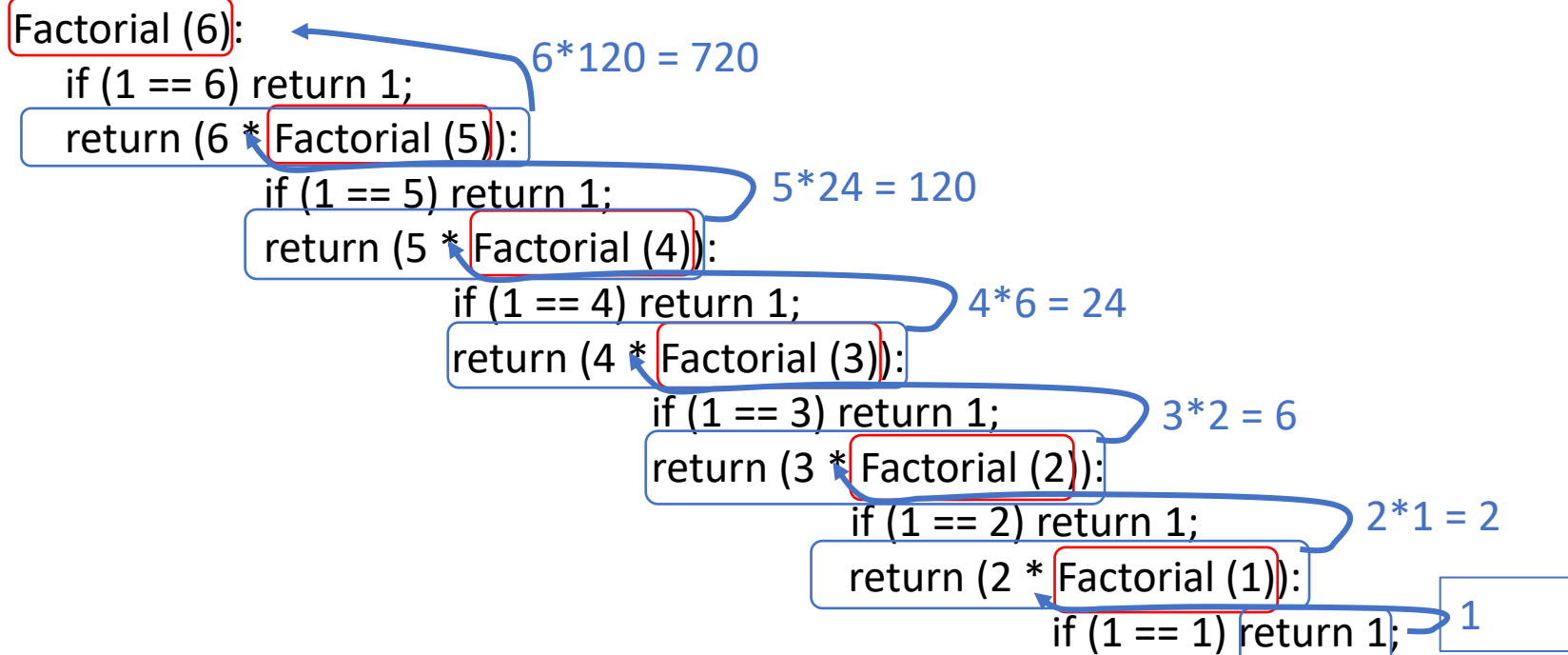
Der Stack wächst und wächst, bis für den Prozess/ Thread kein Platz mehr verfügbar ist, um all die Daten zu halten.

Laufzeitfehler “*Stack overflow*”:

Der Computer bricht die Ausführung ab und meldet “Ich verweigere den erneuten Aufruf dieser Funktion, weil ich keinen Speicher mehr habe!”

