10 Werteaustausch zwischen Funktionen

Mechanismus	Aufrufer → Funktion	Funktion → Aufrufer
Parameter	ja	mit Hilfe von Zeigern
Funktionswert	nein	ja
globale Variablen	ja	ja

- Verwendung globaler Variablen?
  - Variablen, die von mehreren Funktionen verwendet werden und/oder oft als Parameter übergeben werden müssten
    - ➤ Menge der Funktionen muss überschaubar bleiben
      → Zugriff auf Modul begrenzen (globale static-Variablen)
    - ➤ **sonst sehr schlechter Programmierstil**
  - Variablen, die keiner Funktion als Variable oder Parameter fest zugeordnet werden können
    - ➤ grundsätzlich fragwürdig – evtl. ein Design-Fehler?
    - ➤ sonst Modul suchen, dem die Variable zugeordnet werden kann

11 Getrennte Übersetzung von Programmteilen — Beispiel

- Hauptprogramm (Datei **`fplot.c`**)

```
#include "trig.h"
#define INTERVALL 0.01

/*
 * Funktionswerte ausgeben
 */
int main(void)
{
        char c;
        double i;

        printf("Funktion (Sin, Cos, Tan, Cot)? ");
        scanf("%x", &c);

        switch (c) {
        ...
        case 'T':
          for (i=-PI/2; i < PI/2; i+=INTERVALL)
                printf("%lf %lf\n", i, tan(i));
          break;;
        ...
        }
        return(0);
}
```

11 Getrennte Übersetzung — Beispiel (2)

- *Header*-Datei (Datei **`trig.h`**)

```
#include <stdio.h>
#define PI 3.1415926535897932
double tan(double), cot(double);
double cos(double), sin(double);
```


11 Getrennte Übersetzung — Beispiel (3)

- Trigonometrische Funktionen (Datei **`trigfunc.c`**)

```
#include "trig.h"

double tan(double x) {
       return(sin(x)/cos(x));
}

double cot(double x) {
       return(cos(x)/sin(x));
}

double cos(double x) {
       return(sin(PI/2-x));
}
```

...

11 Getrennte Übersetzung — Beispiel (4)

- Trigonometrische Funktionen — Fortsetzung
  (Datei **`trigfunc.c`**)

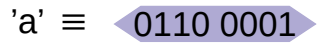
...

```
double sin (double x)
{
        double summe;
        double x_quadrat;
        double rest;
        int k;

        k = 0;
        summe = 0.0;
        rest = x;
        x_quadrat = x*x;

        while ( fabs(rest) > 1e-9 ) {
                summe += rest;
                k += 2;
                rest *= -x_quadrat/(k*(k+1));
        }
        return(summe);
}
```

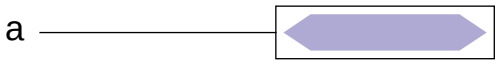

Zeiger(-Variablen)

1 Einordnung

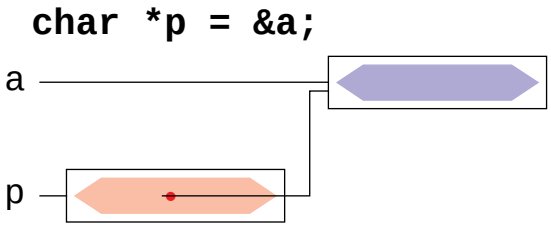
- **Konstante:**
  Bezeichnung für einen Wert



- **Variable:**
  Bezeichnung eines Datenobjekts



- **Zeiger-Variable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt



2 Überblick

- Eine Zeigervariable ***(pointer)*** enthält als Wert die Adresse einer anderen Variablen
  - ➥ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen
- Daraus resultiert die große Bedeutung von Zeigern in C
  - ➥ Funktionen können ihre Argumente verändern ***(call-by-reference)***
  - ➥ dynamische Speicherverwaltung
  - ➥ effizientere Programme
- Aber auch Nachteile!
  - ➥ Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variable zugreifen?)
  - ➥ häufigste Fehlerquelle bei C-Programmen

