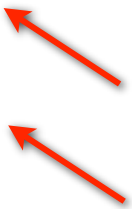
Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung: Zeiger / Pointer

```
int   i = 10;
int  *j = &i;
int **k = &j;
printf("%x\t%d\n", &i, i);
printf("%x\t%x\t%d\n", &j, j, *j);
printf("%x\t%x\t%x\t%d\n", &k, k, *k, **k);
```

name	address	contends
i	0x123aa8	10
j	0x123aab	0x123aa8
k	0x123ab0	0x123aab

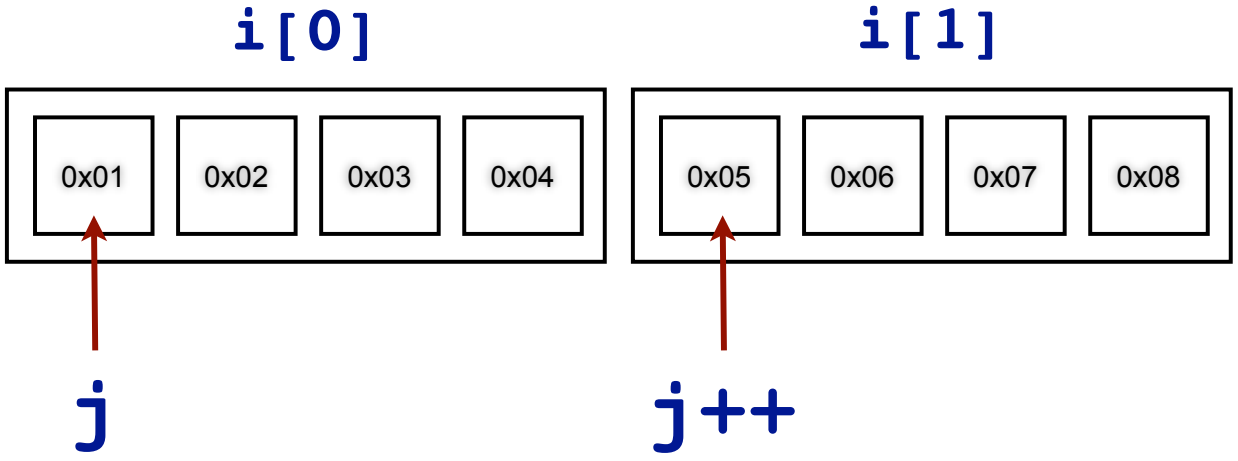


Letzte Vorlesung: Pointerarithmetik

- Man kann Integers von Pointern subtrahieren und addieren

```
int i[5] = { 1, 2, 3, 4, 5 };
int *j = i;   /* (*j) == ? */
j++;          /* (*j) == ? */
j += 2;       /* (*j) == ? */
j -= 3;       /* (*j) == ? */
```

- Pointerarithmetik addiert/subtrahiert Adressen nicht direkt, sondern verschiebt in Vielfachen der Bytezahl eines Typs:



Gliederung

Dynamische Speicherbereitstellung (1)

- Erinnerung: C Erlaubt folgende Anweisungen nicht:

```
int n = 10;
int arr[n];  /* not legal C */
```

- Dynamische Speicherbereitstellung:

```
void *malloc(int size)
void *calloc(int nitems, int size)
```

- Speicherfreigabe

```
void free(void *ptr)
```

- Funktionsdefinitionen in `<stdlib.h>`
- Was bedeutet `void*`?
- Ein Zeiger auf irgendwas
- Datentyp muss bei Zuweisung durch Type-Cast angegeben werden

Dynamische Speicherbereitstellung (2)

- `malloc` und `calloc` geben die Adresse des allokierten Speicherblocks zurück
- Die Operationen können fehlschlagen!
- Wenn sie fehlschlagen, wird `NULL` zurück gegeben

```
int *arr = (int *) malloc(10 * sizeof(int));
/* code that uses arr... */
```

- Besser:

```
int *arr = (int *) malloc(10 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "out of memory!\n");
    exit(1);
}
```

Dynamische Speicherbereitstellung (3)

- Speicherfreigabe

```
#include <stdlib.h>
int *foo(int n) {
  int i[10];  /* memory allocated here */
  int i2[n];  /* ERROR: NOT VALID! */
  int *j;

  j = (int *)malloc(n * sizeof(int));
  /* Alternatively: */
  /* j = (int *)calloc(n, sizeof(int)); */

  return j;
} /* i's memory deallocated here; j's not */
```

Dynamische Speicherbereitstellung (4)

- Memory leaks

```
void leaker() {
  int *arr = (int *)malloc(10 * sizeof(int));
  /* Now have allocated space for 10 ints;
   * do something with it and return without
   * calling free().
   */
} /* arr memory is leaked here. */

void not_leaker() {
  int *arr = (int *)malloc(10 * sizeof(int));
  /* Now have allocated space for 10 ints;
   * do something with it.
   */
   free(arr);  /* free arr's memory */
} /* No leak. */
```

Dynamische Speicherbereitstellung (5)

- Memory leaks

```
void not_leaker2() {
  int arr[10];
  /* Now have allocated space for 10 ints;
   * do something with it.
   */
} /* No leak. */
```

- Hier muss `free()` nicht aufgerufen werden, da statisch auf dem Stack bereitgestellter Speicher automatisch freigegeben wird

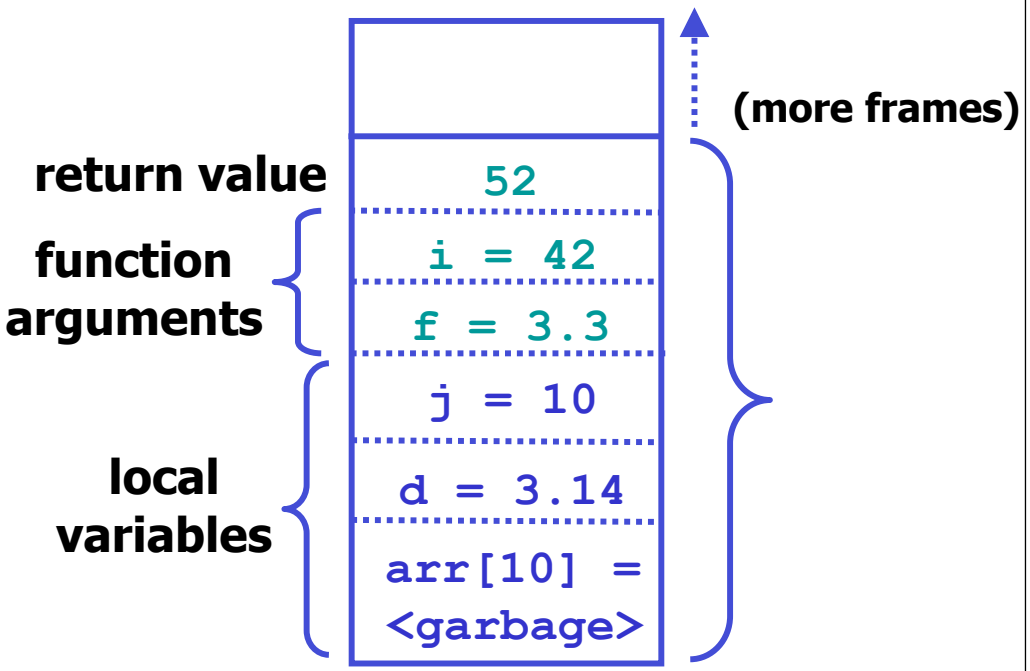
```
void crasher() {
  int arr[10];
  /* Now have allocated space for 10 ints;
   * do something with it.
   */
  free(arr); /* BAD! */
}
```

Stack und Heap (1)

- Lokale Variablen, Argumente von Funktionen und Rückgabewerte werden auf dem Stack gespeichert
- Für jeden Funktionsaufruf wird ein eigener Stack-Frame angelegt, in dem eine begrenzte Anzahl an Daten gespeichert werden können
- Belegter Speicher auf dem Stack wird freigegeben, sobald ein Stack-Frame gelöscht wird
- Wenn eine Funktion beendet wird, wird der Stack-Frame gelöscht
- Frage: Wie sieht der Stack aus?

