

2 Struktur eines C-Programms

globale Variablendefinitionen

Funktionen

```
int main(int argc, char *argv[]) {
    Variablendefinitionen
    Anweisungen
}
```

■ Beispiel

```
int main(int argc, char *argv[]) {
    printf("Hello World!");
}
```

■ Übersetzen mit dem C-Compiler:

```
cc -o hello hello.c
```

■ Ausführen durch Aufruf von `hello`

3 Datentypen und Variablen

■ Datentypen legen fest:

- ◆ Repräsentation der Werte im Rechner
- ◆ Größe des Speicherplatzes für Variablen
- ◆ erlaubte Operationen

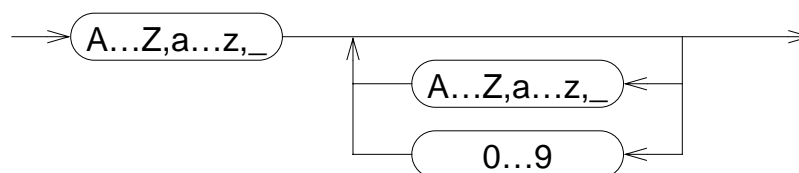
3.1 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

char	Zeichen (im ASCII-Code dargestellt, 8 Bit)
int	ganze Zahl (16 oder 32 Bit)
float	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
double	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
void	ohne Wert

3.2 Variablen

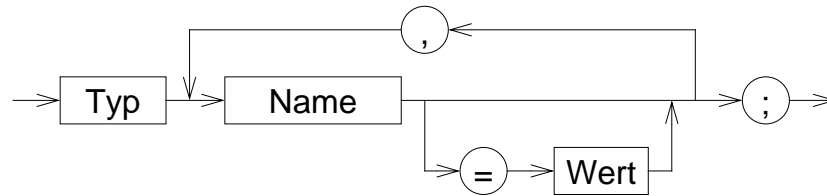
- Variablen besitzen
 - ◆ **Namen** (Bezeichner)
 - ◆ Typ
 - ◆ zugeordneten Speicherbereich für einen Wert des Typs
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
 - ◆ **Lebensdauer**
- Variablenname:



(Buchstabe oder `_`,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder `_`)

3.2 Variablen (2)

- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich



3.2 Variablen (3)

- Variablen-Definition: Beispiele

```

int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
char trennzeichen;
  
```

- Position von Variablendefinitionen im Programm:
 - ◆ nach jeder "{"
 - ◆ außerhalb von Funktionen
- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar

3.3 Strukturierte Datentypen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit

```
struct person {
    char *name;
    int alter;
};
```

- Variablen-Definition

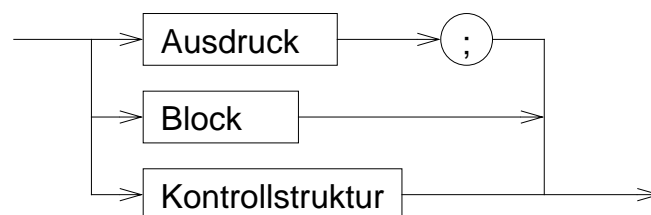
```
struct person p1;
```

- Zugriff auf Elemente der Struktur

```
p1.name = "Hans";
```

4 Anweisungen

Anweisung:



- Beispiele:

◆ **a = b + c;**

◆ **{ a = b + c; x = 5; }**

◆ **if (x == 5) a = 3;**

4.1 Blöcke

- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen

```

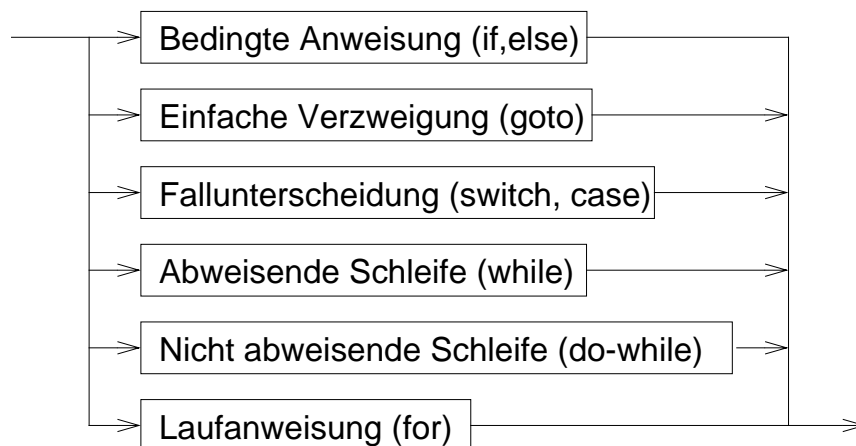
main()
{
    int x, y, z;
    x = 1;
    {
        int a, b, c;
        a = x+1;
        {
            int a, x;
            x = 2;
            a = 3;
        }
        /* a: 2, x: 1 */
    }
}

```

4.2 Kontrollstrukturen

- Kontrolle des Programmablaufs in Abhängigkeit vom Ergebnis von Ausdrücken

Kontrollstruktur:



4.2.1 Schleifensteuerung

■ break

- ◆ bricht die umgebende Schleife bzw. **switch**-Anweisung ab

```
char c;

do {
    if ( (c = getchar()) == EOF ) break;
    putchar(c);
}
while ( c != '\n' );
```

■ continue

- ◆ bricht den aktuellen **Schleifendurchlauf** ab
- ◆ setzt das Programm mit der Ausführung des Schleifenkopfes fort

5 Funktionen

■ Funktion =

Programmstück (Block), das mit einem **Namen** versehen ist und dem zum Ablauf **Parameter** übergeben werden können

■ Funktionen sind die elementaren Bausteine für Programme

- ➔ verringern die **Komplexität** durch Zerteilen umfangreicher, schwer überblickbarer Aufgaben in kleine Komponenten
- ➔ erlauben die **Wiederverwendung** von Programmkomponenten
- ➔ verbergen **Implementierungsdetails** vor anderen Programmteilen (**Black-Box-Prinzip**)

5.1 Beispiel Sinusberechnung

```
#include <stdio.h>
#include <math.h>

double sinus (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);
}
```

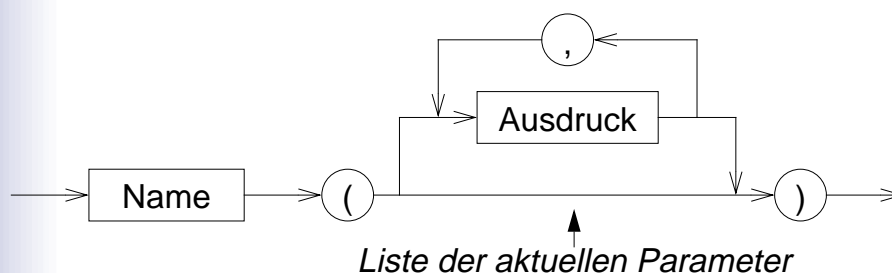
```
int main(int argc, char *argv[])
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
        wert, sinus(wert));
}
```

- beliebige Verwendung von **sinus** in Ausdrücken:

```
y = exp(tau*t) * sinus(f*t);
```

5.2 Funktionsaufruf (2)



- Die Ausdrücke in der Parameterliste werden ausgewertet, **bevor** in die Funktion gesprungen wird
➔ **aktuelle Parameter**
- Anzahl und Typen der Ausdrücke in der Liste der aktuellen Parameter müssen mit denen der formalen Parameter in der Funktionsdefinition übereinstimmen
- Die Auswertungsreihenfolge der Parameterausdrücke ist **nicht** festgelegt

