3.11 Funktionstypen

- Überblick
  - Unterprogramme (UP) ohne Parameter
  - UP mit Parametern
  - Funktionen, Prozeduren
  - Rekursive UP

Funktionstypen

Eine Funktion ist ein benanntes, unabhängiges C-Codefragment, das eine bestimmte Aufgabe ausführt und optional einen Wert an das aufrufende Programm zurückgibt.

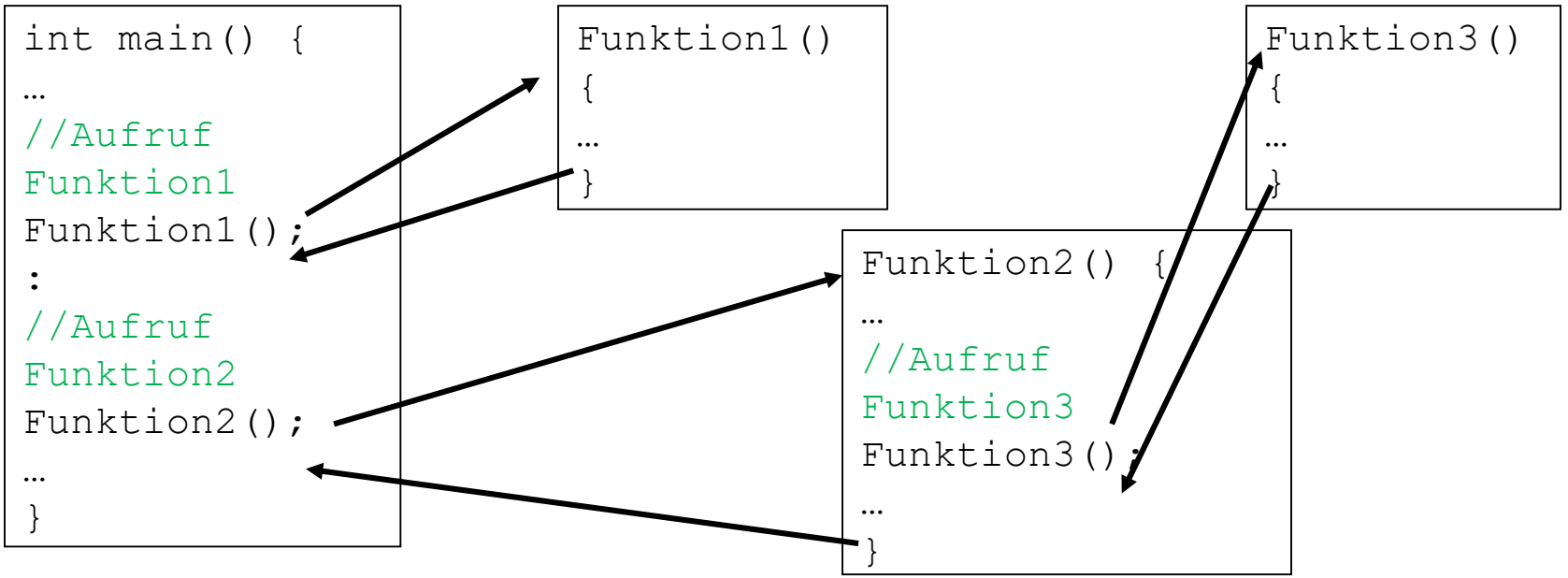
- Eine C-Funktion ist **benannt**. Jede Funktion hat einen eindeutigen Namen. Über diesen Name kann die C-Funktion später aufgerufen werden.
- Eine C-Funktion ist **unabhängig**. Eine Funktion kann ihre Aufgabe ausführen, ohne dass davon andere Teile des Programms betroffen sind oder diese Einfluss auf die Funktion nehmen.
- Eine Funktion führt eine **bestimmte Aufgabe** aus.
  Eine Aufgabe ist ein bestimmter, klar definierter Job, den das Programm im Rahmen seines Gesamtziels ausführen muss.
- Eine Funktion kann einen **Wert** an das aufrufende Programm **zurückgeben**.
  Wenn ein Programm eine Funktion aufruft, führt es die in der Funktion enthaltenen Anweisungen aus. Beim Rücksprung aus der Funktion können mit entsprechenden Anweisungen Informationen an das aufrufende Programm übermittelt werden.

Funktionstypen (parameterlos)

Syntax: Rückgabe_Typ Funktionsname ([Übergabeparameter])

```
{
  /* C-Anweisungen */
}
```

Wenn ein Programm eine Funktion aufruft, springt die Programmausführung in die Funktion und kehrt anschließend wieder zum aufrufenden Programm zurück.



void in Funktionen

Falls kein Wert zurückgegeben wird,

- sollte der Rückgabewert der Funktion void sein
- keine return-Anweisung mit einem Ausdruck benutzt werden

Beispiel:

```
void squares() {
    int loop;
    for (loop=1; loop<10; loop++)
       printf("%d \n", loop*loop);
}
int main() {
    squares();
    return EXIT_SUCCESS;
}
```

Bemerkung:
Selbst wenn keine Parameter übergeben werden, müssen Klammern nach dem Funktionsnamen folgen: ()

Lokale und globale Variablen

- Lokale Variablen
  - In Funktionen (allg.: Blöcken) können Variablen lokal deklariert werden.

```
void  A() {
  int i,j;
  float x;

    … //Anweisungen
}
```

  - Verwendung nur innerhalb der eigenen Funktion (allg.: Blocks) möglich.