Stack und Heap (2)

```
float contrived_example(int i, float f)
{
    int j = 10;
    double d = 3.14;
    int arr[10];
    /* do some stuff,
       then return */
    return (j + i);
}
```



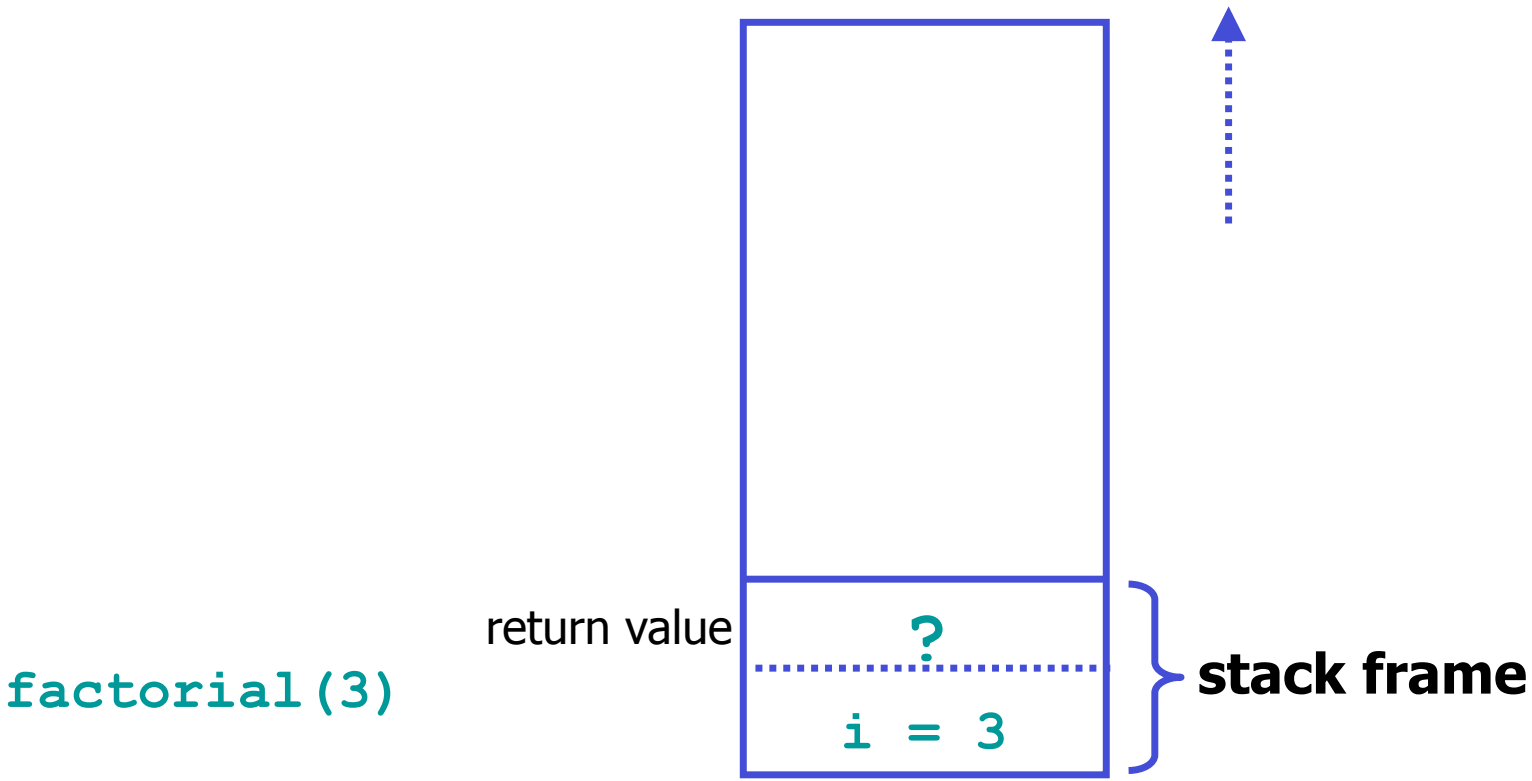
Stack und Heap (3)

- Weiteres Beispiel:

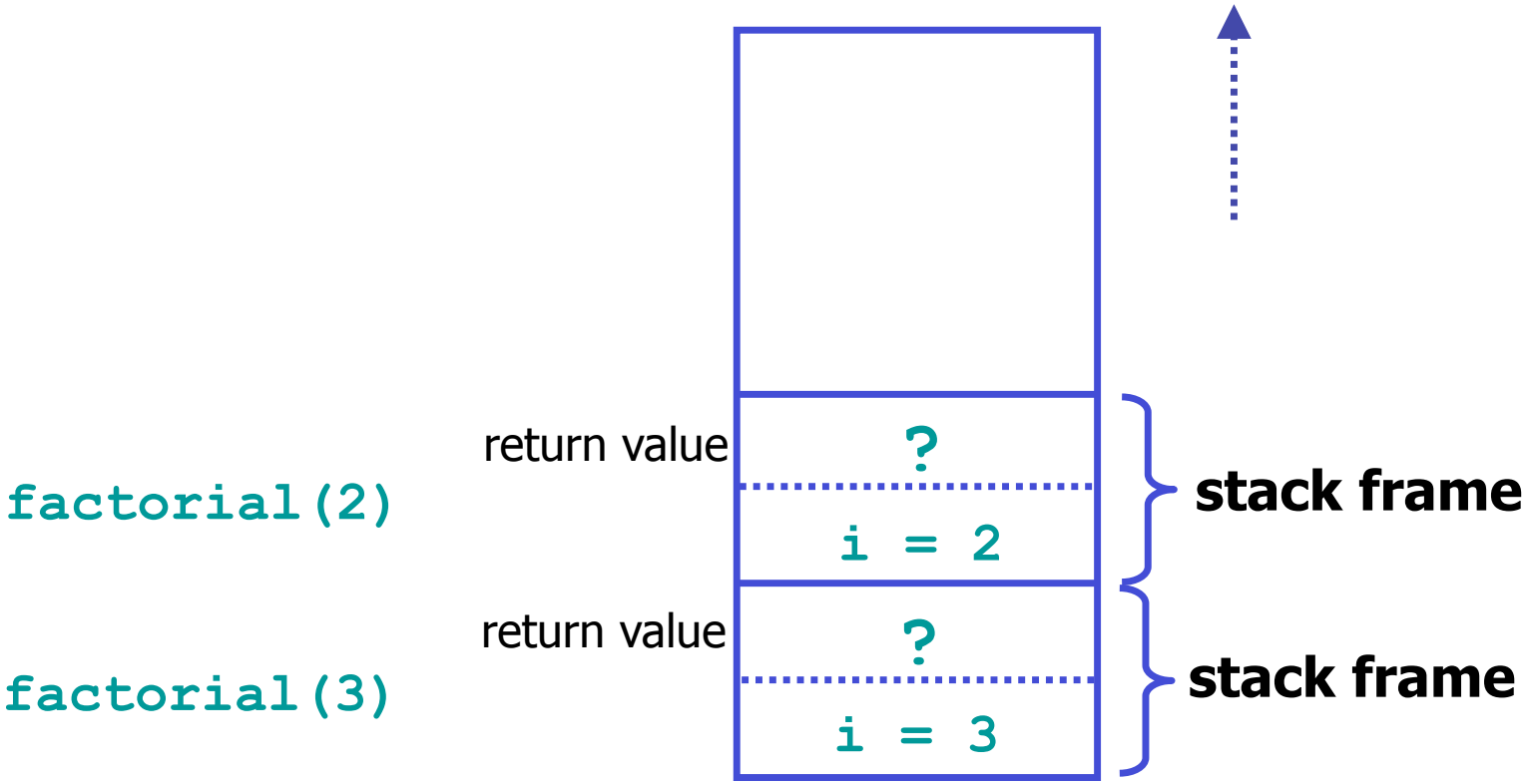
```
int factorial(int i)
{
    if (i == 0) {
        return 1;
    } else {
        return i * factorial(i - 1);
    }
}
```