Rekursive Kategorie: Berechnungen

```
int Factorial (int number)
{
    if (1 == number) return 1;
    else return number * Factorial(number - 1);
}
```



Rekursion in Aktion

Ohne Rekursion schwer lösbare Probleme:

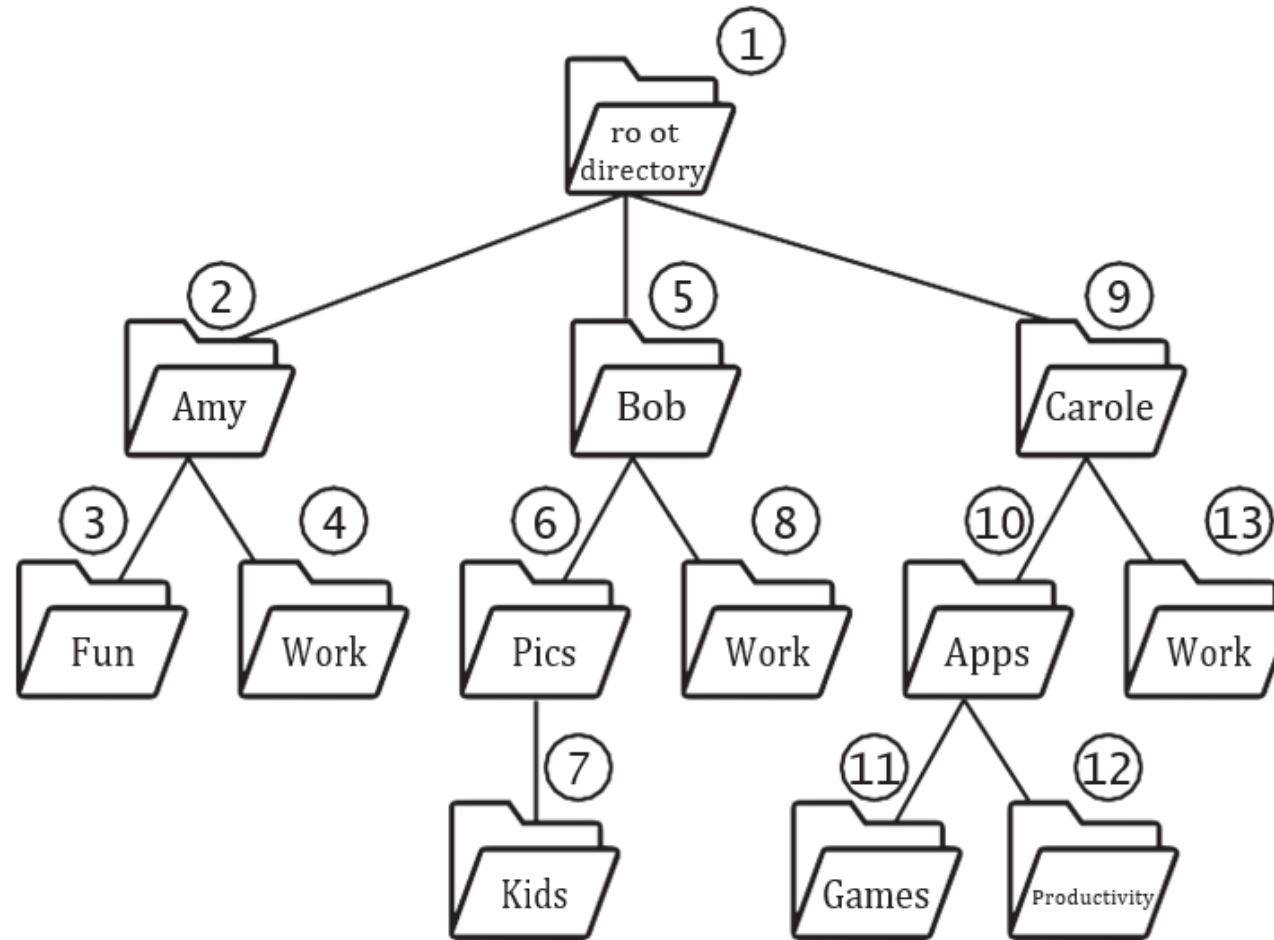
... wenn wir in geschachtelte Tiefen eines Problems eintauchen müssen, ohne vorab zu wissen, wieviele Ebenen es sein werden.

Bsp.: Traversierung durch ein Filesystem (z.B., um alle Namen aller Subdirectories und Files auszugeben)

- Auflisten aller Unterverzeichnisse *innerhalb* aller Unterverzeichnisse eines Directories...