- Globale Variablen
  - Deklaration außerhalb von UPs (d.h. Modulebene)

```
#include <stdio.h>

int g;
float f;

void A() {
    …
}

int main(){
  …
}
```

  - Benutzung innerhalb aller Funktionen und Prozeduren möglich.

Gültigkeitsbereich von Variablen

Gültigkeitsbereich einer Variablen:

- bezeichnet den Codeabschnitt, in dem man auf die Variable zugreifen kann.
- gilt für alle Arten von Variablen - einfache Variablen, Arrays, Strukturen, Zeiger etc. – sowie für die symbolischen Konstanten. (const)
- legt Lebensdauer einer Variablen fest, d.h., wie lange eine Variable im Speicher existiert
- Reservierung des Speicherplatzes
  - globale Variablen werden statisch angelegt (einmal für Programmablauf).
  - lokale Variablen werden für jeden Funktionsaufruf eigens angelegt.

	lokale Variablen	globale Variablen
angelegt	*dynamisch*: neu bei jedem Funktionsaufruf	*statisch*: einmalig zu Programmbeginn
freigegeben	jeweils am Ende der Funktion	am Programmende

- globale Variablen bleiben also über Prozeduraufrufe hinweg erhalten.

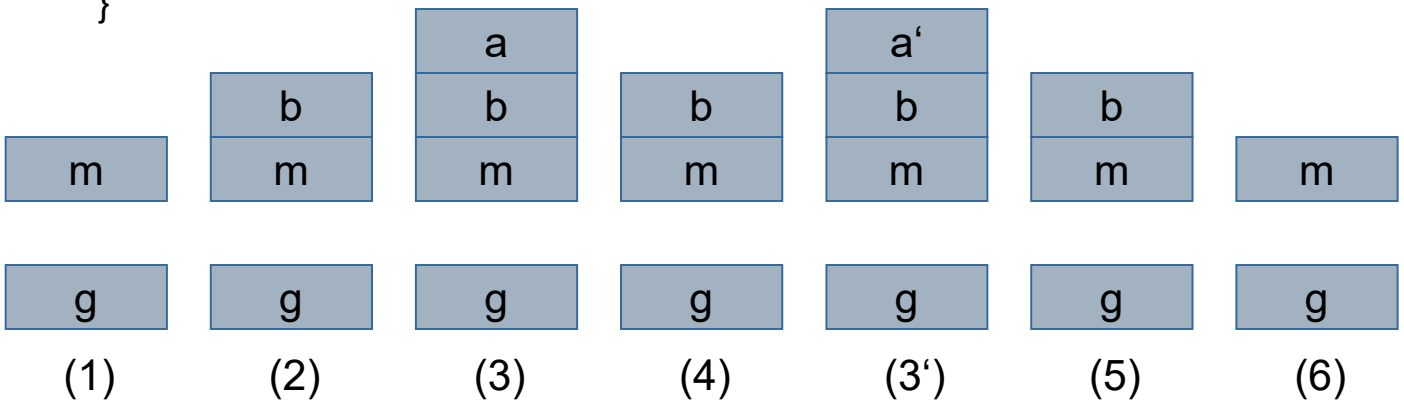
Beispiel für Speicherbelegung

```
#include <stdio.h>
    int g;   …;

    void A () {
            int a;
            … (3) …
    }
    void B () {
            int b;
            (2) … A(); … (4) … A(); … (5)
    }
```

```
void C() {
        int m;
        (1) … B(); … (6)
}

int main() {
        …; C();
}
```



lokale Variablen (Kellerspeicher, Stapel, “Stack“)

globale Variablen

Gültigkeitsbereich von Variablen

- **Sichtbarkeitsbereich** (Gültigkeitsbereich, *Scope*)
  - Programmstück, in dem auf eine Variable zugegriffen werden kann.
  - ab Deklaration bis zum Ende des Blocks, in dem Deklaration steht.
  - außerhalb dieses Blocks ist der Bezeichner nicht sichtbar.

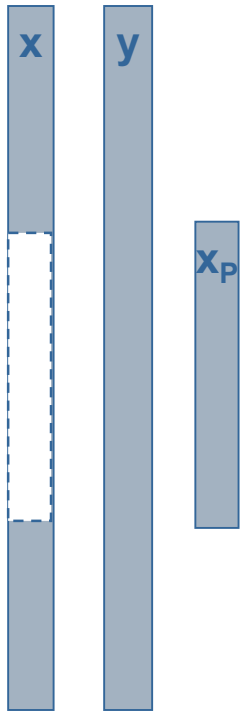
- **Verdeckung** im Beispiel rechts:
  - lokales x verdeckt in A globales x.
  - Zugriff auf x in A betrifft lokales x.
  - globales x lebt noch, ist in A aber nicht sichtbar (*verschattet*).