- Wie sieht der Stack aus für `factorial(3)`?
- Für jeden Stack-Frame haben wir
  - Hier: keine lokalen Variablen
  - Ein Argument (`i`)
  - Einen Rückgabewert
- Jeder Rekursionsaufruf erzeugt einen Stack-Frame

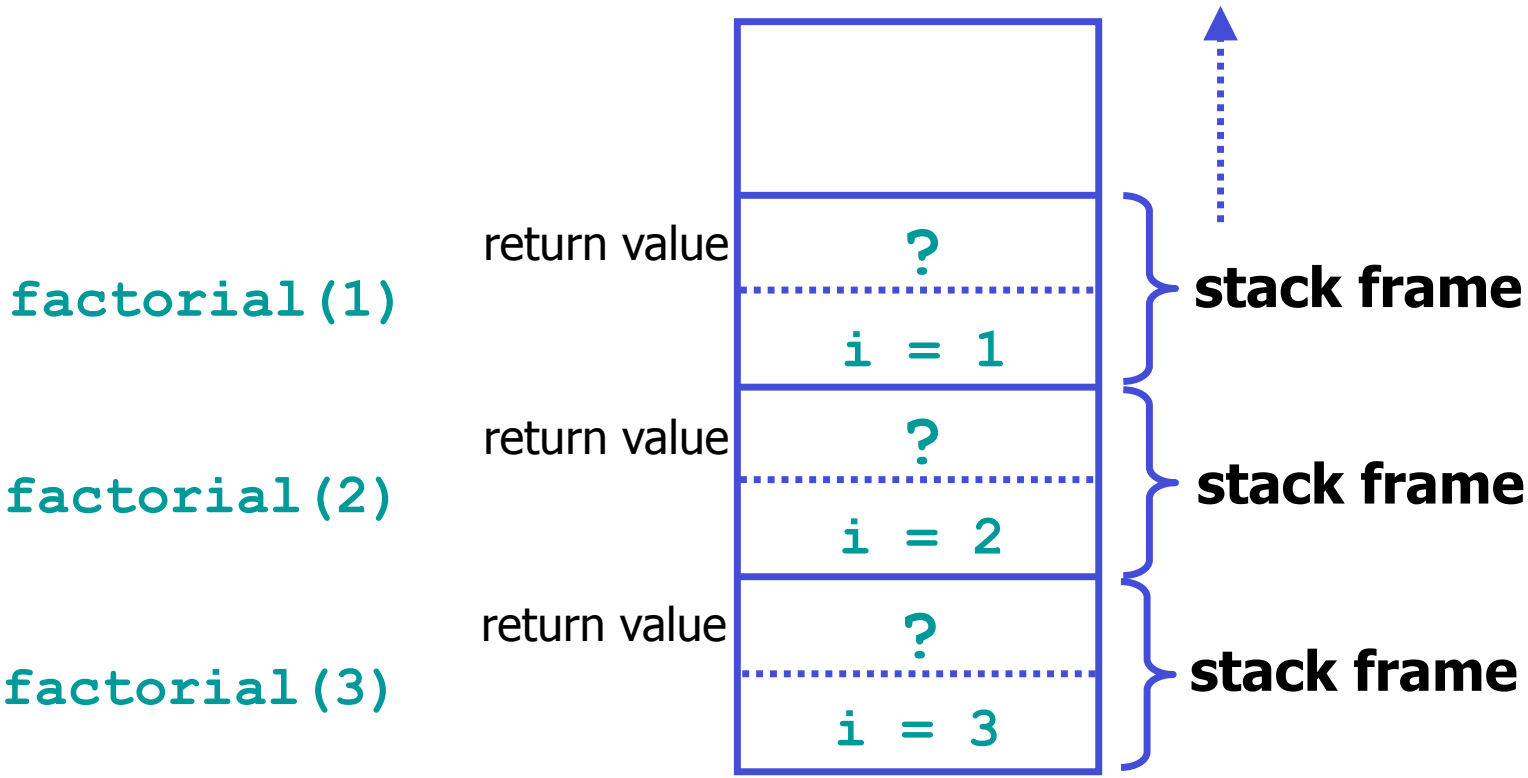
Stack und Heap (4a)



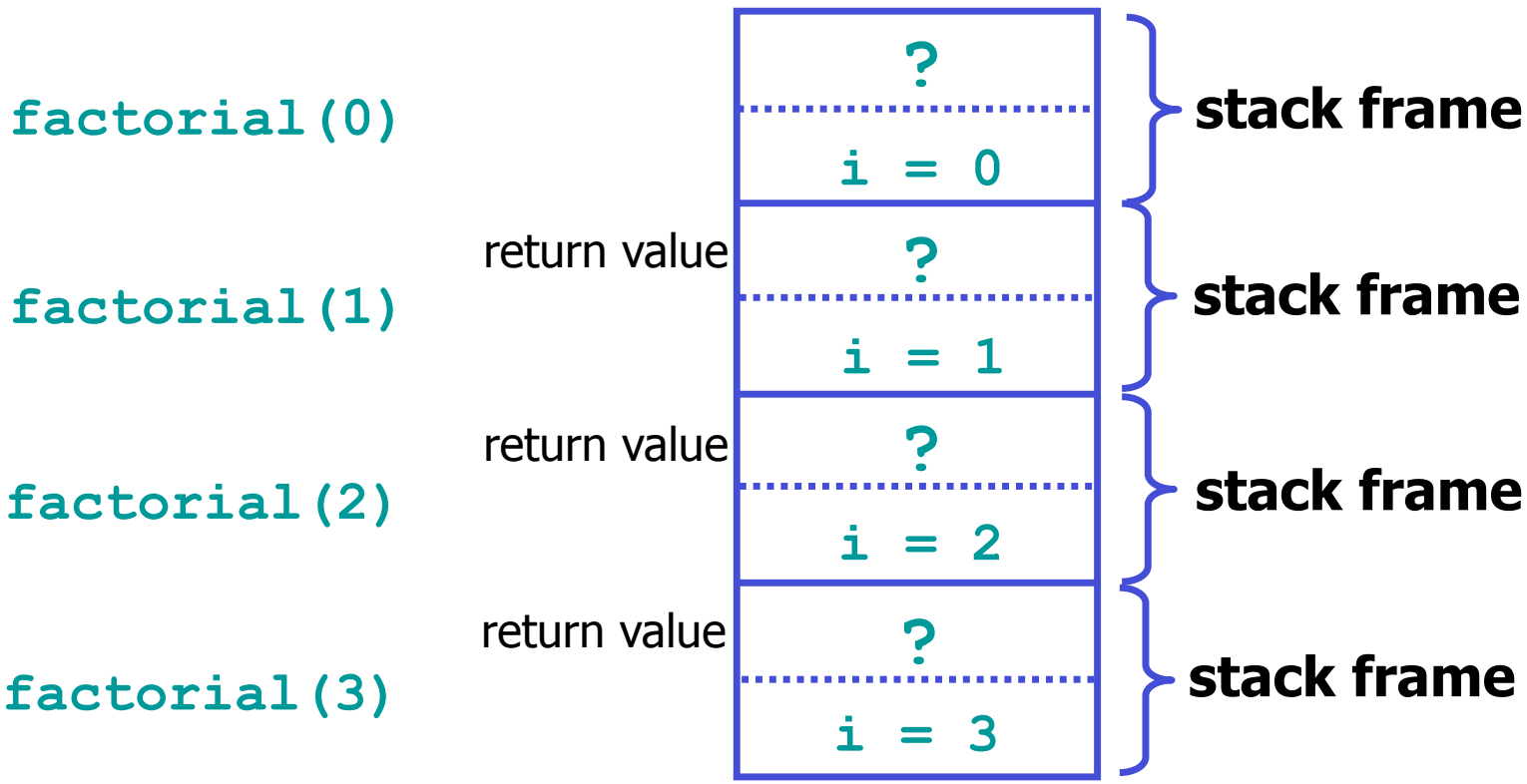
Stack und Heap (4b)