Wenn wir so tief gehen wollen wie die Subdirectories – dann glänzt Rekursion wirklich!

Rekursion in Aktion



Zusammenfassung

Rekursion ist oft eine gute Wahl für einen Algorithmus, der nicht weiß, wieviele Ebenen tief er in ein Problem hinabsteigen muss...

1. Die Funktion unten druckt jede zweite Zahl von einer unteren bis zu einer oberen Grenze aus.
Wenn z.B. die untere 0 und die obere 10 ist, würde sie drucken:

0
2
4
6
8
10

Identifizieren Sie das Abbruchkriterium in dieser Funktion:

```
A: void print_every_other (int low, int high)
   {
B:     if (low > high) return;
C:     printf ("%d\n", low);
D:     print_every_other(low + 2, high);
   }
```

2. Ihr Kumpel hat mit Ihrem Computer herumgespielt und Ihre Fakultätsberechnung geändert, so dass sie $\text{Factorial}(n - 2)$ statt $\text{Factorial}(n - 1)$ rechnet. Schätzen Sie mal, was passieren wird, wenn wir die Fakultät von 6 mit dieser Funktion berechnen:

```
int Factorial (int n)
{
    if (n == 1) return 1;
    printf("%d ",n);
    return (n * Factorial(n - 2));
}
```

- A: Ergebnis ist 720
- B: Ergebnis ist 600
- C: Ergebnis ist 48
- D: Ergebnis ist 1
- E: Funktion läuft endlos...

3. Gegeben ist eine Funktion, der wir 2 Parameter (low und high) übergeben. Die Funktion gibt die Summe aller Zahlen von low bis high zurück. Ist z.B. low=1 und high=10, wird die Funktion die Summe aller Zahlen von 1 bis 10 zurückgeben, also 55. Unserem Code fehlt jedoch das Abbruchkriterium, und er wird endlos laufen! Fixen Sie den Code durch Hinzufügen des korrekten Abbruchkriteriums:

```
int Sum(int low, int high)
{
    return (high + Sum(low, high - 1));
}
```

Rekursiven Code schreiben lernen

Rekursive Muster (patterns) und manche Technologien helfen dabei, “rekursiv programmieren” zu lernen.

Kategorien geeigneter Probleme:

- z.B. Algorithmen mit dem Ziel, eine **Aufgabe wiederholt auszuführen** (wie der NASA Raumschiff- Countdown-Algorithmus).

Für Probleme dieser Kategorie gilt:

- die **letzte Codezeile** der Funktion ist ein **einfacher, erneuter Aufruf der Funktion selbst!**

Anderes Beispiel für wiederholte Ausführung:

Schreiben Sie einen “in place”- Algorithmus, der ein Array von Zahlen bekommt, und jede dieser Zahlen vom Index 0 an verdoppelt.

```
void Double_array (int array[], int size)    // unfertig
{
    Double_array(array, size-1);    // final line will be the recursive call
}

void Double_array(int array[], int size)    // immer noch unfertig
{
    array[0] *= 2;    // we need to add the code that will actually double the number
                     // but how to increase the index?
    Double_array(array, size-1);
}
```

Rekursiver Trick: Übergabe zusätzlicher Parameter

Der nächste Trick... Wir übergeben zusätzliche Parameter!

```
void Double_array(int array[], int size, int index) // immer noch unfertig
{
    // Aufruf jetzt mit Startindex:

    Double_array (array, size-1, 0);
}
```

Code:

```
void Double_array (int array[], int size,
                  int index)
{
    // Abbruchkriterium fehlt noch
    array[index] *= 2;
    Double_array (array, size-1, index + 1);
}
```

Mit Abbruchkriterium:

```
void Double_array (int array[], int size,
                  int index)
{
    if (0 >= size) return; // Abbruchkriterium
    array[index] *= 2;
    Double_array (array, size-1, index + 1);
}
```

Rekursive Kategorie: Berechnungen

Zweite generelle Kategorie:

- Durchführung einer **Berechnung** basierend auf einem **Teilproblem**

Beispiele

(diese Funktionen bekommen einen Input, und geben das Ergebnis der Berechnung auf diesem Input zurück):

- Eine Funktion, die die **Summe mehrerer Zahlen** zurückgibt, oder
- Eine Funktion, die den **größten Wert in einem Array findet**.
- Die Fakultät von 6 ist:
 $6 * 5 * 4 * 3 * 2 * 1$

Wir nähern uns diesem Problem des Berechnens der Fakultät basierend auf einem Teilproblem.
Ein *Teilproblem* ist eine Variante des Problems basierend auf kleineren Inputdaten.