Beispiel

```
#include <stdio.h>
 int x,y;
 float f;

 void A() {
    int x;
    …
 }

 int main() {
     …; A();
 }
```



Gültigkeitsbereich von Variablen

Statische und automatische Variablen

- Lokale Variablen sind standardmäßig automatisch.
- automatische Variablen behalten zwischen den einzelnen Aufrufen der Funktion, in der sie definiert sind, ihre Werte nicht bei.
- Damit eine lokale Variable ihren Wert zwischen den Aufrufen behält, muss sie mit dem Schlüsselwort static als statisch definiert werden.
- Wie sieht es mit statischen globalen Variablen aus?

```
#include <stdio.h>
void funk1(void); // Forward Deklaration

int main(void){
        int count;
        for (count = 0; count < 20; count++){
                printf("In Durchlauf %d: ", count);
                funk1();}
        return 0;}

void funk1(void){
        static int x = 0;       /* statisch: Wert bleibt erhalten  */
        int y = 0;              /* automatisch: Wert immer neu */
        printf("x = %d, y = %d\n", x++, y++);}
```

Funktionen mit Parametern

Parameter (Argumente) sind Werte, die beim Aufruf an eine Funktion (bzw. Unterprogramm- UP) übergeben werden.

Deklaration (Signatur)

```
void printMax (int x, int y);
```

Verwendung (Aufruf)

```
printMax (100, 2*i);
```

Definition

```
void printMax (int x, int y) {
      if (x > y) printf(„ x: %d \n “, x);
      else printf(„y: %d \n “, y);
 }
```

- Formale Parameter
  - gehören zur UP-Deklaration, stehen im UP-Kopf.
  - sind lokale Variablen.
- Aktuelle Parameter
  - werden beim Aufruf angegeben.
  - sind Ausdrücke (Werte).
- Parameterübergabe
  - Werte aktuelle Parameter aus.
  - Weise Werte an entsprechende formale Parameter zu.
  - formale Parameter sind Kopien der aktuellen Parameter

Werteübergabe – Call by Value

- Der aktuelle Parameter ergibt einen Ergebnis**wert**, der anschließend in die Orte (Platzhalter) der formalen Parameter **kopiert** wird.

- **Einordnung :**

  Ein aktueller Werte-Parameter erlaubt – aus Sicht der aufrufenden Umgebung – nur **Lesezugriff** („read-only value“)

- **Lebensdauer** des Parameters :

  Es ist keine Zuweisung zur Kommunikation mit der Funktions-Umgebung möglich; die „Lebensdauer“ eines aktuellen Parameters ist auf die Ausführungsdauer der Prozedur beschränkt und somit nur als **interne Variable** verwendbar !

- **Probleme** durch Kopieren aktueller Parameter

  Insbesondere bei strukturierten (Daten-) Objekten, z.B. Felder*, Strukturen, kann das Anlegen von Kopien hohen Zeit- und Platzaufwand bedeuten !

* Gilt nicht für C!

Werteübergabe – Call by Reference

- Beispiel: Durchschnitt von float-Werten eines Arrays

```
float find_average(int size, float list[]) {
    float sum=0.0; int i;
    for (i=0;i<size;i++) sum+=list[i];
    return(sum/size);
}
```

- Aufruf der Funktion

```
void foo() {
    float b[]={0.3, 1.4, 2.5, 3.6, 4.7}, result;
    int s = 5;
    result=find_average(s,b);
    printf("average=%f n",result);
}
```

Werteübergabe – Call by Reference

Adressübergabe (Referenz-/Zeigerübergabe, „**call-by-reference**“)

Schema: Der aktuelle Parameter wird durch seine **Adresse** (Ort im Speicher), z.B. mittels eines Variablennamen, identifiziert. Diese Adresse wird direkt an den formalen Parameter gebunden.

Vorteile :

- **Steigerung der Effizienz** gegenüber Kopieren
- Möglichkeit der Variablennutzung zur **Rückgabe** von Prozedur-Ergebnissen

Ein für die Rückgabe von Ergebnissen genutzter formaler Parameter muss als Adress- (Referenz-) Parameter deklariert werden !

Nachteile :

- Die Veränderung des Wertes eines aktuellen Parameters – der als Referenz-Parameter deklariert wurde – führt **immer** zu **(kontrollierten) Seiteneffekten** !

Werteübergabe – Call by Reference (Beispiel)

/* Übergabe der Variablen x als Zeigerparameter p */

```
#include <stdio.h>

void increment_p(int* p)  {
     (*p) = (*p) + 1;
}

int main()  {
    int x;
    x = 3;

    increment_p(&x);
    printf("Das Ergebnis ist %d\n", x);
     return 0;
}
```

Werteübergabe – Call by Reference (Beispiel – so nicht !)

/* Übergabe der Variablen x als Zeigerparameter p */

```
#include <stdio.h>

    void increment_p(int* p)  {
     int x = *p;  //(*p) = (*p) + 1;
                  //Lokaler Parameter x wird mit
     x = x + 1;   //Beendigung der Funktion gelöscht;
     p = &x;      //Adresse in p entspricht nicht mehr
}                 //der von b in main! Änderung in
                  //increment nicht in aufrufender
int main()  {     //Umgebung sichtbar!
    int *b; int a;
    b = &a;
    *b = 8;
    increment_p(b);
    printf("Das Ergebnis ist %d\n", *b);
}
```