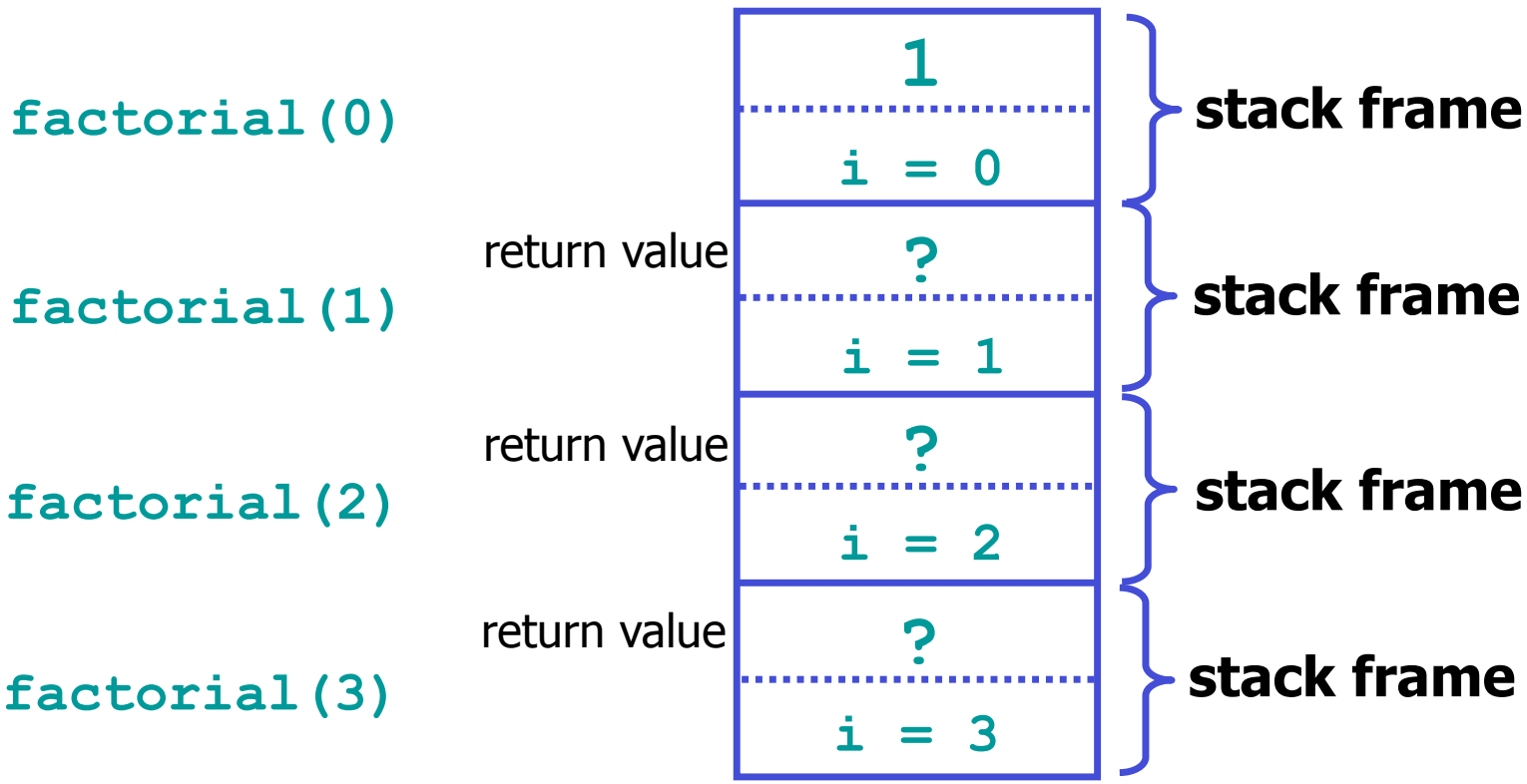
Stack und Heap (4c)



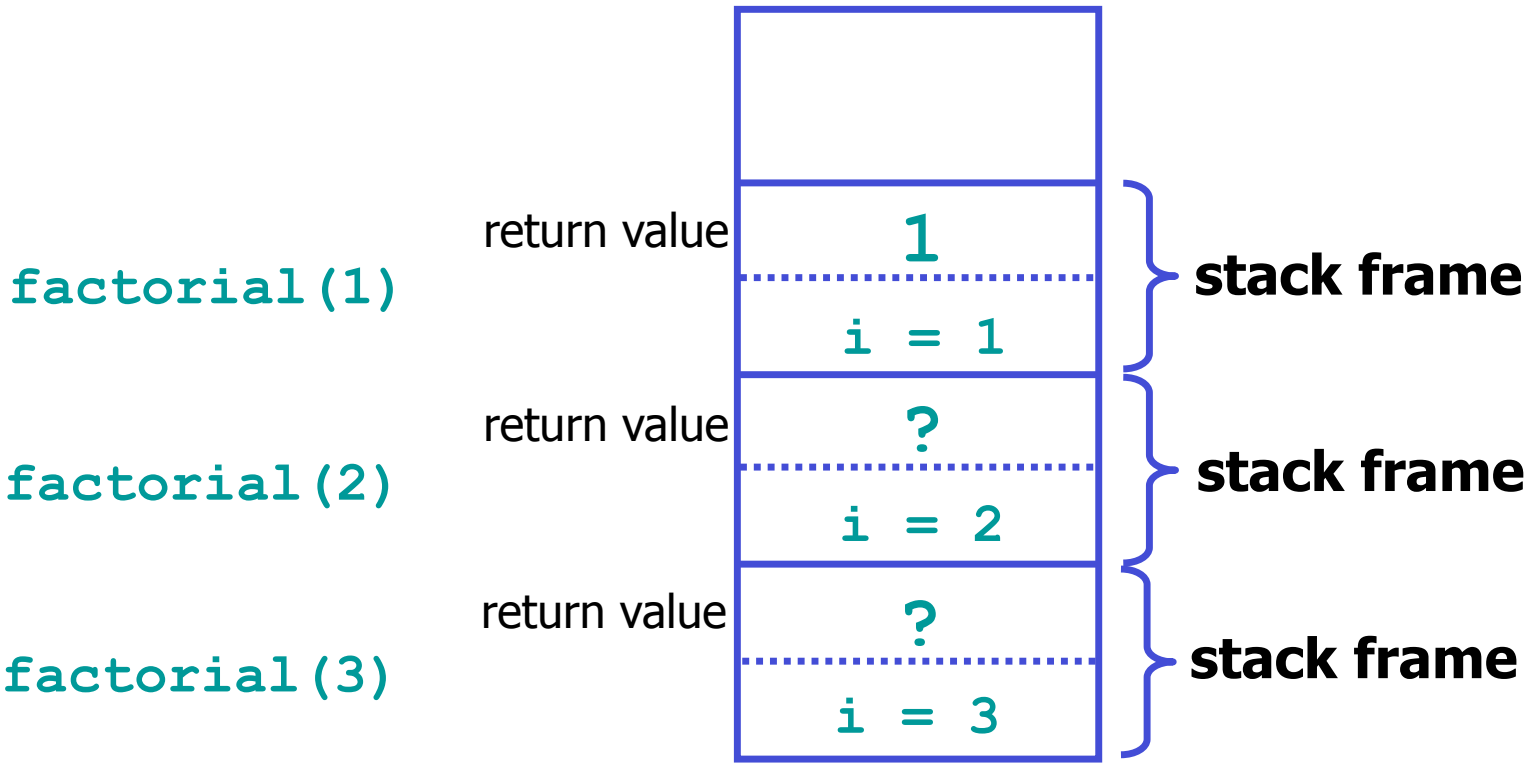
Stack und Heap (4d)