5.3 Funktionen — Regeln

- Funktionen werden global definiert
- `main()` ist eine normale Funktion, die aber automatisch als erste beim Programmstart aufgerufen wird
- rekursive Funktionsaufrufe sind zulässig
 - ➔ eine Funktion darf sich selbst aufrufen (z. B. zur Fakultätsberechnung)

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

5.3 Funktionen — Regeln (2)

- Funktionen müssen **deklariert** sein, bevor sie aufgerufen werden
 - = Rückgabetyt und Parametertypen müssen bekannt sein
 - ◆ durch eine Funktionsdefinition ist die Funktion automatisch auch deklariert
- wurde eine verwendete Funktion vor ihrer Verwendung nicht deklariert, wird automatisch angenommen
 - Funktionswert vom Typ `int`
 - 1 Parameter vom Typ `int`
 - ➔ **schlechter Programmierstil → fehleranfällig**

5.3 Funktionsdeklaration

- soll eine Funktion vor ihrer Definition verwendet werden, kann sie durch eine **Deklaration** bekannt gemacht werden (Prototyp)

◆ Syntax:

```
Typ Name ( Liste formaler Parameter );
```

- Parameternamen können weggelassen werden, die Parametertypen müssen aber angegeben werden!

◆ Beispiel:

```
double sinus(double);
```

5.3 Funktionsdeklarationen — Beispiel

```
#include <stdio.h>
#include <math.h>

double sinus(double);
/* oder: double sinus(double x); */

main()
{
    double wert;

    printf("Berechnung des Sinus von ");
    scanf("%lf", &wert);
    printf("sin(%lf) = %lf\n",
           wert, sinus(wert));
}
```

```
double sinus (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);
}
```

5.4 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
 - call by value (wird in C verwendet)
 - call by reference (wird in C **nicht** verwendet)
- call-by-value: Es wird eine Kopie des aktuellen Parameters an die Funktion übergeben
 - die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - die Funktion kann den Wert des formalen Parameters (also die Kopie!) ändern, ohne daß dies Auswirkungen auf den Wert des aktuellen Parameters beim Aufrufer hat
 - die Funktion kann über einen Parameter dem Aufrufer keine Ergebnisse mitteilen

6 C-Preprozessor

- bevor eine C-Quelle dem C-Compiler übergeben wird, wird sie durch einen Makro-Preprozessor bearbeitet
- Anweisungen an den Preprozessor werden durch ein #-Zeichen am Anfang der Zeile gekennzeichnet
- die Syntax von Preprozessoranweisungen ist unabhängig vom Rest der Sprache
- Preprozessoranweisungen werden nicht durch ; abgeschlossen!
- wichtigste Funktionen:

#define	Definition von Makros
#include	Einfügen von anderen Dateien

6.1 Makrodefinitionen

- Makros ermöglichen einfache textuelle Ersetzungen (parametrierbare Makros werden später behandelt)
- ein Makro wird durch die **#define**-Anweisung definiert

- Syntax:

```
#define Makroname Ersatztext
```

- eine Makrodefinition bewirkt, daß der Preprozessor im nachfolgenden Text der C-Quelle alle Vorkommen von **Makroname** durch **Ersatztext** ersetzt

- Beispiel:

```
#define EOF -1
```

6.2 Einfügen von Dateien

- **#include** fügt den Inhalt einer anderen Datei in eine C-Quelldatei ein

- Syntax:

```
#include < Dateiname >  
oder  
#include "Dateiname"
```

- mit **#include** werden *Header*-Dateien mit Daten, die für mehrere Quelldateien benötigt werden, einkopiert
 - Deklaration von Funktionen, Strukturen, externen Variablen
 - Definition von Makros
- wird **Dateiname** durch < > geklammert, wird eine **Standard-Header-Datei** einkopiert
- wird **Dateiname** durch " " geklammert, wird eine Header-Datei des Benutzers einkopiert (vereinfacht dargestellt!)

