

*Every set of cards made for any formula will at any future time recalculate that formula with whatever constants may be required.*

*Thus the Analytical Engine will possess a library of its own.*

*Charles Babbage (1864)*

## Unterprogramme - Prozeduren

---

*[...] procedures are one of the most powerful features of a high level language, in that they both simplify the programming task and shorten the object code.*

*C. A. R. Hoare (1981)*

# Unterprogramme - Funktionen

- Das verwenden von Funktionen entspricht einer Modularisierung des Programm-Codes.
- Warum Modularisierung?
  - ◆ Bessere Lesbarkeit
    - ⇒ Der Quellcode eines Programms kann schnell mehrere tausend Zeilen umfassen. Beim Linux Kernel: über 15 Millionen Zeilen!
  - ◆ Wiederverwendbarkeit
    - ⇒ Wiederkehrende Probleme müssen nur einmal gelöst werden.
  - ◆ Wartbarkeit
    - ⇒ Fehler lassen sich durch die Modularisierung leichter finden und beheben. Darüber hinaus ist es leichter, weitere Funktionalitäten hinzuzufügen oder zu ändern.

# Unterprogramme

- Ein **Unterprogramm** ist ein **parametrisierter Anweisungsblock** mit einem **Namen**
- Der Anweisungsblock heißt **Rumpf** (body) des Unterprogramms
  - ◆ Besteht aus einem Block, der die Berechnungsvorschrift als Anweisungssequenz enthält
- Vor dem Rumpf steht der **Kopf** (head, header)
  - ◆ Enthält **Namen** des **Unterprogramms** sowie **Namen** und **Typen** der **Parameter** und den etwaigen **Ergebnistyp**
  - ◆ Die im **Kopf** **deklarierten Parameter** heißen genauer **formale Parameter** und gelten im Rumpf als **lokale Variablen**

```
Rückgabetyf Funktionsname(Parameterliste)
{
    Anweisungen
}
```

# Unterprogramme

- Der **Block** kann mit passenden **aktuellen Parametern** über **seinen Namen aufgerufen** (call) werden
  - ◆ Dann werden an der Stelle des Aufrufs die Anweisungen des Unterprogramms ausgeführt
  - ◆ Dabei haben die formalen Parameter jetzt Werte, die sich aus den aktuellen Parametern ergeben haben
  - ◆ Am Ende wird der Aufruf durch das berechnete Ergebnis ersetzt
  - ◆ Anschließend fährt die Berechnung mit der nächsten Anweisung nach dem Aufruf fort

# Unterprogramme, Prozeduren und Funktionen

- Wir betrachten ein **C-Unterprogramm** zur Berechnung der ersten Glieder der harmonischen Reihe

$$\text{harm}(k) = \sum_{i=1}^k \frac{1}{i}$$

```
double    // Typ des Ergebnisses
harm      // Name des Unterprogramms
(int k)   // Eingabeparameter
{ // Initialisierung
    int i = 1;           // Hilfsvariable
    double res = 0.0;    // Resultatsvariable
    // Bearbeitung
    while(i <= k)
    { res += 1.0/i;
      // Take the double 1.0 and
      // not the int 1! Otherwise,
      // you will just add 0, if i>1!
      i++;
    }
    // Ergebnis
    return res;
}
```

# Unterprogramme

- Das **allgemeine** (über C hinausgehende) **Konzept des Unterprogramms** erlaubt zudem beliebig viele Rückgabewerte (speziell auch keinen), sowie **verschiedene Mechanismen zur Übergabe von Parametern und Rückgabewerten**
- Innerhalb eines Unterprogramms können wiederum weitere Unterprogrammaufrufe stehen
  - ◆ Wird innerhalb von **P** wiederum **P** selbst aufgerufen, so spricht man von einem **rekursiven Aufruf** (recursive call)
  - ◆ Ruft **P** ein **Q** und **Q** wiederum ein **P** auf, so spricht man von einem **verschränkt rekursiven Aufruf** (mutually recursive call)

# Unterprogramme

- Wie in jedem Block können auch in Unterprogrammen **lokale Variablen** (local variables) zur Speicherung von Zwischenergebnissen vereinbart werden
  - ◆ Die **formalen Parameter** (formal parameters) sind ebenfalls **lokale Variablen** im Rumpf
    - ⇒ Sie erhalten bei jedem Aufruf neue Werte, die sich aus den **aktuellen Parametern** (actual parameters) ergeben
    - ⇒ Die genaue Art und Weise, wie dies geschieht, ist durch den jeweiligen Mechanismus der **Parameterübergabe** (parameter passing) bestimmt
      - Hier stehen Werte-, Referenz- und Namensübergabe zur Auswahl, die wir später noch genauer untersuchen
- Die aktuellen Parameter können i. a. durch Ausdrücke gegeben sein
  - ◆ Wie  $2*a$  oder  $\sin(\sin(a+x))$

# Unterprogramme, Prozeduren und Funktionen

- **Beispiel (Drucken eines Bitmusters):** Wir können die Programmstücke zum Drucken von Bitmustern aus dem Beispiel weiter oben verpacken, indem wir
  - ◆ einen geeigneten Kopf hinzufügen
  - ◆ und die zu konvertierende Zahl als Parameter übergeben, statt sie von der Kommandozeile einzulesen
    - ⇒ **Bem.:** Wir tun dies hier am Beispiel einer extrem kompakten Programm-Variante, die auf das Drucken der Punkt-Separatoren verzichtet
    - ⇒ In C zeigt das Schlüsselwort **void** an, dass das Unterprogramm kein Ergebnis berechnet

```
#include <stdio.h>

void printBitPattern(int z) {

    unsigned int mask;
    for( mask = 01 << 31; mask != 0; mask >>= 1) {
        printf(((z & mask) != 0) ? "1" : "0");
    }
    printf("\n");
}

int main() {

    int main_z;
    scanf("%i", &main_z);

    printBitPattern(main_z);

    return 0;
}
```



# Unterprogramme und Funktionen

- Berechnet das **Unterprogramm** einen **Rückgabe-Wert** (return value), so sprechen wir von einer (programmiersprachlichen) **Funktion** (function)
  - ◆ Der Rückgabewert wird durch einen Ausdruck festgelegt, der das Unterprogramm beendet
    - ⇒ Zum Beispiel: **return expr;**
  - ◆ Der Wert von **expr** ist dann der Wert des Funktionsaufrufs
- Auf diese Weise können mathematische Funktionen realisiert werden und **Funktionsaufrufe** können sinnvoll in **Ausdrücken** eines Aufrufs **vorkommen**

# Unterprogramme und Funktionen

- Eine Funktion wird **aufgerufen** (call), indem man den Namen, gefolgt von einer Liste aktueller Parameterwerte, angibt
  - ◆ z.B. **f(x, 1, a\*b)** oder **sin(PI/2)**
- Die Berechnung tritt dann in die Funktion ein, deren formale Parameter die beim Aufruf angegebenen aktuellen Parameter als Werte bekommen
- Wie immer in einem Block werden die **lokalen Variablen** (**also auch die Parameter**) in einem **zugehörigen Rahmen** auf dem **Laufzeitstapel** gespeichert