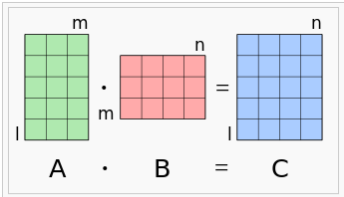
Übergabe von zweidimensionalen Arrays an Funktionen

```c
#include <stdio.h>
#include <stdlib.h>
#define ROW 5
#define COLUMN 5

void print(int arr[][COLUMN], int ROW) {
  int i, j;
  for(i = 0; i < ROW; i++) {
       for(j = 0; j < COLUMN; j++)
              printf("%d; ", feld[i][j]);
       printf("\n");
  }
  printf("\n");
}

int main(void) {
  int val[ROW][COLUMN]; int i, j;
  for(i = 0; i < ROW; i++)
       for(j = 0; j < COLUMN; j++)
              val[i][j] = i+j; print(val, ROW);
  return EXIT_SUCCESS;
}
```

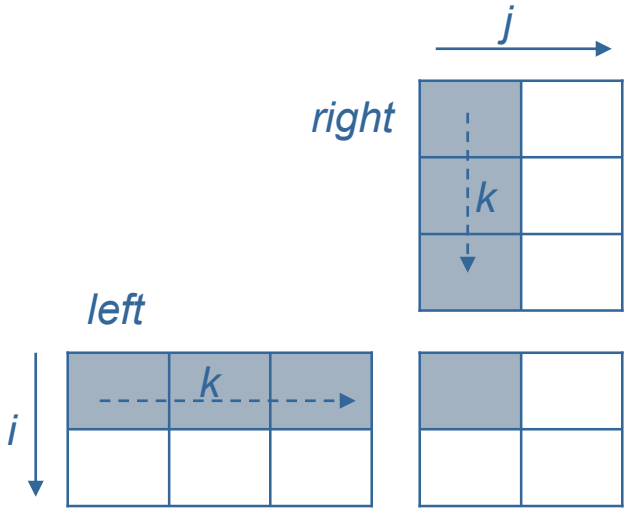
Beispiel: Matrizenmultiplikation



Bei einer Matrizenmultiplikation muss die Spaltenzahl der ersten Matrix gleich der Zeilenzahl der zweiten Matrix sein. Die Ergebnismatrix hat dann die Zeilenzahl der ersten und die Spaltenzahl der zweiten Matrix.

```
/* matrix product of left and right */
void product(float left[][3], float right[][2],
     float prod[][2])
{
   int i, j, k;
   float prod_ij;

   for (i = 0; i<=1;i++) {
      for (j = 0; j<= 1,j++) {
         prod_ij = 0;
         for (k = 0; k <= 2;k++)
            prod_ij = prod_ij +
                      left [i][k] * right [k][j];
         prod [i][j] = prod_ij;
      }
   }
}
```



Rekursive Algorithmen

Jede C Funktion besitzt ihren eigenen lokalen Satz an Variablen.

- Dies bietet ganze neue Möglichkeiten Funktionen zu implementieren, die bis lang noch nicht betrachtet wurden.
- Ein Funktion kann sich selbst rekursiv aufrufen und ermöglicht eine ganz neue Klasse von Algorithmen, diese gehen über die bis lang betrachteten einfachen Schleifen hinaus.
- Dies ist ein mächtiges Konzept der Sprache C, das nicht in jeder Programmiersprache möglich ist. Fortran oder Cobol kennen dies nicht...

Rekursion liegt vor, wenn die Funktion f sowohl auf der linken als auch der rechten Seite einer Formel vorkommt (s. nächste Folie)!

Aus Schleife wird Rekursion...

Ein rekursiver Algorithmus lässt sich aus der Analyse der definierenden Formel gewinnen:

Beispiel: Summe

der Zahlen von 1 bis n

$$\sigma(n) = \sum_{j=1}^{n} j$$

$$\sigma(n) = n+(n-1)+\ldots+2+1$$

$$\sigma(n) = n+\sum_{j=1}^{n-1} j$$

$$\sigma(n) = n + \sigma(n-1)$$

Produkt

$$n! := \prod_{j=1}^{n} j$$

$$1! := 1$$
$$n! := n*(n-1)!$$

Rekursive Algorithmen

In der Mathematik sind viele Funktionen rekursiv definiert.

Der Begriff Rekursivität beinhaltet, daß zur Definition einer Funktion diese selbst wieder benutzt wird, allerdings mit anderen Argumenten. Eine rekursive Definition benötigt stets eine (nichtrekursive) Anfangs- bzw. Abbruchbedingung.

Beispiel Fakultät:

rekursive Definition: $fak(n)=n\ fak(n-1)$

Anfangsbedingung: $fak(0)=1$

Rekursive Algorithmen: Charakteristika

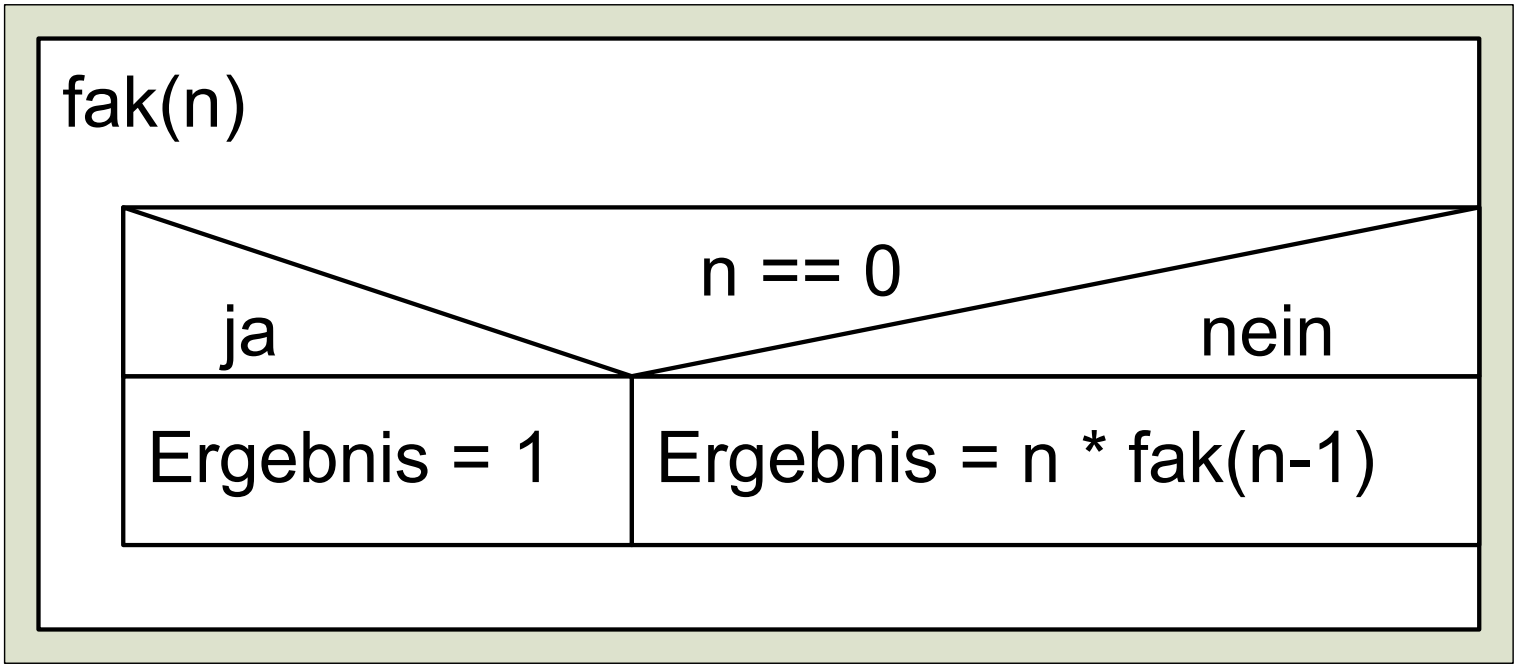
Eine solche Definition ist in der Regel kurz und übersichtlich. Man erkennt sofort die Grundstruktur des Algorithmus.

Eine solche rekursive Definition läßt sich auch leicht unter Verwendung von rekursiven Prozeduren bzw. Funktionen implementieren.

Bestimmte rekursive Algorithmen lassen sich leicht in iterative Algorithmen überführen. Dies trifft insbesondere auf die endständige Rekursion zu, wie z.B. bei der Fakultät.

Rekursive Algorithmen: Fakultät

Das Struktogramm enthält die Umsetzung der rekursiven Definition:



Rekursive UP‘s

UP‘s können sich selbst aufrufen

- direkt rekursiv: m() → m()
- indirekt rekursiv: m() → n() → m()

- Beispiel: Fakultätsfunktion

  Def: n! = 1 * 2 * … * (n-1) * n;

  also: n! = (n-1)! * n; sei 0! = 1

```
unsigned int fact(unsigned int n)
{
  if (n == 0) return (1);
  else  return (fact(n – 1)*n);
}
```

Beispiel zur Fakultät

fact (4) =
fact (3) * 4 =
(fact (2) * 3) * 4=
((fact (1) * 2) * 3) * 4 =
(((fact (0) * 1) * 2) * 3) * 4 =
(((1 * 1) * 2) * 3) * 4 =
((1 * 2) * 3) * 4 =
(2 * 3) * 4 =
6 * 4 =
24

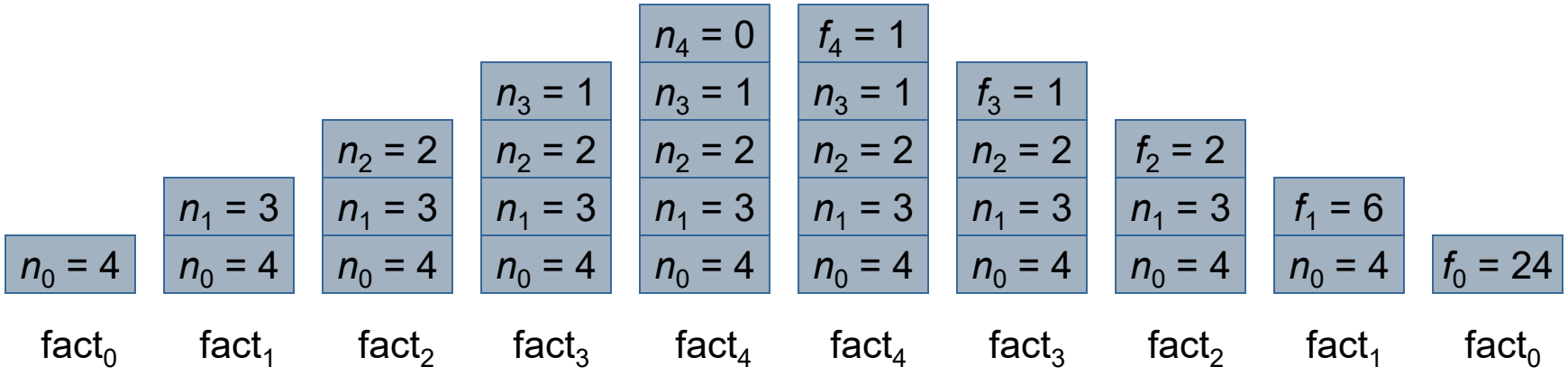
Frage:
Es gibt nur eine lokale Variable *n*, wo die vielen n_i merken?