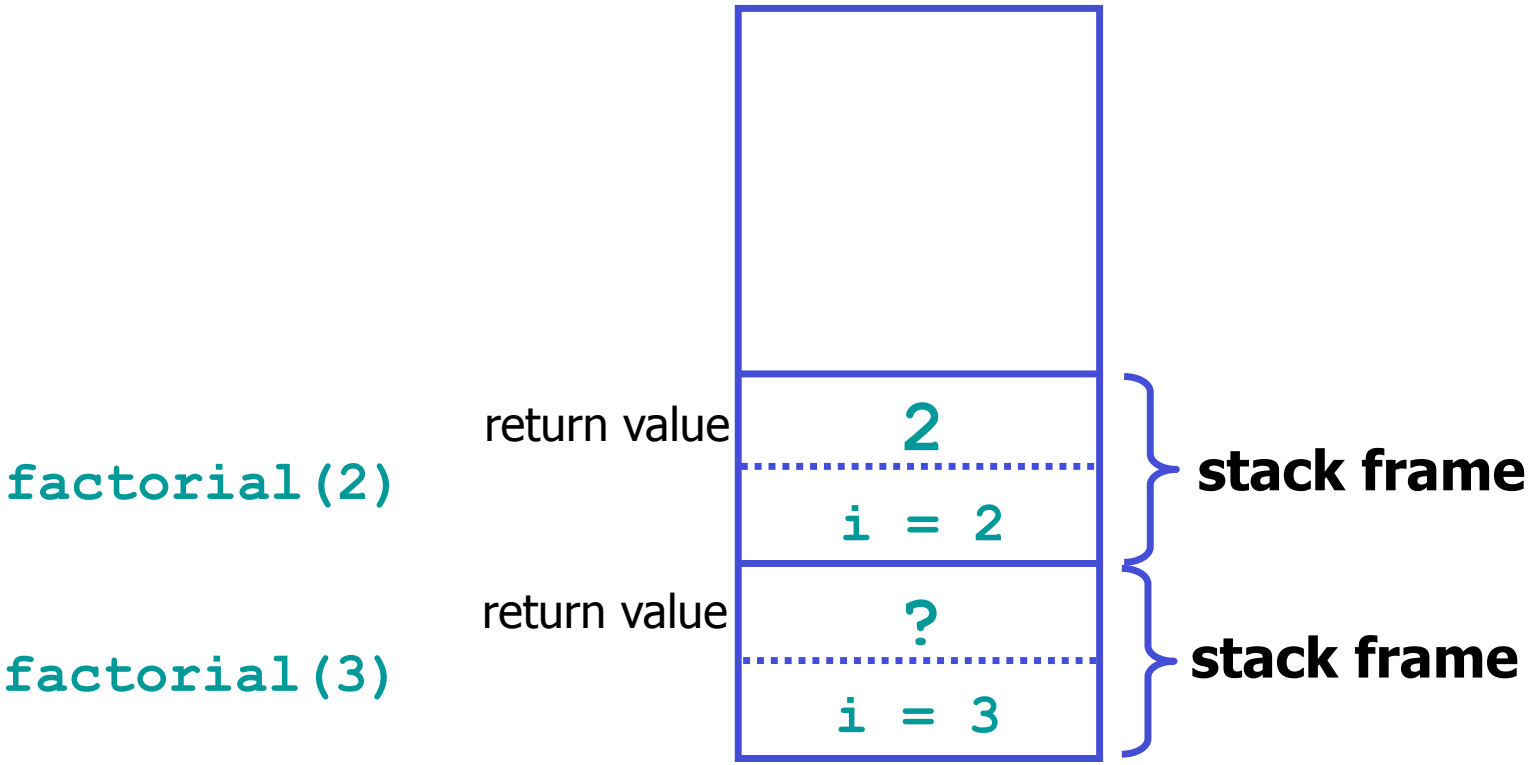
Stack und Heap (4e)



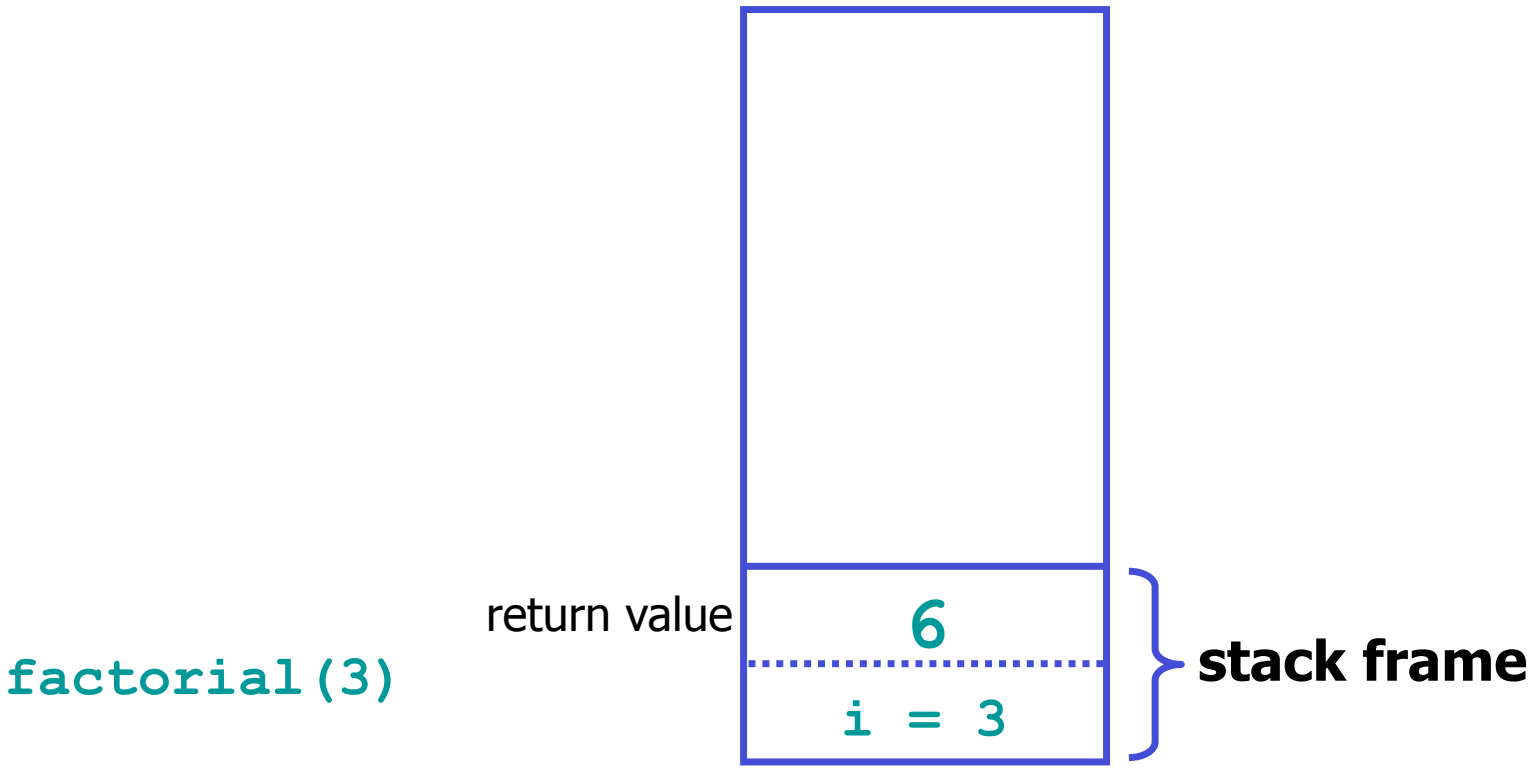
Stack und Heap (4f)