# Unterprogramme und Funktionen: Beispiele

- In der Zuweisung `double y = sin(x) + sin(2*x)` kommen 2 Funktionsaufrufe als Teilausdrücke vor
  - ◆ Der Wert von `sin(x)` ist der Rückgabewert des mit `sin` bezeichneten Unterprogramms
    - ⇒ Wenn es mit (dem Wert von) `x` als aktuellem Parameter versorgt wird
  - ◆ Der durch `sin` bezeichnete Block wird zweimal ausgeführt, einmal mit `x` als aktuellem Parameter und einmal mit `2*x` als aktuellem Parameter

# Eingabeparameter, Ausgabeparameter, Transiente Parameter

- Reinen Funktionen werden als aktuelle Parameter nur völlig ausgewertete Ausdrücke übergeben
- Formale Parameter, deren Zweck nur in der Übernahme von aktuellen Werten besteht, heißen **(reine) Eingabeparameter** (input parameter)
- Dagegen erlauben es **Ausgabeparameter** (output parameter), die (außerhalb gespeicherten) **Werte** von (Ausgabe-) **Variablen** zu **verändern**, deren **Referenzen** als **aktuelle Parameter** übergeben werden
- **Transiente Parameter** (transient parameter) dienen als Eingabe- und Ausgabeparameter

- Unterprogramme, die keinen Wert zurückliefern, sondern Ausgabevariablen verändern, heißen auch **Prozeduren** (procedure)
- Eine Funktion, die gleichzeitig Ausgabeparameter benutzt, heißt **Funktion mit Seiteneffekt** (side effect)
- Das allgemeine Unterprogramm repräsentiert die Berechnung eines Bündels von mathematischen Funktionen, die die Eingabewerte jeweils auf einen der Ausgabewerte abbilden

# Signatur eines Unterprogramms

- Name, sowie Anzahl und Reihenfolge der Parameter und ihrer Typen und evtl. der Typ des Resultates bilden die Signatur (signature) des Unterprogrammes
  - ◆ Wir sprechen auch von der Aufrufschnittstelle (call interface)
- Die Signatur gibt die Syntax des Aufrufs wieder
  - ◆ Sie wird im Kopf des Unterprogrammes festgelegt
- Im allgemeinen können Unterprogramme anhand ihrer Signatur unterschieden werden
  - ◆ Auch wenn sie gleiche Namen haben
  - ◆ Wir sprechen dann vom Überladen (overloading) des Namens
  - ◆ In C nicht möglich, wohl aber in C++ und Java

# Mehrere Funktionen

```
kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos.exe functionChaos.c

kruegerb@shannon ~/c-vorkurs
$ ./functionChaos.exe
Result = 10.000000
kruegerb@shannon ~/c-vorkurs
$
```

**Bis hierher ist alles  
gut!**

**function1 ist innerhalb  
function2 bereits  
bekannt!**

```
#include <stdio.h>

double function1(double la, double lb){
    return la+2.*lb;
}

double function2(double la, double lb){
    return la+function1(la,lb);
}

int main(){

    double a=3., b = 2.;

    a = function2(a,b);

    printf("Result = %f",a);

    return 0;
}
```



# Mehrere Funktionen

```
kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos.exe functionChaos.c

kruegerb@shannon ~/c-vorkurs
$ ./functionChaos.exe
Result = 10.000000

kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos2.exe functionChaos2.c
functionChaos2.c: In function 'function2':
functionChaos2.c:4: Warnung: implizite Deklaration der Funktion »function1«
functionChaos2.c: At top level:
functionChaos2.c:7: Fehler: in Konflikt stehende Typen für »function1«
functionChaos2.c:4: Fehler: vorherige implizite Deklaration von »function1« war hier
```

Hier gibt es Probleme:

Beim Aufruf von function1 ist diese noch nicht bekannt!

Solche Fälle lassen sich in größeren Projekten leider nicht vermeiden.

```
D:\cygwin\home\kruegerb\c-vorkurs\functionChaos2.c - Notepad++
Datei Bearbeiten Suchen Ansicht Kodierung Sprachen Einstellungen Makro Ausführen TextFX Erweiterungen Fenster ?
a02-schleifen-abbruchkriterium-fakultät.css(x) bd a02-schleifen.css(x) step2.bd twodec.c functionChaos2.c

1  #include <stdio.h>
2
3  double function2(double la, double lb){
4      return la+function1(la,lb);
5  }
6
7  double function1(double la, double lb){
8      return la+2.*lb;
9  }
10
11 int main(){
12
13     double a=3., b = 2.;
14
15     a = function2(a,b);
16
17     printf("Result = %f",a);
18
19     return 0;
20 }
21

C source file length: 273 lines: 21 Ln: 10 Col: 1 Sel: 0 Dos\Windows ANSI INS
```



# Lösung: Prototypen

- Auch bei Funktionen wird, genau wie bei Variablen, unterschieden zwischen:

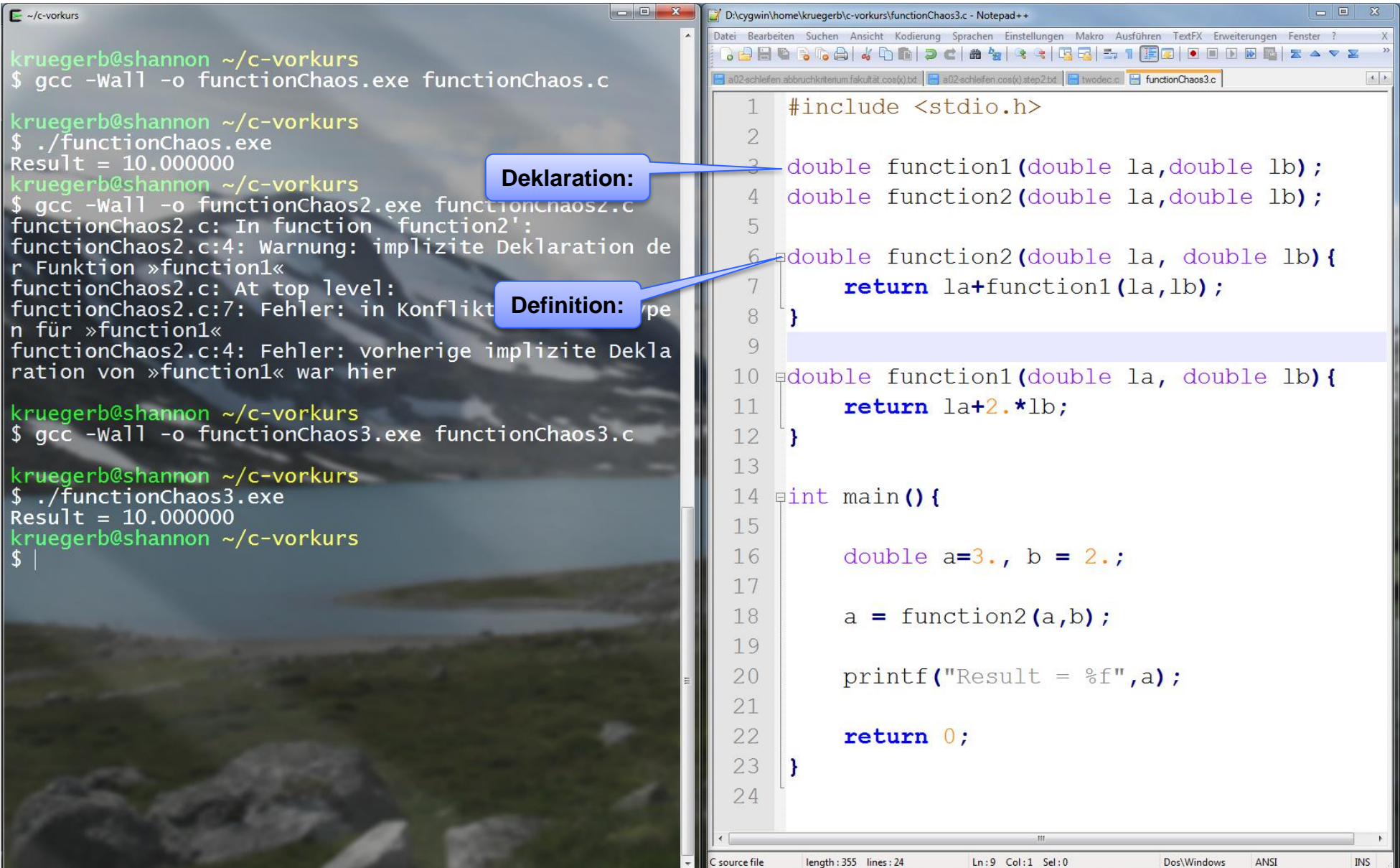
- ◆ Deklaration

- ⇒ Funktionskopf gefolgt von Semikolon:
- ⇒ **typ** **name** (**parameterliste**) ;

- ◆ Definition:

- ⇒ Enthält die Implementierung der Funktion:
- ⇒ **typ** **name** (**parameterliste**)  
    {  
        **Anweisungen**  
    }

# Prototypen im Beispiel



The image shows a terminal window on the left and a code editor on the right. The terminal displays the compilation and execution of three C programs. The first two programs, `functionChaos.c` and `functionChaos2.c`, both output `Result = 10.000000`. The third program, `functionChaos3.c`, also outputs `Result = 10.000000`. The code editor shows the source code for `functionChaos3.c`, which includes a header file `<stdio.h>` and defines two functions: `function1` and `function2`. The `main` function calls `function2` with initial values `a=3.` and `b=2.` and prints the result.

```
kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos.exe functionChaos.c

kruegerb@shannon ~/c-vorkurs
$ ./functionChaos.exe
Result = 10.000000

kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos2.exe functionChaos2.c
functionChaos2.c: In function 'function2':
functionChaos2.c:4: Warnung: implizite Deklaration der Funktion »function1«
functionChaos2.c: At top level:
functionChaos2.c:7: Fehler: in Konflikt mit Prototyp für »function1«
functionChaos2.c:4: Fehler: vorherige implizite Deklaration von »function1« war hier

kruegerb@shannon ~/c-vorkurs
$ gcc -Wall -o functionChaos3.exe functionChaos3.c

kruegerb@shannon ~/c-vorkurs
$ ./functionChaos3.exe
Result = 10.000000

kruegerb@shannon ~/c-vorkurs
$
```

```
#include <stdio.h>

double function1(double la, double lb);
double function2(double la, double lb);

double function2(double la, double lb){
    return la+function1(la,lb);
}

double function1(double la, double lb){
    return la+2.*lb;
}

int main(){
    double a=3., b = 2.;

    a = function2(a,b);

    printf("Result = %f",a);

    return 0;
}
```

C source file   length: 355   lines: 24   Ln: 9   Col: 1   Sel: 0   Dos\Windows   ANSI   INS

# Parameterübergabe

---

- Es gibt mehrere **Verfahren** für **Substitution** der **aktuellen Parameter** für die **formalen Parameter**
  - ◆ **Werteaufruf** (call by value)
  - ◆ **Referenzaufruf** (call by reference)
  - ◆ **Namensaufruf** (call by name)

- **Werteaufruf** (call by value)
  - ◆ Aktueller Parameter **ai** wird ausgewertet
  - ◆ **Wert** wird **formalem Parameter ei** zugewiesen
    - ⇒ Entspricht einer **Zuweisung ei=ai**
    - ⇒ Grundlegender Mechanismus bei vielen Programmiersprachen
      - **Java, C, C++**
  - ◆ Das Unterprogramm kann also als Seiteneffekt **nicht** den aktuellen Parameter ändern

- Referenzaufruf (call by reference)

- ◆ Aktueller Parameter ist Variable (im Unterprogramm)

- ⇒ Zulässig jeder Ausdruck, der einen Linkswert hat
  - Z.B. also auch indizierte Variable wie `a[i+5]`
- ⇒ Linkswert wird für den Linkswert des aktuellen Parameters eingesetzt
- ⇒ Formaler Parameter wird ein **alias** für den **aktuellen Parameter**
  - Wert des formalen Parameters wird also nicht kopiert, sondern formaler Parameter verweist auf Wert

- ◆ Zuweisungen an den formalen Parameter ändern also auch Wert des aktuellen Parameters

- ◆ In **Java nicht direkt verwendet**

- ⇒ Für Objekte wird eine eingeschränkte Variante verwendet
  - Siehe unten

## ● Referenzaufruf (Forts.)

- ◆ Unterstützt in C++ oder Pascal („var“-Parameter)
- ◆ Bei Referenzaufruf kann Kopieren großer Strukturen vermieden werden
  - ⇒ Etwa großer Arrays
  - ⇒ In C++ kann syntaktisch gekennzeichnet werden, ob aktueller Parameter im Unterprogramm geändert werden kann oder nicht
- ◆ Kann in C simuliert werden
  - ⇒ Verwende Zeigervariable
    - Call-by-value der Zeigervariable **px**, die auf die eigentliche Variable **x** verweist, simuliert Referenzaufruf von **x**

- Namensaufruf (call by name)
  - ◆ Aktueller Parameter ist beliebiger Ausdruck
    - ⇒ Wird ohne Auswertung für den formalen Parameter substituiert
  - ◆ Ausdruck wird im Kontext des Unterprogramms immer dort evaluiert, wo der formale Parameter vorkommt
    - ⇒ Verwendung im  $\lambda$ -Kalkül
      - Und funktionalen Sprachen wie etwa LISP
      - Auch in ALGOL 68 konnte Namensaufruf verwendet werden
- Wird von Java, C++, Pascal etc. nicht unterstützt



# Parameterübergabe

- **Beispiel:** Effekte der Übergabemechanismen auf die Reihung **a** beim Prozedur-Aufruf **p(a[i])**