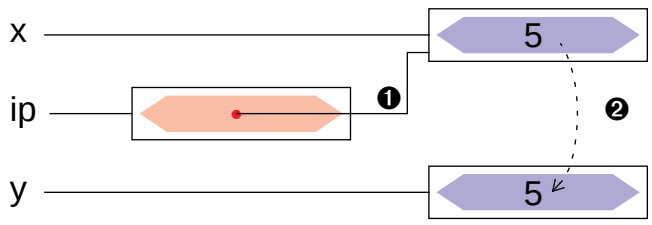
3 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

- **Beispiele**

```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



4 Adressoperatoren

- Adressoperator **&**

  &x der unäre Adress-Operator liefert eine Referenz auf den Inhalt der Variablen (des Objekts) **x**

- Verweisoperator ***

  x** der unäre Verweisoperator ** ermöglicht den Zugriff auf den Inhalt der Variablen (des Objekts), auf die der Zeiger **x** verweist

5 Zeiger als Funktionsargumente

- Parameter werden in C *by-value* übergeben
- die aufgerufene Funktion kann den aktuellen Parameter beim Aufrufer nicht verändern
- auch Zeiger werden *by-value* übergeben, d. h. die Funktion erhält lediglich eine Kopie des Adressverweises
- über diesen Verweis kann die Funktion jedoch mit Hilfe des ***-Operators auf die zugehörige Variable zugreifen und sie verändern

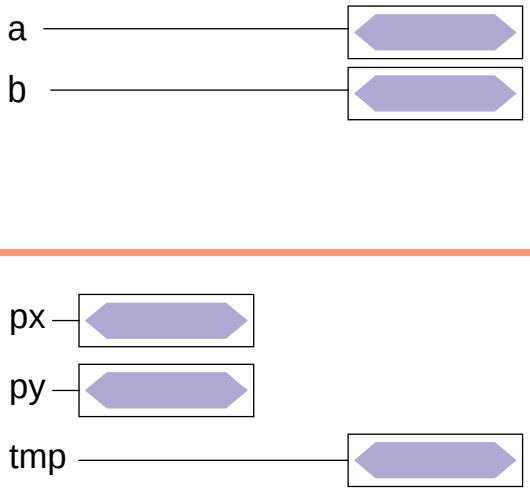
  ➥ *call-by-reference*

5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```

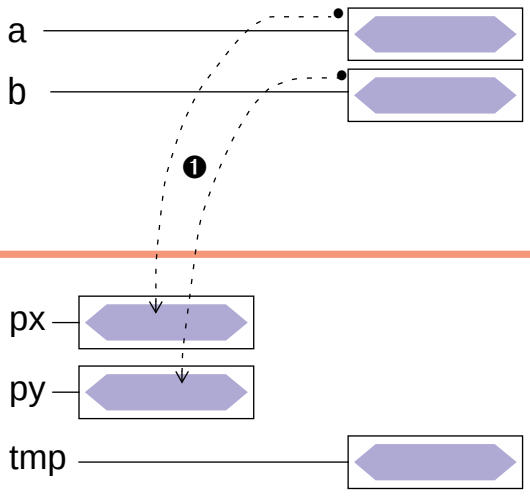


5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b); ❶
    ...
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```

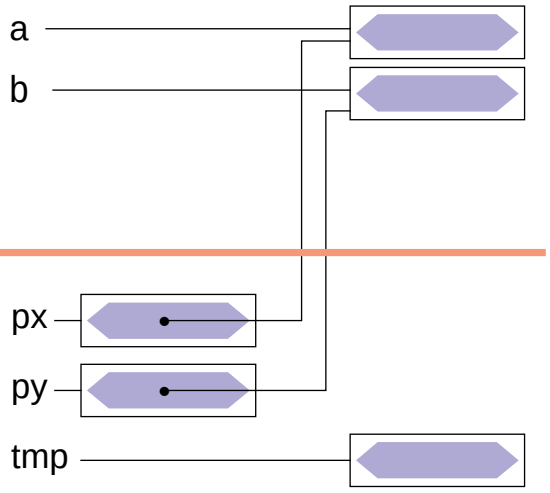


5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}
```

```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;

}
```