7.1 Einordnung

■ **Konstante:**

Bezeichnung für einen Wert

'a' ≡ 0110 0001

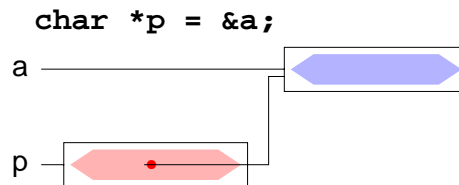
■ **Variable:**

Bezeichnung eines Datenobjekts



■ **Zeiger-Variable (Pointer):**

Bezeichnung einer Referenz auf ein Datenobjekt



7.2 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

1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

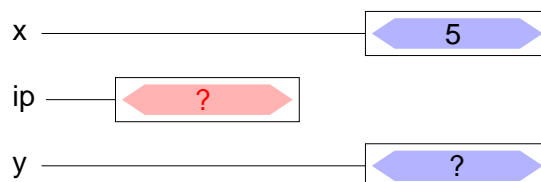
1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;  
int *ip;  
int y;
```



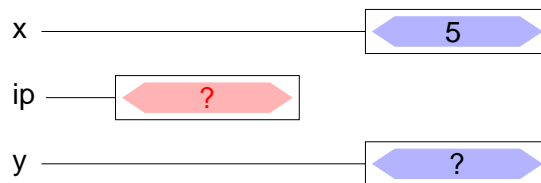
1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ❶
```



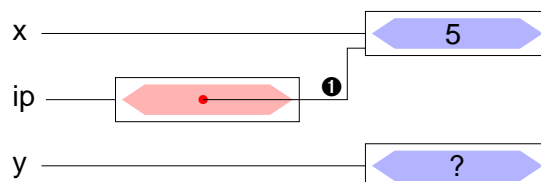
1 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

7.2.2 Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ❶
```



1 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

7.2.2 Beispiele

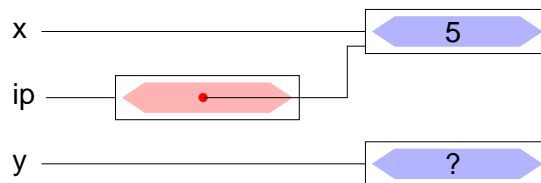
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



1 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

7.2.2 Beispiele

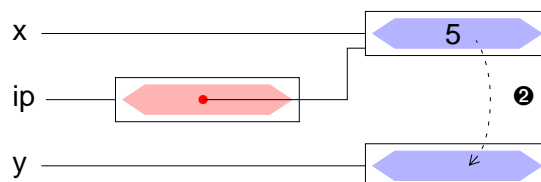
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



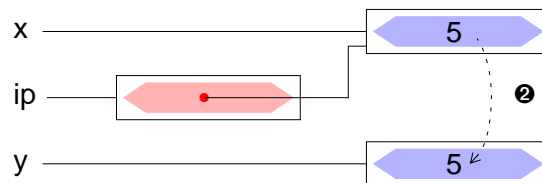
1 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

7.2.2 Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ❶
y = *ip; ❷
```



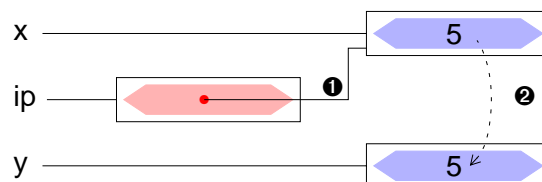
1 Definition von Zeigervariablen

■ Syntax:

Typ *Name ;

7.2.2 Beispiele

```
int x = 5;
int *ip;
int y;
ip = &x; ❶
y = *ip; ❷
```



7.3 Adreßoperatoren

7.3.1 Adreßoperator &

&x der unäre Adreß-Operator liefert die Adresse
der Variablen (des Objekts) **x**