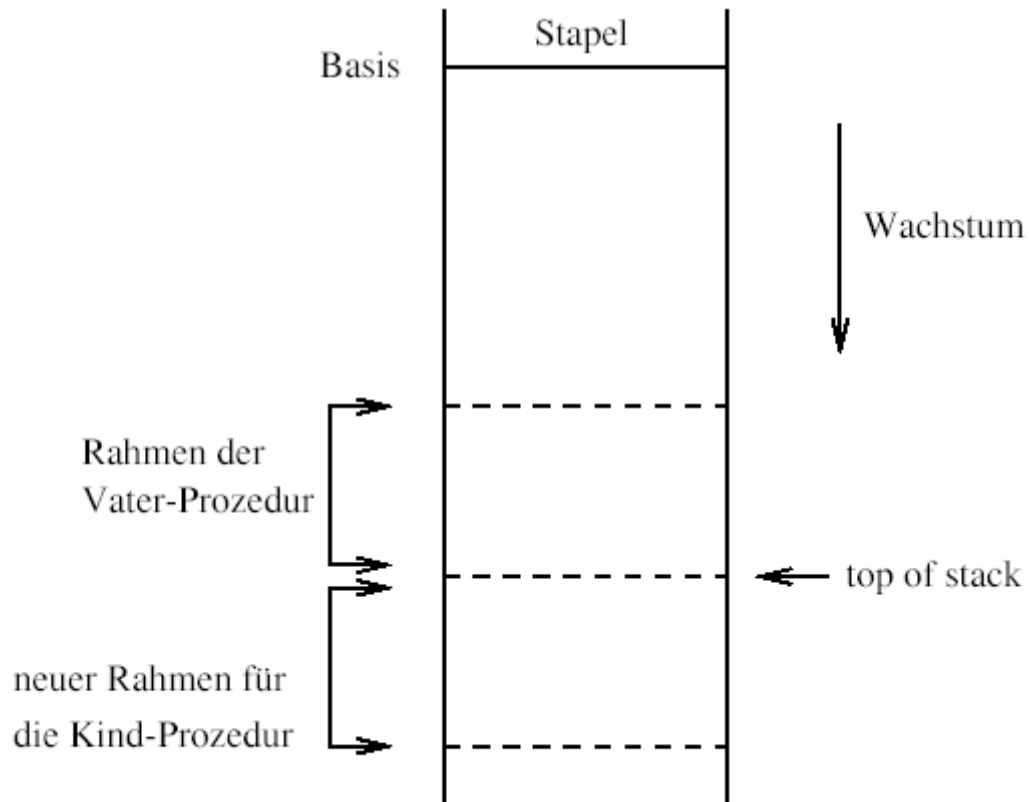
```
class Program {  
    static int i=0;  
    static int[] a=new int[] {10, 20};  
  
    static void p(int x) {  
        i=i+1; x=x+2;  
    }  
  
    public static void main(String[] argv) {  
        p(a[i]);  
        System.out.println(a[0]);  
        System.out.println(a[1]);  
    }  
}
```

**Werte-Aufruf:** Zu Beginn von p wird implizit die Zuweisung **x = 10** ausgeführt. Dann wird das Feld **i** in **Program** inkrementiert und die **lokale Variable x** in **p** um 2 erhöht, was auf **a[0]** **keine Auswirkungen** hat. Es wird **10** und **20** ausgegeben.

**Referenz-Aufruf** (in **Java nicht** möglich): Nun ist **x** ein **alias** für **a[0]**, wir schreiben **x** für **a[0]**. Es wird in **p** also **a[0] = a[0] + 2**; ausgeführt und **12** und **20** ausgegeben.

# Parameterübergabe und Laufzeitstapel

- Speicherbild beim Ablauf einer einfachen Funktion in **Java**, **C/C++**



## Benötigte Verwaltungsgrößen

**Statischer Verweis** (static link) auf den Rahmen, in dem globale Variablen gespeichert sind

**Dynamischer Verweis** (dynamic link) auf den Rahmen der Prozedur-Inkarnation, aus der der gegenwärtige Prozeduraufruf getätigt wurde. Das ist immer der Rahmen direkt unterhalb auf dem Stapel.

Die **Rücksprungadresse** (return address) spezifiziert die Anweisung, mit der die Berechnung nach Beendigung des Unterprogramm-Aufrufs fortfährt.

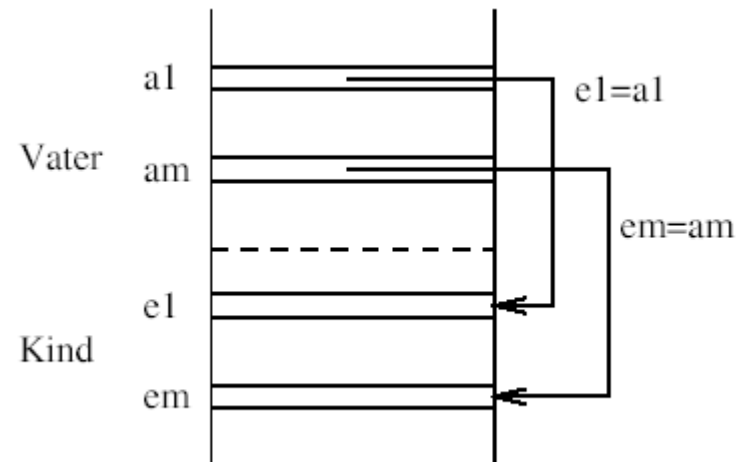
# Parameterübergabe und Laufzeitstapel: Wertaufwurf

Der Einfachheit halber seien die aktuellen Parameter Variablen im Block des Vaters. Die formalen Parameter sind lokale Variablen im Block des Sohnes. Bei der **Werteübergabe** werden die Werte der **aktuellen Parameter** (Ausdrücke oder Variablen) vom Vater zum Sohn **kopiert**. Dies geschieht in einem vom Übersetzer automatisch erzeugten Zuweisungsblock zu Beginn der Sohn-Prozedur.

Te1 e1 = a1;

...

Tem em = am;

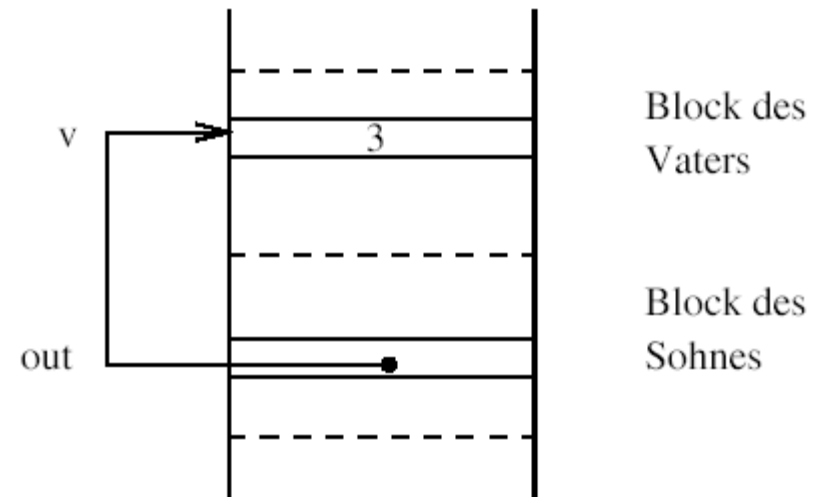


# Parameterübergabe und Laufzeitstapel: Wertaufufruf

- Nach Beendigung des Unterprogramms werden die lokalen Variablen aufgelöst und der Speicherrahmen freigegeben
- Die Werte der formalen Parameter werden **nicht** an die aktuellen Parameter zurück überwiesen, falls Werteübergabe vereinbart war
  - ◆ Dieser Fall gilt grundsätzlich für C und Java
- Diese Übergabetechnik kann im Prinzip aber ebenso in umgekehrter Richtung auf Ausgabeparameter angewendet werden
  - ◆ Im früher für FORTRAN verwendeten Mechanismus der **Werte- und Resultatsübergabe** (call by value and result) werden beim Austritt aus der Prozedur die Werte der formalen Ausgabeparameter an die aktuellen Ausgabeparameter zugewiesen

# Parameterübergabe und Laufzeitstapel: Referenzaufruf

- Grundidee der **Referenzübergabe** ist es, nicht den Wert selbst zum Kind zu kopieren, sondern nur eine **Referenz auf das Objekt** zu übergeben, die immer gleich groß ist (z. B. ein 32-Bit-Zeiger)
- Sei **out** ein formaler und **v** ein aktueller Ausgabeparameter.