Die Berechnung der Fakultät von 6 basiert auf dem Teilproblem der Berechnung der Fakultät von 5!

Rekursive Kategorie: Berechnungen

Schlüsselzeile: *return number * Factorial (number - 1),*
berechnet das Ergebnis als Zahl, multipliziert mit dem Teilproblem von *Factorial (number-1)*.

Zwei Ansätze für Berechnungen

Für das Schreiben der Berechnungsfunktion haben wir 2 potentielle Ansätze:

- wir können eine Lösung von “bottom up” (iterativer Ansatz) bauen, oder
- wir können das Problem “top down” angehen, indem wir die Berechnung des Problems auf seinem Teilproblem aufsetzen

Informatikliteratur benutzt die Begriffe *bottom up* und *top down* in Bezug auf Algorithmenstrategien.

Wir können auch für die bottom-up-Strategie die Rekursion benutzen. Dafür müssen wir den Trick der Übergabe von extra Parametern einsetzen:

```
int Factorial (int n, int leastFactor, int intermedResult)    Aufruf: int x = factorial (6, 1, 1);
{
    if (leastFactor > n) return (intermedResult);
    return Factorial (n, leastFactor + 1, intermedResult *leastFactor);
}
```

Die Bottom-up-Rekursion verfolgt den gleichen Ansatz wie eine iterative Schleife – aber für Top-down brauchen wir die Rekursion!

Top-Down-Rekursion: Lösung von Teilproblemen

Der top-down-Ansatz *bietet eine neue mentale Strategie für die Behandlung eines Problems.*

Wenn wir top-down vorgehen, verlagern wir sozusagen das Problem...

Die Schlüsselzeile unserer Fakultätsimplementierung ist: `return number * Factorial(number - 1);`

Mit dieser Codezeile sagen wir:

“Es kümmert mich nicht, wie die Fakultäts-Funktion unter der Haube funktioniert.
Alles, was ich weiß, ist, dass sie funktioniert.”

Wenn wir also die Fakultäts-Funktion nutzen können, um das Ergebnis des Teilproblems zu erhalten, können wir das Ergebnis des Teilproblems einfach mit weiteren Zahlen multiplizieren, um das korrekte Ergebnis des eigentlichen Problems zu erhalten.

Top-Down-Rekursion: Lösung von Teilproblemen

Bei Behandlung von Top-Down-Problemen hilft es, folgende 3 Gedanken im Hinterkopf zu behalten:

- Stellen Sie sich vor, die Funktion, die Sie implementieren, hätte jemand anders geschrieben
- Was ist das Teilproblem des Problems?
- Beobachten Sie, was passiert, wenn Sie die Funktion auf das Teilproblem anwenden, und von da aus weitergehen.

Top-Down-Rekursion : Lösung von Teilproblemen

Funktion “Sum” soll alle Zahlen in einem gegebenen Array summieren.
Für das Array [1, 2, 3, 4, 5] wird sie 15 zurückgeben.

- Wir stellen uns vor, die “Sum”-Funktion sei bereits implementiert und würde funktionieren.
- Welches Teilproblem nutzt die Sum-Funktion?

Das Teilproblem hier ist das Array [2, 3, 4, 5] —alle Zahlen des Arrays ausser der aktuell (!) Ersten.