7.3.2 Verweisoperator *

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

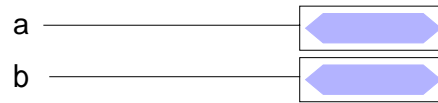
7.4 Zeiger als Funktionsargumente

- 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*

7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

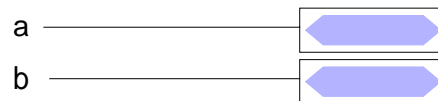
```
main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b);
}
```



7.4 ... Zeiger als Funktionsargumente (2)

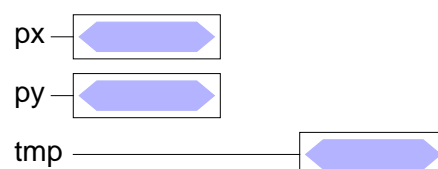
■ Beispiel:

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



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

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```



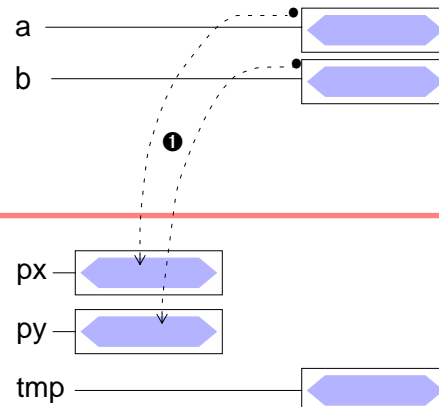
7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```



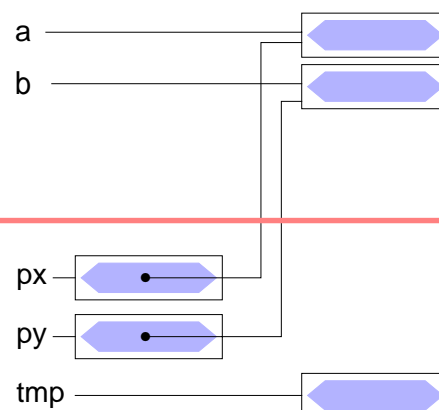
7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

    tmp = *px;
    *px = *py;
    *py = tmp;
}
```



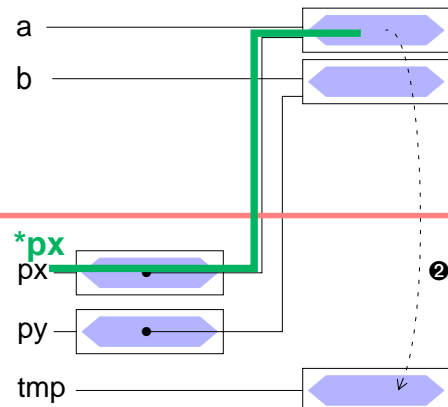
7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

```
main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}
```

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

    tmp = *px; ②
    *px = *py;
    *py = tmp;
}
```



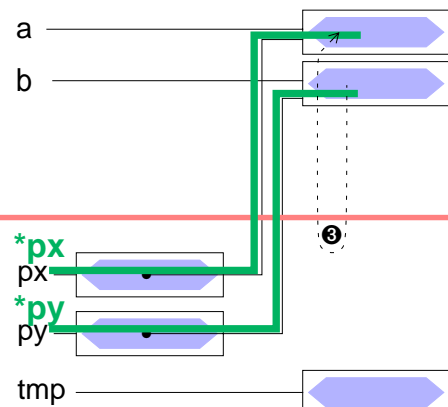
7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

```
main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}
```