Aufrufkeller für lokale Variablen

Beispiel: fact(4)

- Parameter sind lokale Variablen.
- Für jeden UP-Aufruf werden eigene lokale Variablen angelegt.

```
unsigned int fact (unsigned int n)
{
      if (n == 0) return (1);
      else  return (fact (n – 1) * n);
}
```



Rekursion allgemein

- Allgemeines Schema

  if *Problem_klein_genug* **then**

  nicht-rekursiver Zweig; // Terminierungsfall

  else

  rekursiver Zweig mit kleinerem Problem;

- Nachweis der Terminierung
  - Es gibt einen Terminierungszweig.
  - Das Problem wird bei jedem rekursiven Aufruf signifikant „kleiner“, d.h. der Abstand zum Terminierungsfall wird kleiner.
  - Bsp. Fakultät:

    -- Terminierungsfall: $n = 0$

    -- Verkleinerung: Aufruf mit $n - 1 < n$

Rekursion vs. Iteration

- Rekursive Lösungen
  - Rekursion ist mächtiges, allgemeines Programmierprinzip.
  - Jede rekursive Lösung läßt sich auch iterativ (d.h. mit Schleifen) lösen:
    Beispiel: Fact (n) = ProduktBis (n)
- Vergleich
  - Rekursive Formulierung ist oft eleganter.
  - Iterative Lösung ist oft effizienter aber komplizierter.

Entrekursivierung: Beispiel *Fakultät*

- Unmittelbare rekursive Lösung

```
int fact (int n)
{
  if (n == 0) return 1;
  else
    return (n *fact(n-1));
}
```

- Problem
  - Nachklappern der Multiplikationen.
  - Idee der endständigen Rekursion:
    -- führe Ergebnis rekursiv mit.
    -- *n* zählt Schritte bis zum Ende.
  - → Rekursion ohne Nachklappern!
  - → alte Werte auf dem Stapel werden nicht mehr benötigt.
  - → einfache Überführung in Iteration.

- Endständig-rekursive Lösung

```
int fact (int n){
   return fact1 (n, 1);
}

int fact1 (int n, int res){
 if (n == 0) return res;
 else return fact1(n-1,res*n);
}
```

- Iterative Lösung

```
int fact (int n) {
   int result;

   result = 1;
   while (n > 0) {
     result=result*n;
     n = n - 1;
   }
return result;
}
```

Rekursive Algorithmen: Fibonacci-Zahlen

Ein neugeborenes Hasenpaar wird in einen umzäunten Garten gesetzt. Jedes Hasenpaar „erzeugt“ während seines Lebens jeden Monat ein weiteres Paar. Ein neugeborenes Paar wird nach einem Monat fruchtbar und bekommt somit nach zwei Monaten seine ersten Nachkommen. Es soll angenommen werden, daß die Hasen nie sterben. Wie groß ist die Anzahl der Hasenpaare im n-ten Monat?

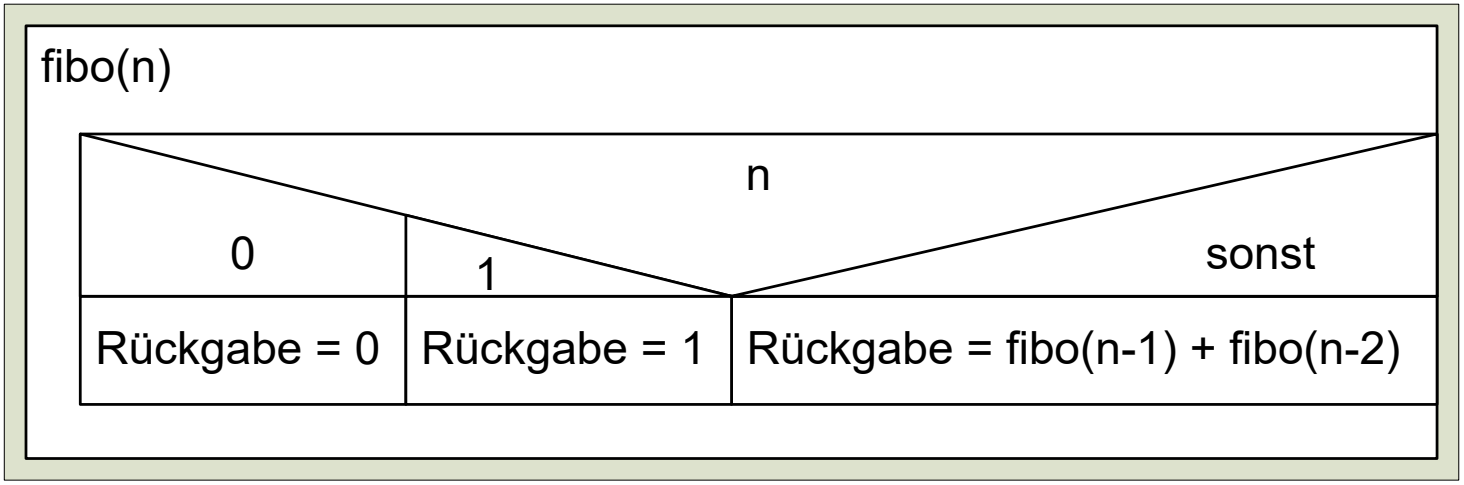
Rekursive Definition der Fibonaccischen Zahlenfolge:

$$fibo(n) = fibo(n-1) + fibo(n-2) \quad (n \geq 2)$$

Anfangsbedingung: $fibo(0)=0$, $fibo(1)=1$

Dies ergibt (beginnend bei n=0): 0, 1, 1, 2, 3, 5, 8, 13, 21, …

→Zu Beginn des zweiten Jahres sind das schon fibo(13) = 233 Mümmelnasenpaare!!



Beispiel: Fibonacci-Zahlen

- Definition

  fib(0) = 0; fib(1) = 1;

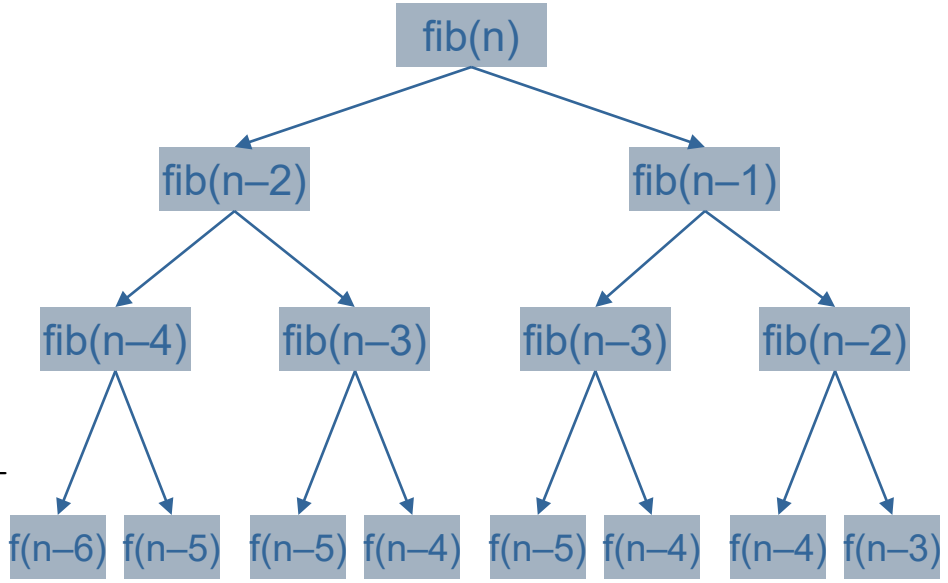
  fib(n) = fib(n – 1) + fib(n – 2) für n > 1

  Resultierende Folge:

  0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

- Problem

  Explosion der Aufrufe (= 2^n)