Stack und Heap (4g)



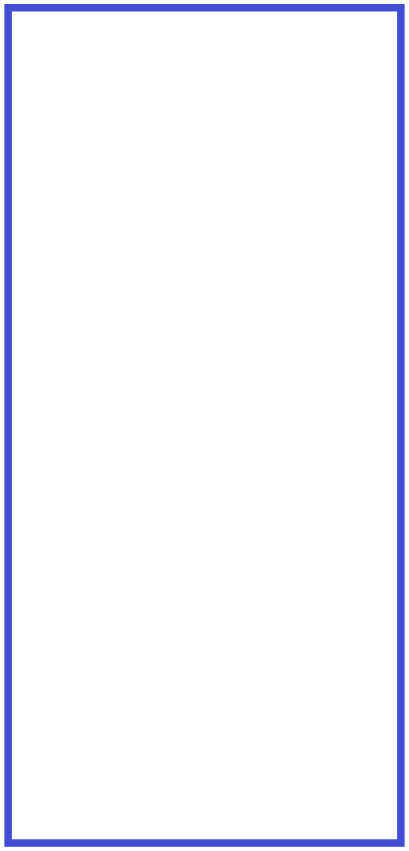
Stack und Heap (4h)



Stack und Heap (4i)

```
factorial(3)
```

result: 6



Stack und Heap (5)

- Heap ist allgemein verfügbarer, nicht verwalteter, Arbeitsspeicher
- Speicher im Heap wird mit einem der `*alloc()`-Varianten reserviert und muss mit `free()` freigegeben werden
- Beispiel:

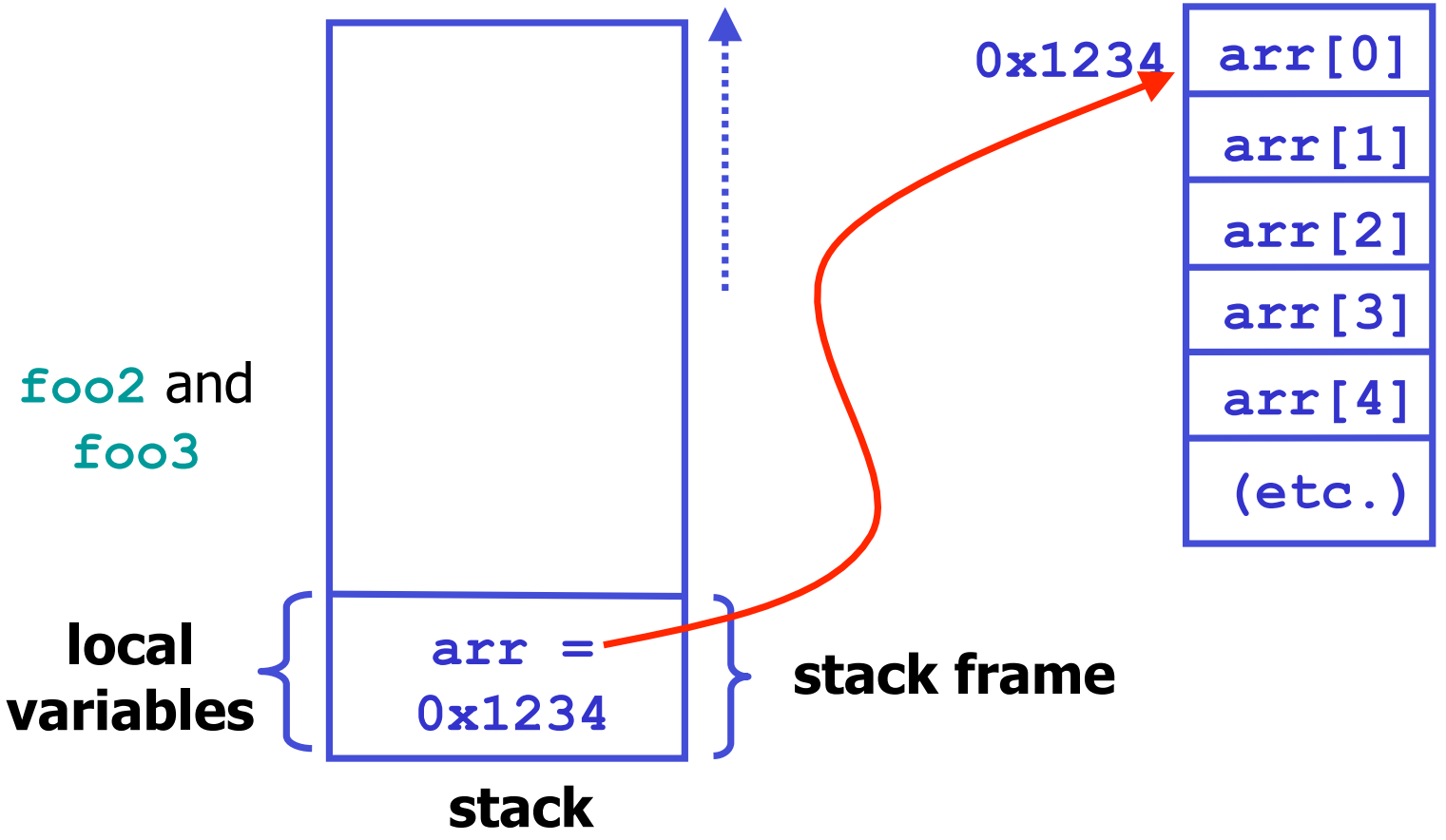
```
void foo2() {
    int *arr;

    /* allocate memory on the heap: */
    arr = (int *)calloc(10, sizeof(int));

    /* do something with arr */
}
/* arr is NOT deallocated */
```

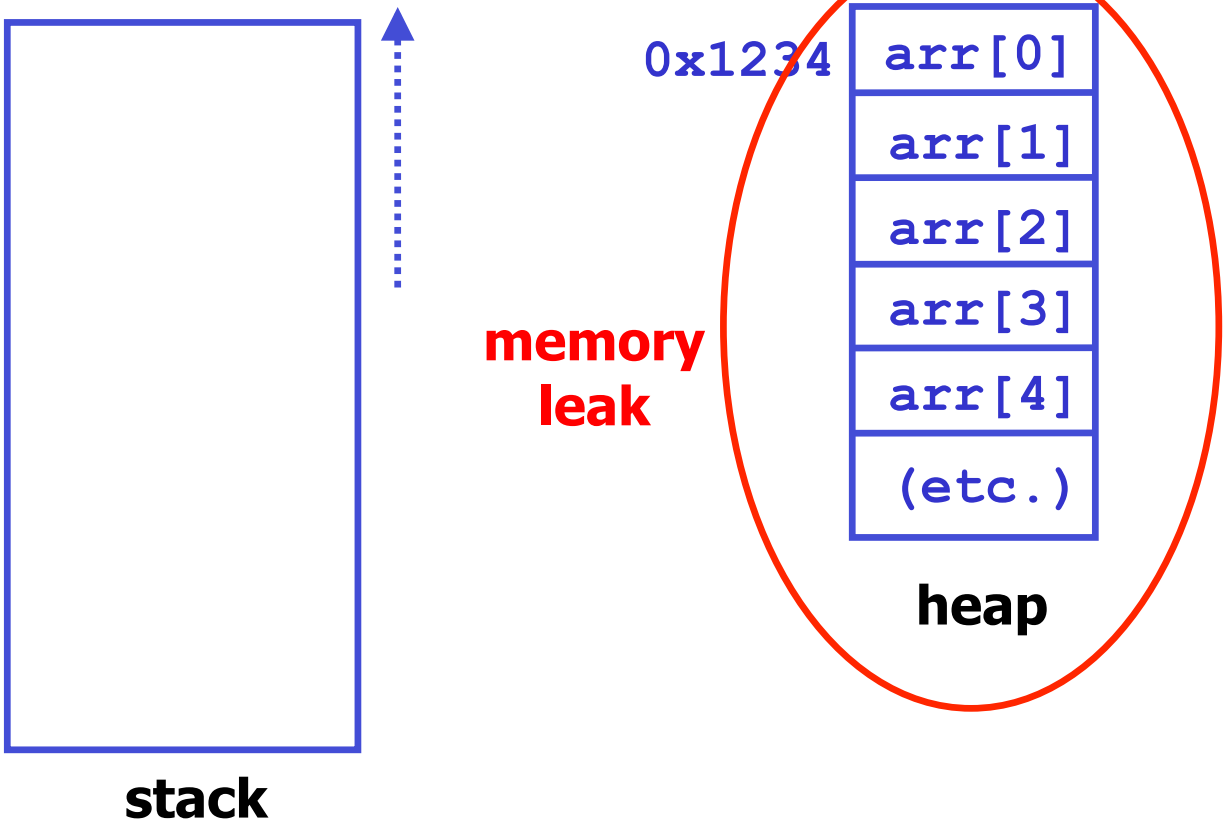
- Wie sehen Stack und Heap in diesem Fall aus?