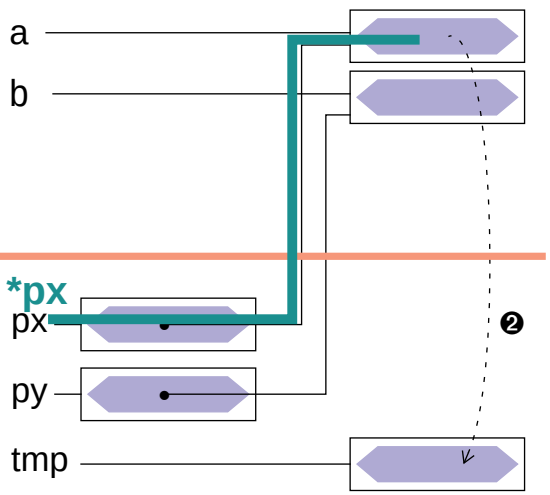


5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}
```

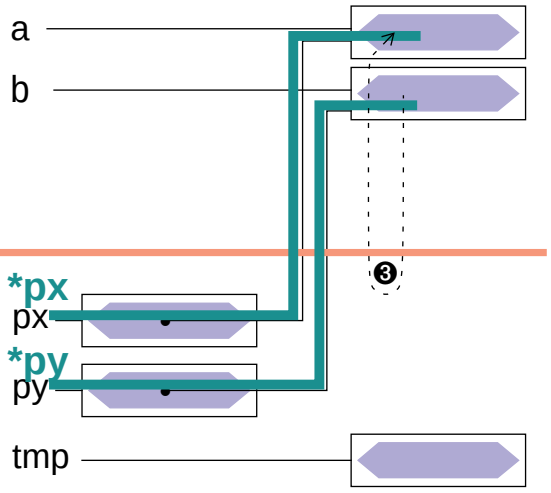
```
void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py;
    *py = tmp;

}
```



5 Zeiger als Funktionsargumente (2)

- Beispiel:

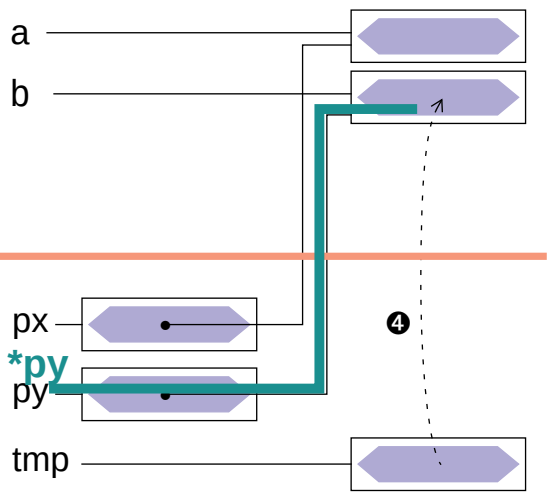
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py; ❸
    *py = tmp;

}
```



5 Zeiger als Funktionsargumente (2)

- Beispiel:

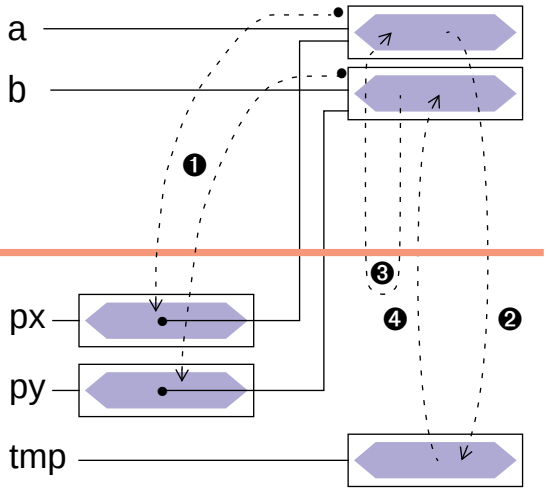
```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b);
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp; ❹

}
```