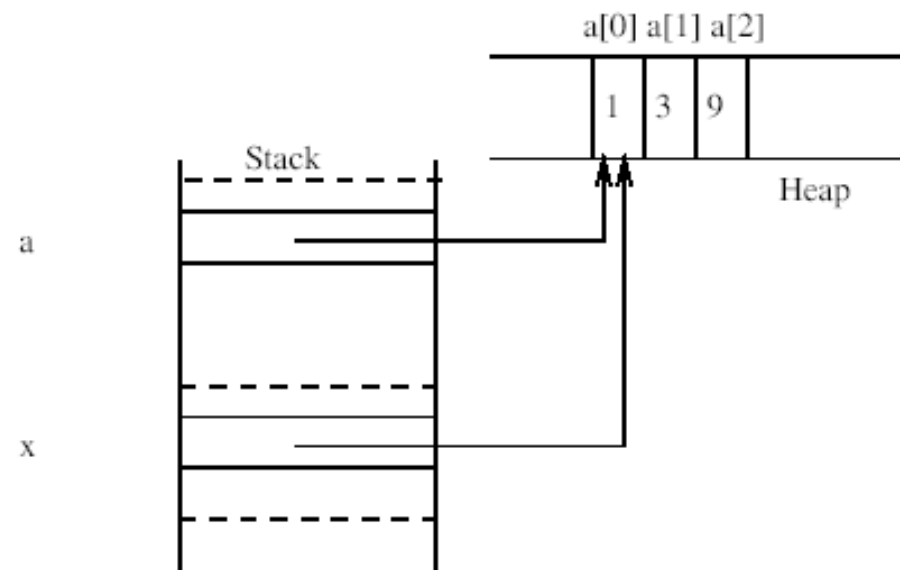
- **Bemerkung:** In C, wo es keine Referenzübergabe gibt, kann man den Effekt dieses Mechanismus dadurch erhalten, dass man in der Kind-Prozedur **Zeiger-Variablen** benutzt und die **zusätzliche Indirektion** beim Zugriff **explizit ausprogrammiert**
  - ◆ In C++ existieren dagegen allgemeine Referenzvariablen

# Parameterübergabe und Laufzeitstapel

## ● Beispiel: Werteübergabe bei Referenzvariablen

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```

Speicherbild der eingeschränkten Referenzübergabe in Java am Punkt 1. Das Speicherbild sieht nach der Übergabe eines Parameters vom Klassentyp `int[]` wie folgt aus:



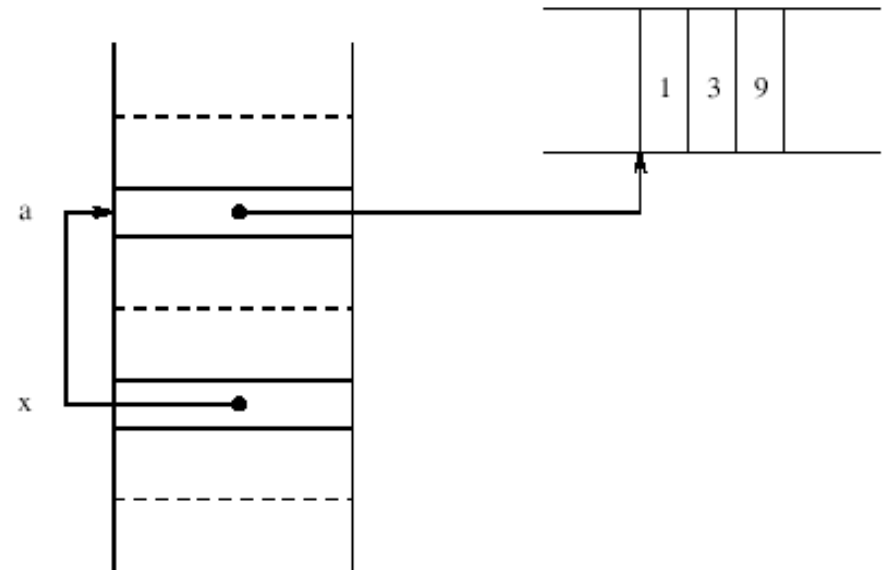
# Parameterübergabe und Laufzeitstapel

## ● Beispiel bei Referenzübergabe

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```

Speicherbild bei uneingeschränkter Referenzübergabe des Parameters **x** am **Punkt 1** (in Java nicht möglich).

Wir hätten eine weitere Indirektion erhalten:



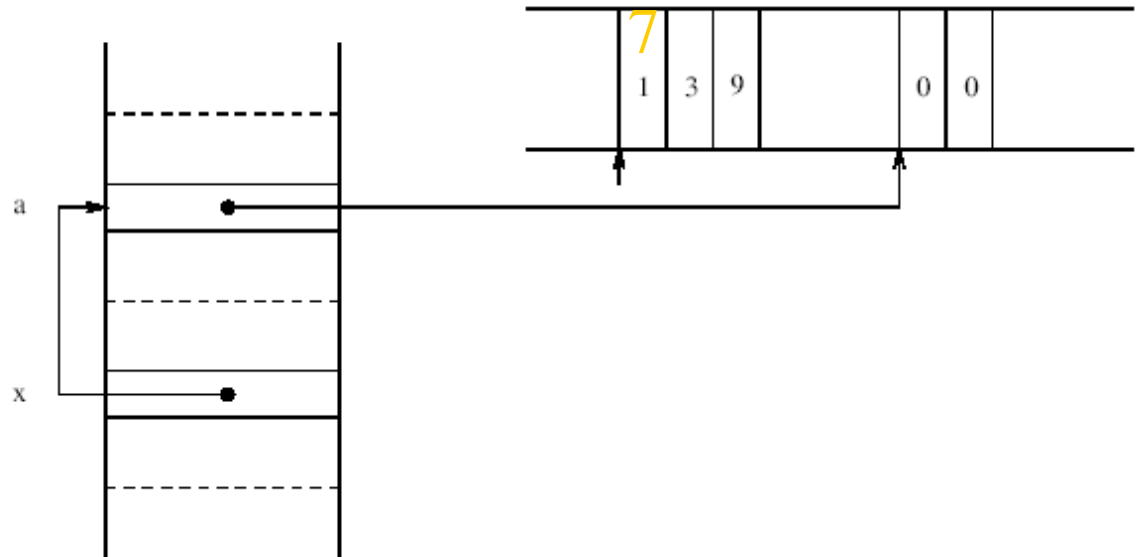


# Parameterübergabe und Laufzeitstapel

## ● Beispiel bei Referenzübergabe

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```

Speicherbild bei uneingeschränkter Referenzübergabe des Parameters **x** am **Punkt 3** (in Java nicht möglich).  
Änderung des Wertes auch beim Vater