Stack und Heap (6a)



Stack und Heap (6b)

(after `foo2` exits, without freeing memory)



Dynamische Speicherbereitstellung (6)

- 2D-Arrays in C
  - Statisch:

```
double data[100][3];  /* 100 * double[3] */
```

  - Zugriff:

```
int i;
for (i = 0; i < 100; i++) {
  data[i][0] = 10.0;
  data[i][1] = 1.0;
  data[i][2] = 0.0;
}
```

  - Dynamisch:

```
double **data;
data = (double**)malloc(100 * sizeof(double*));
for (i = 0; i < 100; i++)
  data[i] = (double*)malloc(3 * sizeof(double));
```

Dynamische Speicherbereitstellung (7)

- 2D-Arrays in C
- Zugriff: Wie im statischen Fall
- Speicherfreigabe:

```
free(data);
```

- Ist nicht ausreichend!
- Korrekt ist:

```
/* free memory */
for (i = 0; i < 100; i++) {
    free(data[i]);
}
free(data);
```

- Merke: Für jedes `malloc()` muss in der Regel ein `free()` aufgerufen werden

Gliederung

Strukturen: `struct` (1)

- Eine Möglichkeit, mehrere Datentypen in einem zusammengesetzten Datentyp zu vereinigen
- Deklaration:

```
struct point {
    int x;
    int y;
    double dist; /* from origin */
};  /* MUST have semicolon! */
```

- Erfolgt in der Regel außerhalb von Funktionen
- Erzeugung / Initialisierung:

```
struct point p;
p.x = 0;    /* "dot syntax" */
p.y = 0;
p.dist = sqrt(p.x * p.x + p.y * p.y);
```

Strukturen: `struct` (2)

- Benutzung von `structs`:

```
void foo() {
    struct point p;
    p.x = 10; p.y = -3;
    p.dist = sqrt(p.x * p.x + p.y * p.y);
    /* do stuff with p */
}
```

- Dynamische Allokierung von `structs`:

```
struct point *make_point(void) {
    struct point *p;
    p = (struct point *)
      malloc(sizeof(struct point));
    return p;
}  /* free struct elsewhere */
```

Strukturen: `struct` (3)

- Pointer auf `structs`:

```
void init_point(struct point *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;

    /* syntactic sugar: */
    p->x = p->y = 0;
    p->dist = 0.0;
}
```

- Strukturen können selber auch `structs` enthalten:

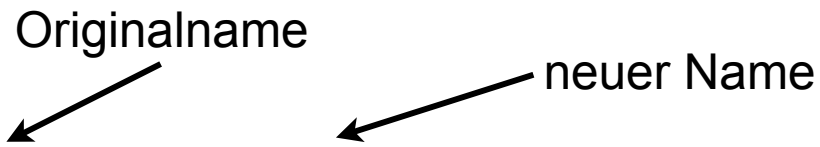
```
struct foo {
    int x;
    struct point p1;    /* Unusual */
    struct point *p2;   /* Typical */
};
```

struct / typedef (1)

- Rekursive `structs`

```
struct node {
    int value;
    struct node *next;
};
```

- ... können aber nicht `struct node next` enthalten
- Warum?
- Wiederkehrende Eingabe von `struct point` lästig
- Benutze `typedef` als Alias:




```
typedef struct point Point;
typedef int Length;
```

struct / typedef (2)

- Typenkomponente in typedef kann auch ein struct sein

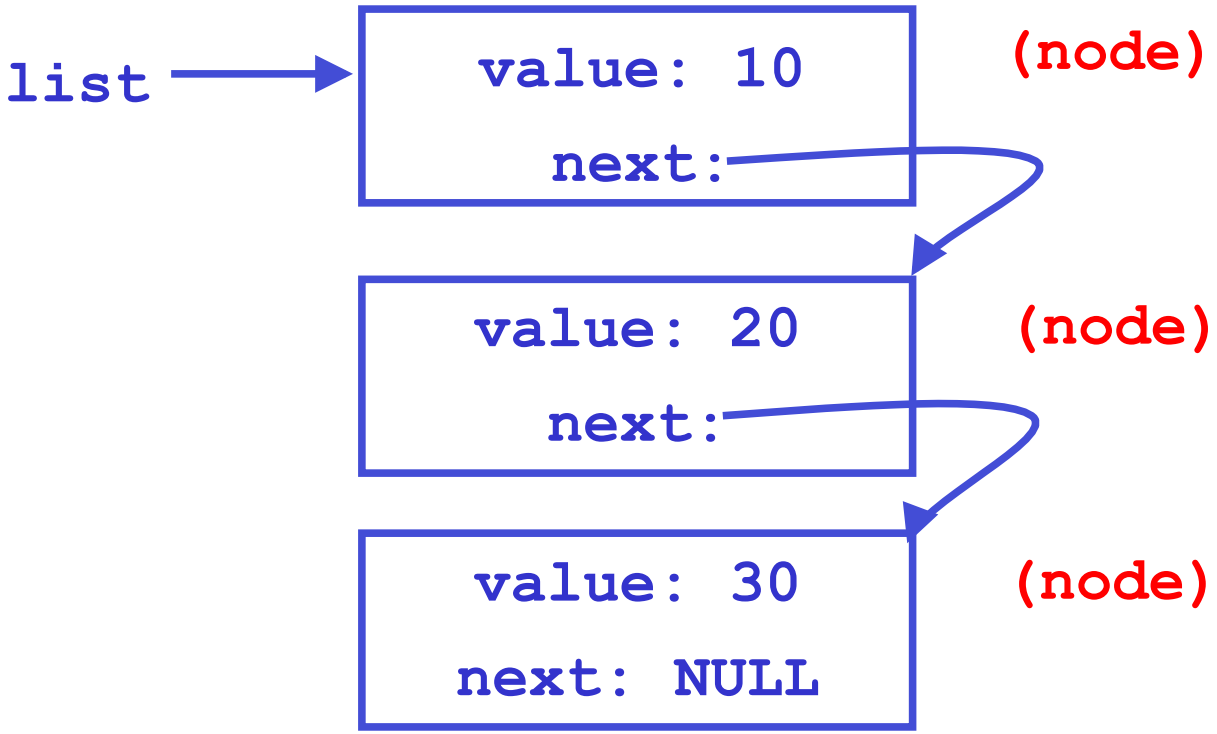
```
typedef struct {          /* no name for struct */
    int x;
    int y;
    double dist;
} Point;
Point p1, p2;              /* no "struct" */
```

- Dies ist eine anonyme Struktur
- Rekursiv

```
typedef struct _node {
    int value;
    struct _node *next;
} node;
```

- Diese Struktur ist nicht anonym

Beispiel: Verkettete Liste (1)



- `node` ist in der Liste
- `next` zeigt auf nächsten `node`
- Wenn `next == NULL`, dann ist das Ende der Liste erreicht

Beispiel: Verkettete Liste (2)

- Erzeugung einer verlinkten Listen:

```
list = NULL;                    /* Empty list. */

node *n = (node *) malloc(sizeof(node));
n->value = 30;
n->next = NULL;
list = n;                       /* now 1-node list */

node *n = (node *) malloc(sizeof(node));
n->value = 20;
n->next = list;
list = n;                       /* now 2-node list */

node *n = (node *) malloc(sizeof(node));
n->value = 10;
n->next = list;
list = n;                       /* now 3-node list */
```

- ... ohne Check von `malloc()` auf Fehler!!!!