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

    tmp = *px;
    *px = *py; ③
    *py = tmp;
}
```



7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

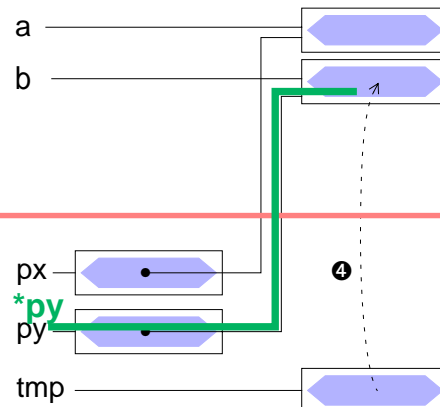
```

main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}

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

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④
}

```



7.4 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

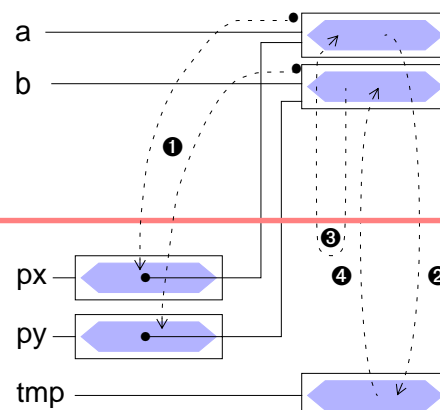
```

main(void) {
    int a, b;
    void swap (int *, int *);
    ...
    swap(&a, &b); ①
}

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

    tmp = *px; ②
    *px = *py; ③
    *py = tmp; ④
}

```



7.5 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
 - Adresse einer Struktur mit &-Operator zu bestimmen
 - Name eines Feldes von Strukturen = Zeiger auf erste Struktur im Feld
 - Zeigerarithmetik berücksichtigt Strukturgröße

- Beispiele

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

- Besondere Bedeutung zum Aufbau verketteter Strukturen

7.5 Zeiger auf Strukturen

- 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';
```

7.6 Zusammenfassung

■ Variable

```
int a;
a — 5
```

■ Zeiger

```
int *p = &a;
a — 5
p — •
```

■ Struktur

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

s1 —

■ Zeiger auf Struktur

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

s1 —

sp — •

7.7 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln

➤ z. B. zum Anfordern von Speicher für ein Feld (→ malloc)

■ Syntax:

sizeof x liefert die Größe des Objekts x in Bytes

sizeof (Typ) liefert die Größe eines Objekts vom Typ Typ in Bytes

- Das Ergebnis ist vom Typ **size_t** (entspricht meist **int**)
(**#include <stddef.h>!**)

■ Beispiel:

```
int a; size_t b;
b = sizeof a;      /* => b = 2 oder b = 4 */
b = sizeof(double) /* => b = 8 */
```

7.8 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```

$d = (i * f);$

→float
→double

- In manchen Fällen wird eine explizite Typumwandlung benötigt (vor allem zur Umwandlung von Zeigern)

◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a      (int *) a
(float) b    (char *) a
```

8 Speicherverwaltung

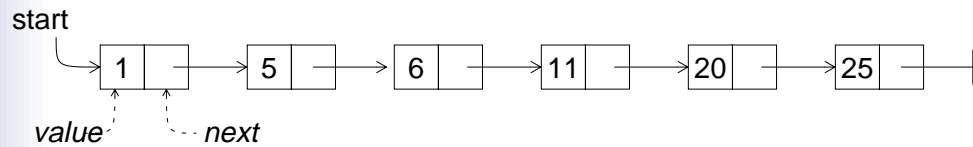
- `void *malloc(size_t size)`: Reservieren eines Speicherbereiches
- `void free(void *ptr)`: Freigeben eines reservierten Bereiches

```
struct person *p1 = (struct person *) malloc(sizeof(struct person));
if (p1 == NULL) {
    ...
}
...
free(p1);
```

- malloc-Prototyp ist in `stdlib.h` definiert (`#include <stdlib.h>`)

9 1. Aufgabe

9.1 Warteschlange als verkettete Liste



■ Strukturdefinition:

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

■ Funktionen:

- ◆ `void append_element(int)`: Anfügen eines Elements ans Listenende
- ◆ `int remove_element()`: Entnehmen eines Elements vom Listenanfang