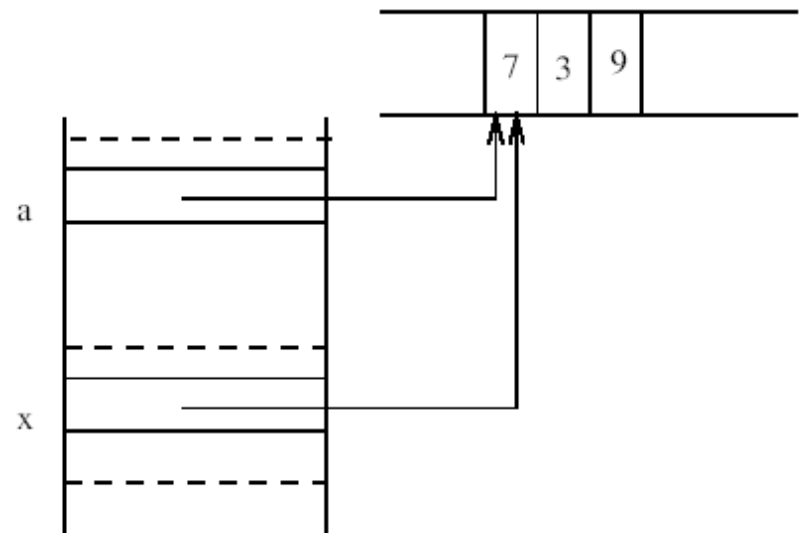


# Parameterübergabe und Laufzeitstapel

## ● Beispiel (Forts.): Werteübergabe bei Referenzvariablen

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```

Speicherbild der **eingeschränkten Referenzübergabe** in Java am **Punkt 2**. Die Zuweisung an ein Element ändert den Wert des Arrays auch beim Vater.

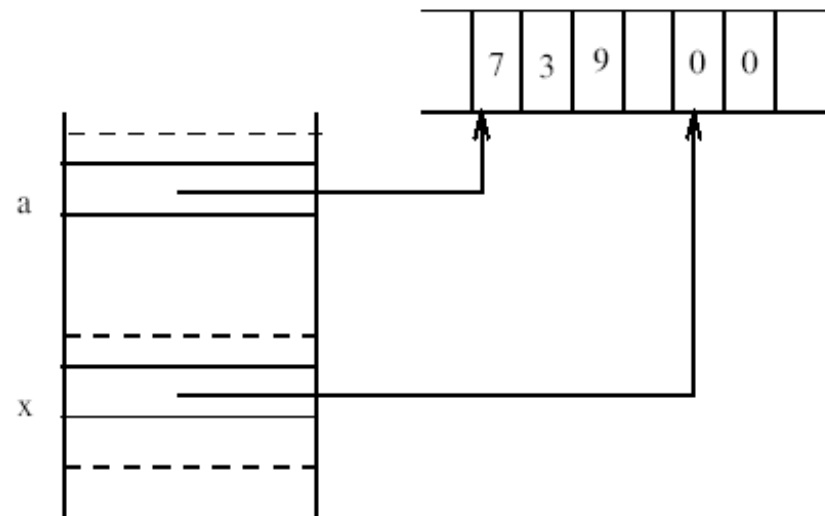


# Parameterübergabe und Laufzeitstapel

## ● Beispiel (Forts.): Werteübergabe bei Referenzvariablen

Speicherbild der eingeschränkten Referenzübergabe in Java am Punkt 3. Das **new** ändert nur die Referenz im Sohn.

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```

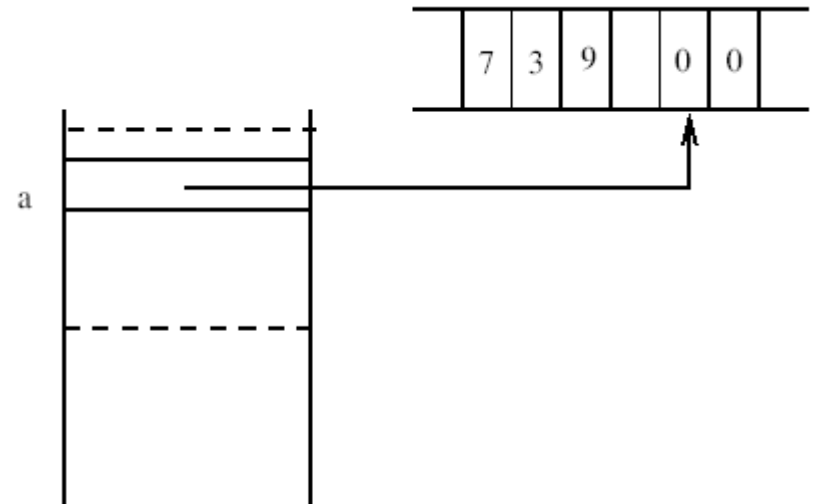


# Parameterübergabe und Laufzeitstapel

## ● Beispiel (Forts.): Werteübergabe bei Referenzvariablen

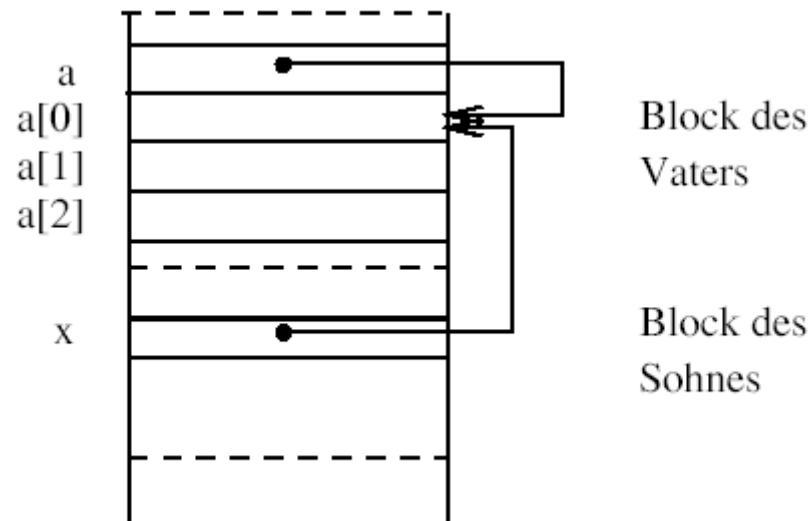
Speicherbild der eingeschränkten Referenzübergabe in Java am Punkt 4 aufgrund der Zuweisung.

```
void father() {  
    int[] a = new int[3];  
    a[0]=1; a[1]=3; a[2]=9;  
    a=son(a);  
    // Punkt 4  
}  
  
int[] son(int[] x) { // Punkt 1  
    x[0]=7; // Punkt 2  
    x = new int[2]; // Punkt 3  
    return x;  
}
```