Beispiel: Verkettete Liste (3)

- Iteration durch Liste:

```
node *n;
/* Set all node values to zero. */
for (n = list; n != NULL; n = n->next) {
    n->value = 0;
}
```

union (1)

- unions sind ähnlich wie structs
- Schlüsselwort union

```
union {
  float f;
  int i;
} var;
```

- Im Gegensatz zu structs wird in unions nicht für jeden „Member“ Speicher bereit gestellt. Die Größe einer union hängt vom „längsten“ Datentyp ab
- D.h. es lässt sich immer nur eine der Wahlmöglichkeiten speichern

```
var.f = 23.5;
printf("value is %f\n", var.f);
var.i = 5;
printf("value is %d\n", var.i);
```

union (2)

- Oft treten unions und structs gemeinsam auf

```
/* code for types in union */
#define FLOAT_TYPE 1
#define INT_TYPE 2

struct var_type {
  int type_in_union;
  union {
    float un_float;
    int un_int;
  } vt_un;
} var_type;

/* [snip] */
switch(var_type.type_in_union){
   default:
     printf("Unknown type in union\n"); break;
   case FLOAT_TYPE:
     printf("%f\n", var_type.vt_un.un_float); break;
```

`enum` - Aufzählungen

- Einsatz sehr häufig
- Definition etwas halbherzig (in Pascal gibt es richtige Aufzählungstypen)

```
enum e_tag {
  A, B, C, D = 20, E, F, G = 20, H
} var;
```

- In C helfen sie, die Anzahl der `#defines` zu verringern
- Ersetzt

```
#define A 0
#define B 1
/* and so on */
```

- Im Programm können nun die Abkürzungen verwendet werden

```
switch(var){
     case D: break;
}
```

- Numerische Werte einer Variablen spielen häufig keine Rolle

Externe Variablen

- Manchmal ist es nötig, dass mehrere Dateien sich eine globale Variable teilen
- Diese Variable ist nur einmal definiert
- extern-Deklaration in den Header-Files, d.h. die Variable ist woanders definiert
- Symbol für die Variable wird beim Linken gefunden und ersetzt

```
/* In header file "foo.h": */
extern int max_value;

/* In header file “bar.h“: */
extern int max_value;

/* In file "foo.c": */
/* global variable: */
int max_value = 1000000;
```

const

- Wir kennen bereits

```
#define SOME_CONSTANT 100
```

- Besser:

```
const int SOME_CONSTANT = 100;
```

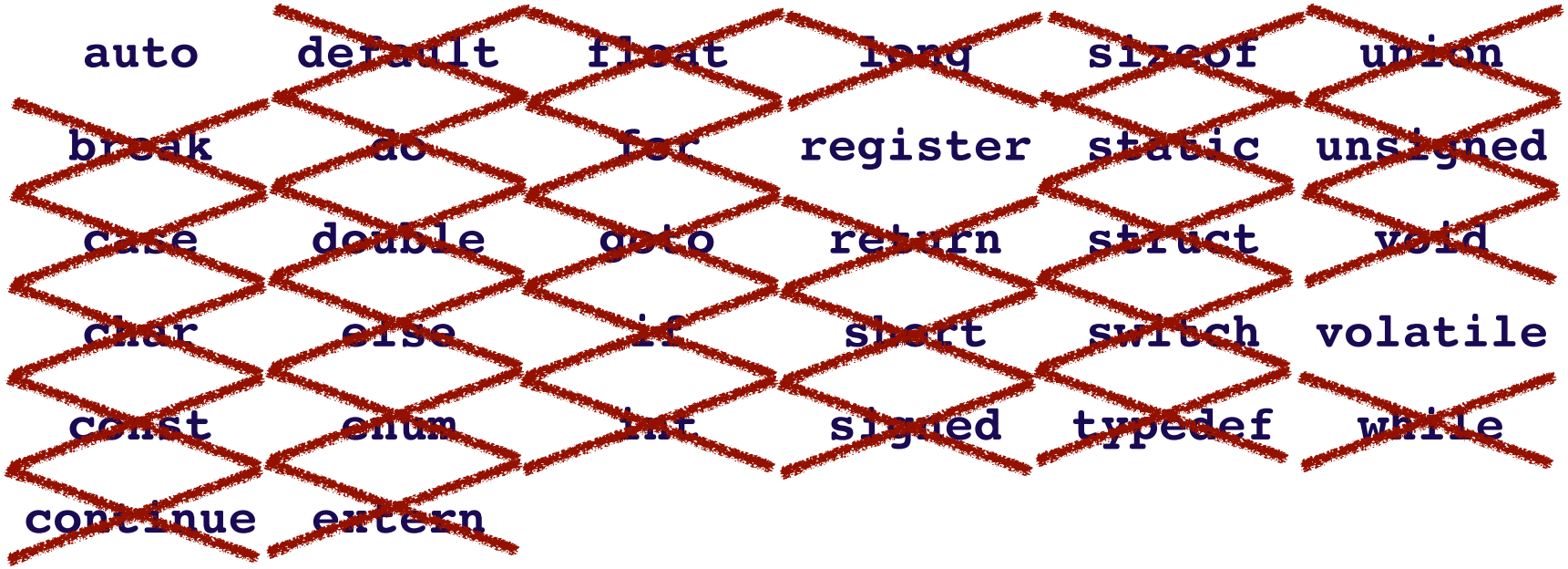
- Warum ist das besser?
- Eine mit `const`-Variablen lassen sich auch statische Arrays erzeugen:

```
const int ARRAY_SIZE;
int array[ARRAY_SIZE];
```

`static`

- zwei Bedeutungen
- bei Variablen:
  - Wert wird an einer festen Adresse auch nach Ablauf einer Funktion gespeichert
  - Man kann bei nächsten Aufruf der Funktion auf den Wert beim letzten Durchlauf zugreifen
- bei Funktionen:
  - die als `static` deklarierte Funktion wird auf die aktuelle Datei beschränkt
  - d.h. falls der Name öfters vorkommt, wird die in der aktuellen Datei bekannte Version verwendet, vorher definierte sind „vergessen“

Bestandsaufnahme - ANSI C Schlüsselwörter



Bestandsaufnahme - ANSI C Schlüsselwörter

auto	default	float	long	sizeof	union
break	do	for	register	static	unsigned
case	double	goto	return	struct	void
char	else	if	short	switch	volatile
const	enum	int	signed	typedef	while
continue	extern				

- `auto`, `register` und `volatile` sind i.d. Regel obsolet
- Ursprünglich gedacht, um dem Compiler Informationen zu geben in welchen Speicher eine Variable gehalten werden soll
- Noch verschwiegen:
  - Unbestimmte Anzahl Parameter (`printf()`)
  - Bitpatterns in `structs`