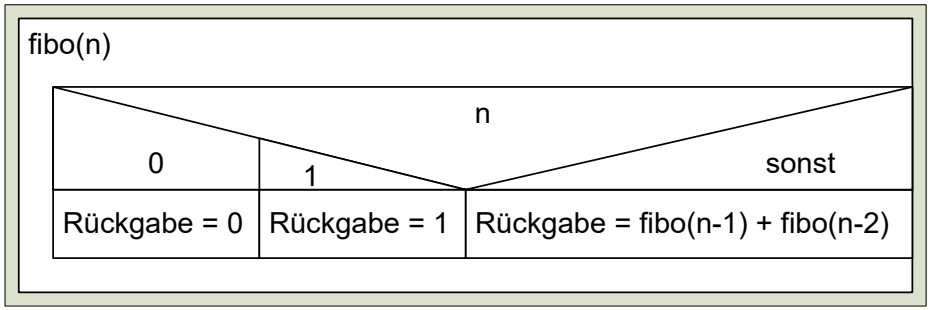


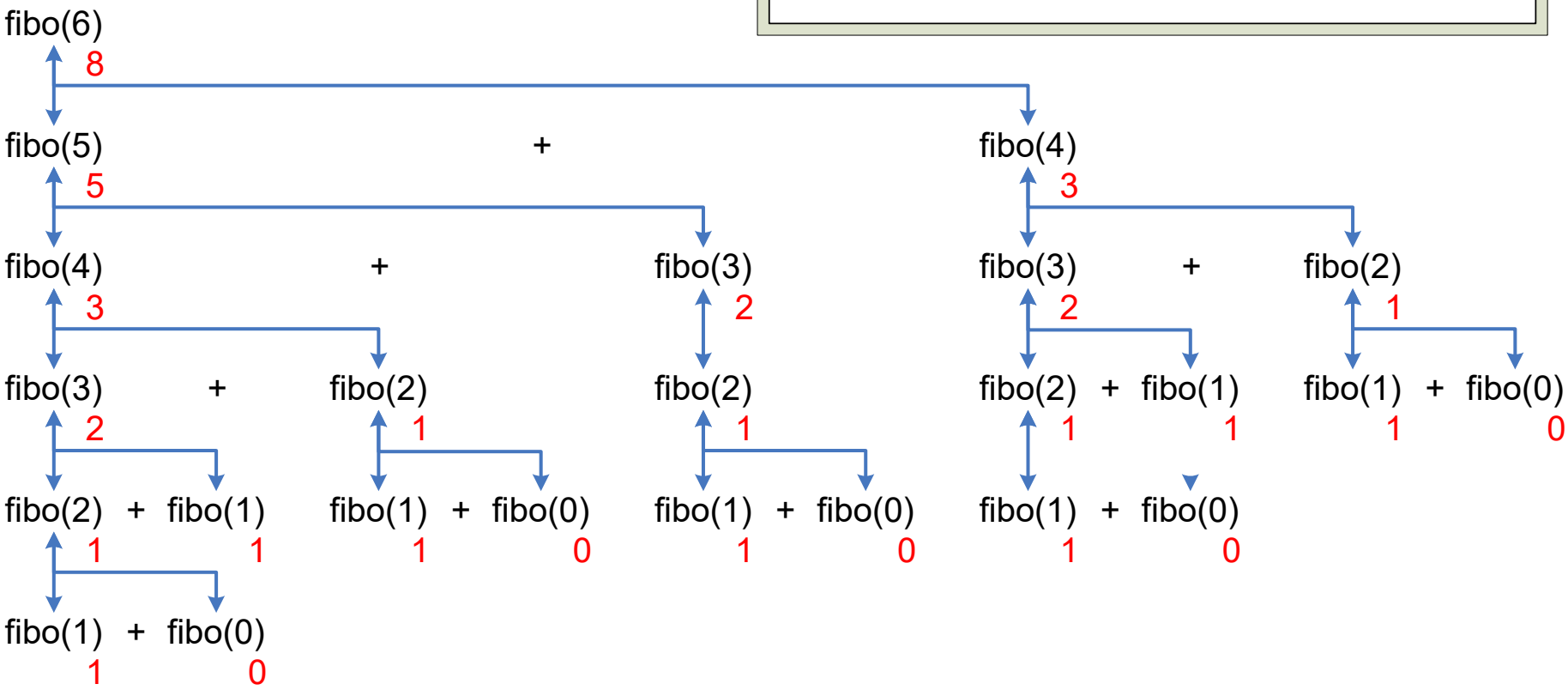
- Unmittelbare rekursive Lösung

```
unsigned int fib (
   unsigned int n)
{ if (n == 0) return (0);
   if (n == 1) return (1);
   else return (fib(n - 1) +
   fib(n - 2));
}
```

Beispiel: Fibonacci-Zahlen

Beispielrechnung für n=6





Fibonacci-Zahlen iterativ

Endständige Rekursion

- übergebe Ergebnis in Parametern.
- *n* zählt Schritte bis zum Ende (hier für n >= 1.

```
typedef unsigned int u_int;
u_int fib (u_int n)
{  return (fib1 (n, 1, 0));
}

u_int fib1(u_int n, u_int res,
           u_int p)
{  if (n ==1) return (res);
   else
   return (fib1(n-1, res +p, res));
}
```

Einfache Umwandlung in Schleife mit

Zuweisung $(n, p, pp) \leftarrow (n–1, p+pp, p)$:

Jetzt nur mehr linear viele Aufrufe.

Iterative Lösung

```
u_int fib (u_int n) {
u_int previous, pprev ;

   result = 1; previous = 0;
   while (n > 1) {
        pprev = previous;
        previous = result;
        result = result + pprev;
        n=n-1;
   }
return (result);
}
```

Vergleich zur Rekursion:

= nur lineare Anzahl Additionen in *n*.

\+ keine sich stapelnden Variablen.

Fazit Rekursion

Wann soll Rekursion in der Programmierung verwendet werden? !

- In der funktionalen Programmierung müssen rekursive Funktionen immer dann verwendet werden, wenn die Anzahl der auszuführenden Operationen nicht von vornherein beschränkt ist,
  - denn dafür ist Rekursion das einzige zur Verfügung stehende Sprachkonzept (es gibt keine Iteration).

- Rekursive Funktionen eignen sich besonders, falls die Datentypen, auf denen die Funktionen arbeiten, selbst rekursiv sind

Beispiel: rekursive Datentypen - Binärbaum

```
typedef struct  struct{
  int value;
  struct node *left;
  struct node *right;
}node;

int x;

node *insert( node *root) {
if( root==NULL) {                                  //1. neues Element ist die Wurzel des Baums
    root=(node*) malloc( sizeof( node));
    root->value=x;
    root->left=root->right=NULL;
  }
  // 2. x ist kleiner als die Wurzel des (Teil-)Baumes – Einordnung links:
  else if (root->value >= x)  root->left=insert(root->left); // rekursiver Aufruf
  // 3. x ist kgrößer als die Wurzel des (Teil-)Baumes – Einordnung rechts:
  else if (root->value < x)    root->right = insert( root->right);
  return root;
}
```