# Parameterübergabe und Laufzeitstapel

- Ein mögliches Speicherbild in C
  - ◆ Reihungen oder andere komplexe Objekte können auch auf dem Stack angelegt werden
    - ⇒ Sehr gefährlich (etwa dangling pointers)



# Spezifikation von Unterprogrammen

---

# Spezifikation von Unterprogrammen

## ● Grobschema der Spezifikation

- ◆ „Dokumentationskommentare“ von Java inzwischen auch für andere Programmiersprachen gebräuchlich

⇒ Tool-Unterstützung

– Zum Beispiel doxygen [www.doxygen.org/](http://www.doxygen.org/)

```
/** Name und Kurzbeschreibung von f(p1, ..., pk).  
 * Anforderungsteil:  
 * - Anforderung an die pi  
 * - Anforderungen an den Datenteil der Klasse  
 * Zusicherungsteil:  
 * - Zusicherung an das Ergebnis res  
 * - Zusicherungen an geworfene Ausnahmen  
 * - Zusicherungen an den Datenteil der Klasse  
 * - Zusicherungen an veränderte Parameter-Objekte  
 * Weitere Kommentare  
 */  
T f(T1 p1, ..., Tk pk)
```

# Spezifikation von Unterprogrammen

## ● Dokumentationskommentare

### ◆ javadoc-Kommentare

```
/** Exponentiationsfunktion power.  
 * Achtung: kein besonderer Test auf Ergebnisüberlauf.  
 * @see Math.pow  
 * @param a Basis, --  
 * @param b Exponent, b >= 0  
 * @return a^b.  
 * @author Wolfgang Küchlin  
 * @author Andreas Weber  
 * @version 2.1, Jun 1998  
 */  
int power(int a, int b)
```



# Rekursion in Unterprogrammen

- Durch Unterprogramme können bequem rekursive Algorithmen realisiert werden
  - ◆ Beispiel: Berechnung des ggT

```
/** Größter gemeinsamer Teiler.  
 * Anforderung:  
 *   a: a>=0, a>b.  
 *   b: b>=0, a>b.  
 * Zusicherung:  
 *   res: res ist die größte Zahl mit res<=a, res<=b,  
 *       a%res==0, b%res==0.  
 */  
int ggT(int a, int b) {  
    // Trivialfall  
    if (b==0) return a;  
    // Reduktion und Rekursion  
    return ggT(b, a%b);  
}
```

# Rekursion in Unterprogrammen

- Beweis der partiellen Korrektheit des Beispiels ggT
  - ◆ Die Reduktion ist korrekt: Wenn *res* sowohl *b* als auch *a%b* teilt, dann teilt es auch *a* wegen
$$\Rightarrow a = (a/b)b + a\%b \quad (*)$$
  - ◆ Damit ist *res* ein gemeinsamer Teiler von *a* und *b*
  - ◆ Dass *res* sogar der *größte* gemeinsame Teiler von *a* und *b* ist, sieht man wie folgt:
    - ⇒ Sei *r* irgendein gemeinsamer Teiler von *a* und *b*
    - ⇒ Dann ist *r* wegen (\*) auch ein gemeinsamer Teiler von *a%b*
      - Da *a%b = a - (a/b)b* und *r* beide Summanden teilt
    - ⇒ Somit gilt *r* teilt *res*

# Beispiel für rekursive Funktion: Ackermann-Funktion

- Ackermann-Funktion  
 $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  ist wie folgt definiert

$$A(x, y) = \begin{cases} y + 1 & \text{falls } x = 0 \\ A(x - 1, 1) & \text{falls } y = 0 \\ A(x - 1, A(x, y - 1)) & \text{sonst} \end{cases}$$

```
/** Ackermann-Funktion.  
 * @param a: a >= 0  
 * @param b: b >= 0  
 * @return res: res == Ackermann(a,b)  
 */
```

```
int A(int a, int b) {  
    int res;  
    if( a == 0 ) res = b+1;  
    else  
        if( b == 0 ) res = A(a-1, 1); // Aufruf 1  
        else res = A(a-1,           // Aufruf 2  
                    A(a,b-1));      // Aufruf 3  
    return(res);  
}
```

## Beispiel für rekursive Funktion: Ackermann-Funktion

- Die Ackermann-Funktion terminiert für alle natürlichen Zahlen als Eingabewerte
  - ◆ Formaler Beweis nicht sehr schwierig
    - ⇒ Es findet immer eine Reduktion statt
- Ackermann-Funktion wächst aber extrem schnell
  - ◆ Schneller als Exponentialfunktion, „Türme von Exponenten“, ...
    - ⇒ Formal: Schneller als jeder „primitiv rekursive Funktion“
    - ⇒ Schon  $\text{ack}(4,2)$  ist von den meisten Rechnern nicht mehr zu bewältigen, das Resultat hat 19727 Stellen!
    - ⇒  $\text{ack}(4,3)$  benötigt bereits mehr Stellen, als es Atome im Universum gibt.

# Beispiel für rekursive Funktion: Ackermann-Funktion

- Ackermanns eigene Definition dieser Funktion war sehr umständlich.  
1955 wurde die eben vorgestellte Version von Rosza Peter gefunden.
- Peter definierte auch eine in der Theoretischen Informatik sehr wichtige Klasse rekursiver Funktionen, die sog. **primitiv-rekursiven** Funktionen, die sich hinsichtlich der Berechnungseffizienz „gutartig“ verhalten.
- Die Ackermann-Funktion ist nicht mehr primitiv-rekursiv

Aber schon einfachste Aufrufe von *ack* demonstrieren eindrucksvoll die enorme „Kraft der Rekursion“:

$$\text{ack}(2,1) = \text{ack}(1, \text{ack}(2,0)) = \dots$$

**Nebenrechnung zur Auswertung des Wertparameters n vor dem Eintritt in ack:**

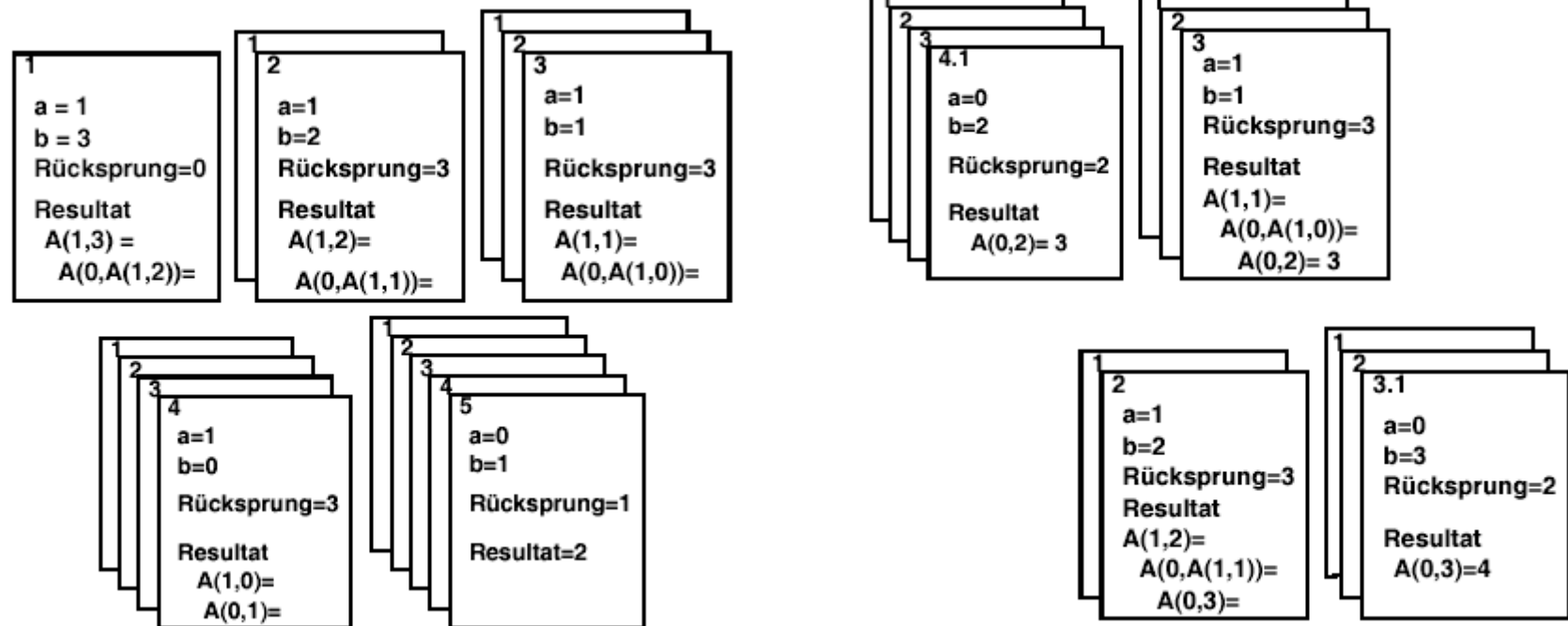
$$\text{ack}(2,0) = \text{ack}(1,1) = \text{ack}(0, \text{ack}(1,0)) = \dots$$

$$\text{neue Nebenrechnung: } \text{ack}(1,0) = \text{ack}(0,1) = 2$$

$$\dots = \text{ack}(0,2) = 3$$

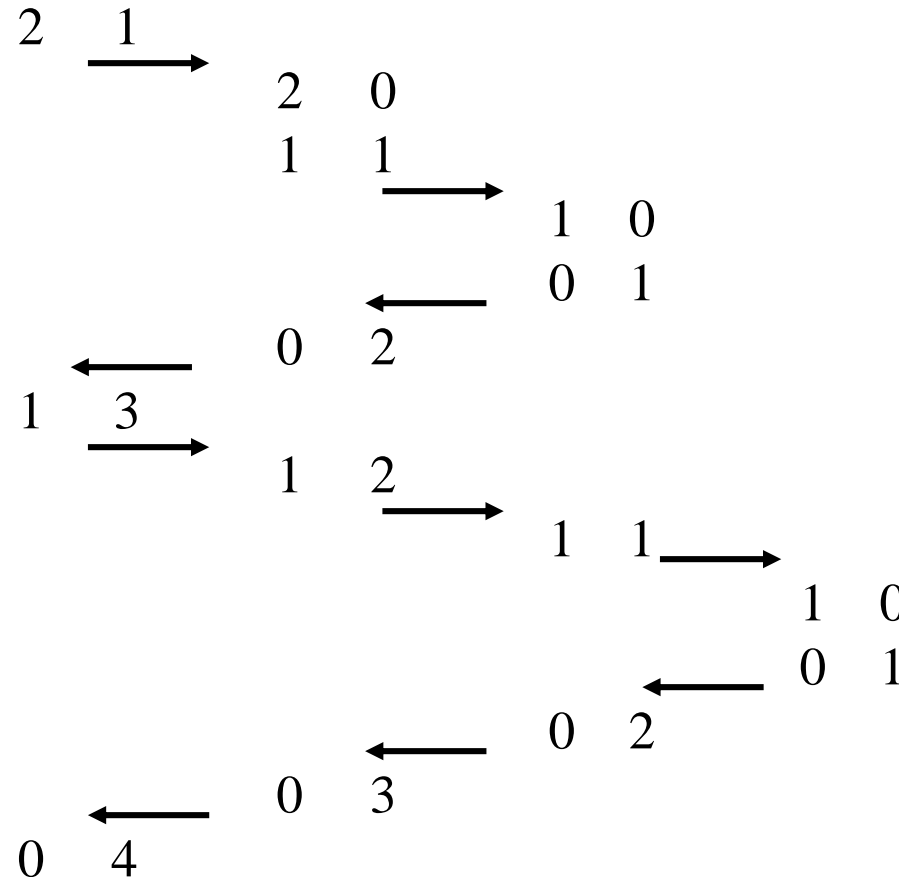
# Beispiel für rekursive Funktion: Ackermann-Funktion

- Wir verfolgen den Verlauf der Berechnung bei der Eingabe  $a = 1$  und  $b = 3$ 
  - Mit der fiktiven Rücksprungadresse 0 zeigen wir das Ende der Berechnung an



# Beispiel für rekursive Funktion: Ackermann-Funktion

- Neben dem immens schnellen Wachstum der Funktionswerte beobachtet man noch einen **anderen Wachstumsmechanismus**: das Anwachsen der Zahl von Nebenrechnungen bei Parameterübergabe und von deren Tiefe



# Ulam's Funktion (1)



## Stanislaw Marcin Ulam

(1909 - 1984)

- polnischer Mathematiker
- lebte seit den 1930er Jahren in den USA
- beteiligt am Los Alamos-Projekt  
(Bau der ersten Atombombe)
- nutzte dort einen der ersten Computer  
(MANIAC)

Ulam und seine Frau

Während von Ackermanns Funktion zumindest noch bewiesen werden kann, dass sie (theoretisch) immer terminiert, hat das von Ulam's Funktion bis heute niemand nachweisen können !



# Ulam's Funktion (2)

```
int ulam (int n){  
    if (n==1) return 1;  
    if (!odd(n)) return ulam(n/2);  
    else return(ulam(3*n+1));  
}
```

ulam(1) = 1

ulam(2) = ulam(1) = 1

ulam(3) = ulam(10) = ulam(5) = ulam(16) = ulam(8) = ulam(4) = ulam(2) = ... = 1

...

ulam(6) = ulam(3) ... = 1

ulam(7) = ulam(22) = ulam(11) = ulam(34) = ulam(17) = ulam(52) = ulam(26) =  
= ulam(13) = ulam(40) = ulam(20) = ulam(10) = ulam(5) = ... = 1

...

Welches **n** auch immer man testet, immer kommt **1** heraus —  
aber kein Mensch weiß (bis heute), ob *ulam* überhaupt für alle **n** terminiert!

## Ulam's Funktion (3)

Dieses Beispiel zeigt das Problem bei der Ulam-Funktion: Der 3. Fall der rekursiven Definition führt immer wieder dazu, dass der Input-Parameter stark anwächst – es scheint so, als ob stets der 2. Fall „die Sache rettet“ – was außerdem zu einer großen Anzahl rekursiver Aufrufe führt :

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