In Pseudocode: **return array[0] + sum(the remainder of the array)**

In C: return (array[0] + sum(array+1, size -1));

Noch ohne Abbruchkriterium:

```
int Sum (int array[], int size)
{
    return (array[0] + Sum(array+1, size- 1));
}
```

Abbruchkriterium hinzugefügt:

```
int Sum (int array[], int size)
{
    if (1 == size) return array[0];
    return (array[0] + Sum(array+1, size- 1));
}
```

Top-Down-Rekursion : Lösung von Teilproblemen

Anderes Beispiel: Funktion "Reverse()" soll einen String umkehren, durch Tausch der korrespondierenden Zeichen. Wenn sie "abcde" erhält, wird sie "edcba" zurückgeben.

1. **Teilproblem:** Nächstkleineres Teilproblem des Problems suchen: dieses sei "bcd".
2. Funktion "Reverse()" ist bereits verfügbar und unser Teilproblem ist "bcd": wir können "bcd" umkehren, und "dcb" zurückerhalten. Aufruf: Reverse("test", 0, strlen("test"));
3. So können wir schreiben:

```
void Reverse(char str1[], int i, int len)
{
    char temp;

    temp = str1[i]; // swap opposite chars
    str1[i] = str1[len - i - 1];
    str1[len - i - 1] = temp;

    Reverse(str1, i + 1, len);
}
```

Abbruchkriterium hinzugefügt:

```
void Reverse(char str1[], int i, int len)
{
    char temp;
    if (i == len/2) return;
    temp = str1[i]; // swap opposite chars
    str1[i] = str1[len - i - 1];
    str1[len - i - 1] = temp;

    Reverse(str1, i + 1, len);
}
```

Top-Down-Rekursion : Lösung von Teilproblemen

Weiteres Beispiel: Funktion soll die Anzahl von 'x's in einem gegebenen String zählen.

Wird der String "axbxcxd" übergeben, gibt sie "3" zurück, weil das Zeichen 'x' dreimal enthalten ist.

1. **Teilproblem:** Originaler String minus dem ersten Zeichen. Für "axbxcxd" ist das Teilproblem also "xbxcxd".
2. Wenn wir die Funktion für das Teilproblem aufrufen, durch den Aufruf `Count_x ("xbxcxd")`, erhalten wir "3". Dazu müssen wir dann "1" addieren, falls das aktuelle Zeichen unseres Strings ein 'x' ist ...
3. Wir können also schreiben:

Abbruchkriterium hinzugefügt:

```
int Count_x(char *string)
{
    if ('x'==string[0]) return (1 + Count_x(string+1));
    else return (Count_x(string+1));
}
```

```
Count_x(char *string)
{
    if (0 == strlen (string)) return 0;
    if ('x'==string[0]) return (1 + Count_x(string+1));
    else return (Count_x(string+1));
}
```

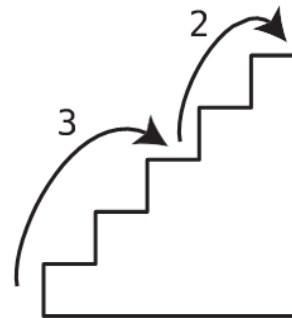
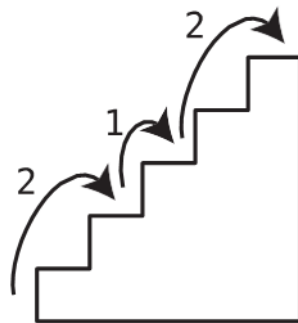
Schöne Rekursionsprobleme

Berühmte Frage:

Angenommen, wir haben eine Treppe mit N Stufen, und eine Person hat die Fähigkeit, eine, zwei oder drei Stufen gleichzeitig zu nehmen:

Wieviele verschiedene “Wege” kann jemand bis zur Spitze nehmen? Schreiben Sie eine Funktion, die das für N Stufen berechnet.

Die Abbildung zeigt nur 3 von vielen möglichen Wegen, eine 5-stufige Treppe hinaufzuspringen...



Schöne Rekursionsprobleme

Offensichtlich gibt es, wenn nur eine Stufe da ist, nur einen möglichen Weg.