5 Zeiger als Funktionsargumente (2)

- Beispiel:

```
void swap (int *, int *);
int main() {
    int a, b;
    ...
    swap(&a, &b); ❶
    ...
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



6 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
  - ➤ Zeigerarithmetik berücksichtigt Strukturgröße
- Beispiele

```
struct student stud1;
struct student *pstud;
pstud = &stud1;            /* ⇒ pstud → stud1 */
```

- Besondere Bedeutung zum Aufbau verketteter Strukturen

6 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
  - ➤ `*`-Operator liefert die Struktur
  - ➤ `.`-Operator zum Zugriff auf Komponente
  - ➤ Operatorenvorrang beachten

➥
```
(*pstud).best = 'n';
```
unleserlich!

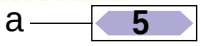
- Syntaktische Verschönerung

➥ `->`-Operator

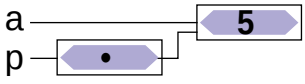
```
pstud->best = 'n';
```


7 Zusammenfassung

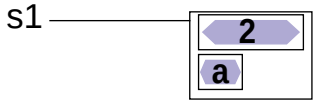
- Variable

```
int a;
```


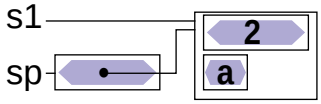
- Zeiger

```
int *p = &a;
```


- Struktur

```
struct s{int a; char c;};
struct s s1 = {2, 'a'};
```


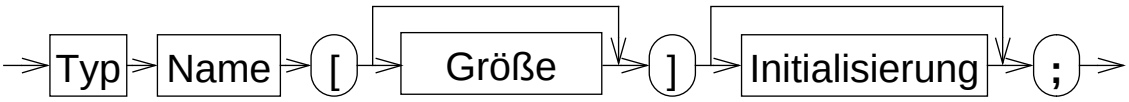
- Zeiger auf Struktur

```
struct s *sp = &s1;
```


Felder

1 Eindimensionale Felder

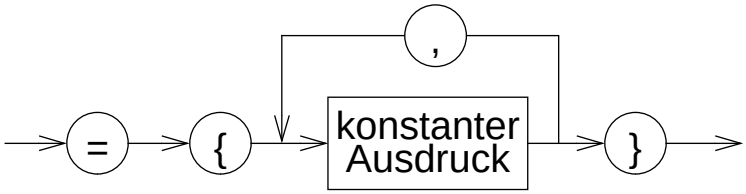
- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefasst werden
- bei der Definition wird die Größe des Felds angegeben
  - ➤ Größe muss eine Konstante sein
  - ➤ ab C99 bei lokalen Feldern auch zur Laufzeit berechnete Werte zulässig
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes



- Beispiele:

```
int x[5];
double f[20];
```


2 Initialisierung eines Feldes



- Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'O', 't', 't', 'o', '\0'};
```

- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};
char name[] = {'O', 't', 't', 'o', '\0'};
```

- werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

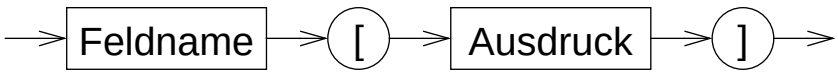
3 Initialisierung eines Feldes (2)

- Felder des Typs ***char*** können auch durch String-Konstanten initialisiert werden

```
char name1[5] = "Otto";
char name2[] = "Otto";
```


4 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= Wert(Ausdruck) < Feldgröße

- Beispiele:

```
prim[0] == 2
prim[1] == 3
name[1] == 't'
name[4] == '\0'
```