Bei 2 Stufen gibt es schon 2 Möglichkeiten. Die Person kann zweimal eine Stufe nehmen, oder zwei Stufen auf einmal. Aufschreiben kann man das so:

1, 1
2

Bei einer Treppe von 3 Stufen hat man 4 verschiedene Möglichkeiten:

1, 1, 1
1, 2
2, 1
3

Bei 4 Stufen gibt es 7 Optionen:

1, 1, 1, 1
1, 1, 2
1, 2, 1
1, 3
2, 1, 1
2, 2
3, 1

Schöne Rekursionsprobleme

Wie würden wir diesen Code schreiben, um alle Wege zu berechnen?

Teilproblem für eine 11-stufige Treppe: 10- stufige Treppe. Heißt, alle Wege zählen, um zur 10. Stufe zu gelangen, und von dort noch einen Schritt zur Spitze.

Lösung ist offensichtlich nicht komplett – man kann auch von der 9. Stufe direkt zur 11. Stufe springen...

Außerdem ist es möglich, von Stufe 8 zu Stufe 11 zu springen, weil man 3 Stufen auf einmal nehmen kann.

-> Die Anzahl der Schritte zur Spitze ist mindestens die Summe aller Wege zu den Stufen 10, 9 und 8...

Für n Stufen ist die Nummer der Wege: **Anzahl von Wegen nach (n - 1) + Anzahl von Wegen nach (n - 2) + Anzahl von Wegen nach (n - 3)**

```
int CountWays (int n)
{
    CountWays (n - 1) + CountWays (n - 2) + CountWays (n - 3);
}
```

Schöne Rekursionsprobleme

Mit Abbruchkriterium:

```
int CountWays (int n)
{
    CountWays (n - 1) + CountWays (n - 2) + CountWays (n - 3);
}
```

Das Ergebnis von CountWays(0) kann beliebig sein – es gibt unendlich viele Möglichkeiten, keine Stufe hochzusteigen.

Das Ergebnis von CountWays(1) muss 1 sein, so starten wir mit diesem Abbruchkriterium: **return 1 if n == 1**

CountWays(2) muss 2 zurückgeben. Es lässt sich (der **Formel oben rechts** folgend) berechnen aus:

CountWays(1) + CountWays(0) + CountWays(-1) ,

wenn wir dafür sorgen, dass auch CountWays(0) 1 zurückgibt, und CountWays(-1) 0 zurückgibt.

Dann enden wir bei CountWays (2) := 2.

So können wir das folgende Abbruchkriterium hinzufügen:

```
if (n<0)    return 0;
if ((1 == n) || (0 == n)) return 1;
```

Und unsere komplette Funktion kann implementiert werden als:

```
int CountWays (int n)
{
    if (n<0)    return 0;
    if ((1 == n) || (0 == n)) return 1;
    return (CountWays (n-1)+CountWays (n-2)+CountWays (n-3)) ;
}
```

Zusammenfassung

Sind sind jetzt gerüstet mit Tricks und Techniken, die den Lernprozess leichter machen.

Obwohl Rekursion ein großartiges Tool für die Lösung einer Reihe von Problemen ist, kann sie Ihren Code doch extrem verlangsamen, wenn Sie nicht aufpassen.

In der nächsten Lektion werden wir versuchen, trotz Rekursion unseren Code lesbar und schnell zu halten.



1. Schreiben Sie eine rekursive Funktion *ArraySum()*, die aus einem Array von Zahlen deren Summe ermittelt.

```
int main(int argc, char** argv)
{
    int array[]={1,1,10,1,1};
    printf ("Summe=%d\n", ArraySum(array, sizeof(array)/sizeof(array[0])));
    return (EXIT_SUCCESS);
}
```



2. Schreiben Sie eine rekursive Funktion *EvenArray()*, die ein Array von Zahlen bekommt und ein neues Array zurückgibt, das nur gerade Zahlen enthält.

```
int main(int argc, char** argv)
{
    int array[]={2,1,10,6,3};
    int* pNewArray=malloc(0);
    int newSize=0;
    EvenArray (array, sizeof(array)/sizeof(array[0]), pNewArray, &newSize);
    while (newSize)
    {
        printf ("newArray[%d]=%d\n", newSize-1, pNewArray[newSize-1]);
        newSize--;
    }
    free(pNewArray);
    return (EXIT_SUCCESS);
}
```