

A Übungen zu Systemprogrammierung I

A.1 Organisatorisches

- Folien der Übungen im WWW
- URL zur Übung
 - ◆ http://www4.informatik.uni-erlangen.de/Lehre/WS03/V_SP1/Uebung/
 - ◆ hier findet man Termine, Aufgaben, Folien zum Ausdrucken und evt. Zusatzinformationen
- Schein / Prüfung
 - ◆ Schein auf erfolgreiche Bearbeitung der Übungsaufgaben (mind. 60%)
 - ◆ Inf. 5. Semester, Computerlinguistik, Mathe: zusätzlich mündl. Prüfung
 - ◆ Prüfungsklausur für Inf. 3. Semester, CE und Wirtschaftsinformatik im Prüfungszeitraum März/April

A.2 Übungsinhalt:

A.2 Übungsinhalt:

- Programmieren einfacher Übungsaufgaben zum Üben des Vorlesungsstoffs
 - ◆ Tafelübungen
 - Vorbereitung und Besprechung der Übungsaufgaben
 - Diskussion von Problemen
 - ◆ Rechnerübungen
 - selbständige Bearbeitung der Programmieraufgaben an den CIP-Workstations der Informatik
 - Übungsleiter steht für Fragen zur Verfügung

B Kurzeinführung in die Programmiersprache C

- Literatur zur C-Programmierung:
 - ◆ Darnell, Margolis. *C: A Software Engineering Approach*. Springer 1991
 - ◆ Kernighan, Ritchie. *The C Programming Language*. Prentice-Hall 1988

B.1 Überblick

- | | |
|---|---|
| ◆ Struktur eines C-Programms | ◆ Speicherverwaltung |
| ◆ Datentypen und Variablen | ◆ 1. Aufgabe |
| ◆ Anweisungen | ◆ Felder |
| ◆ Funktionen | ◆ Strukturen |
| ◆ C-Preprozessor | ◆ Ein- /Ausgabe |
| ◆ Zeiger(-Variablen) | ◆ Fehlerbehandlung |
| ◆ sizeof-Operator | ◆ Dynamische Speicherverwaltung, Teil 2 |
| ◆ Explizite Typumwandlung — Cast-Operator | ◆ Portable Programme |

B.2 Struktur eines C-Programms

B.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`

B.3 Datentypen und Variablen

- Datentypen legen fest:
 - ◆ Repräsentation der Werte im Rechner
 - ◆ Größe des Speicherplatzes für Variablen
 - ◆ erlaubte Operationen

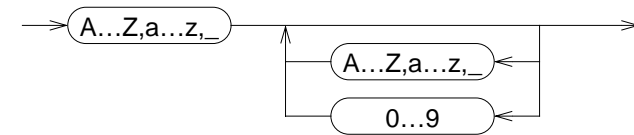
1 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

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

2 Variablen

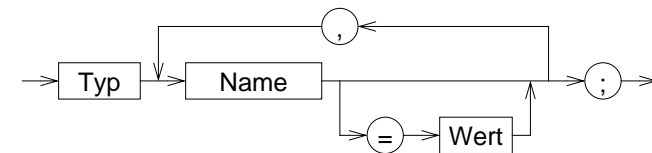
- Variablen besitzen
 - ◆ **Namen** (Bezeichner)
 - ◆ Typ
 - ◆ zugeordneten Speicherbereich für einen Wert des Typs
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
 - ◆ **Lebensdauer**
- Variablenname:



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

2 Variablen (2)

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



2 Variablen (3)

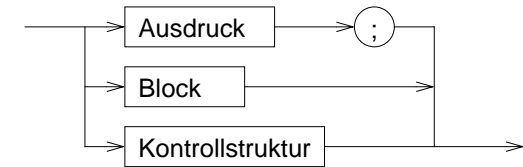
- Variablen-Definition: Beispiele

```
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

B.4 Anweisungen

Anweisung:



1 Ausdrücke - Beispiele

- ◆ $a = b + c;$
- ◆ $\{ a = b + c; x = 5; \}$
- ◆ $\text{if } (x == 5) a = 3;$

3 Strukturierte Datentypen (structs)

- Zusammenfassen mehrerer Daten zu einer Einheit

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

- Variablen-Definition

```
struct person pl;
```

- Zugriff auf Elemente der Struktur

```
pl.name = "Hans";
```

2 Blöcke

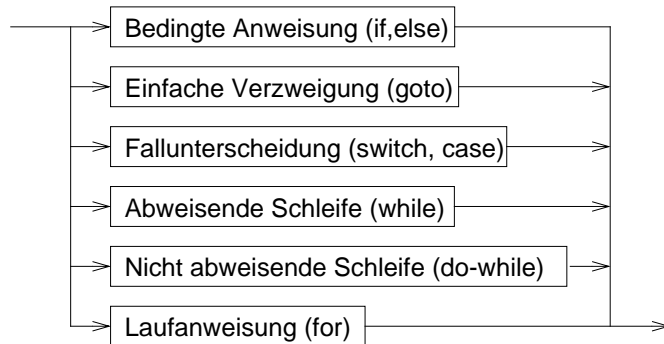
- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen

```
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 */
    }
}
```

3 Kontrollstrukturen

- Kontrolle des Programmablaufs in Abhängigkeit vom Ergebnis von Ausdrücken

Kontrollstruktur:



4 Kontrollstrukturen — 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

B.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**)

1 Funktionsdefinition

- Schnittstelle (Ergebnistyp, Name, Parameter)
- + Implementierung

2 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);
}

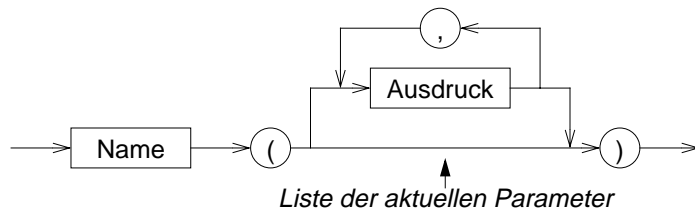
int main(int argc, char *argv[])
{
    double wert;

    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);
```

3 Funktionsaufruf



- 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

4 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) );
}
```

4 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 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);
```

6 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);
}
```

B.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

7 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

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
```

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!)

SP1-Ü

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

SP1-Ü

B.7 Zeiger(-Variablen)

1 Einordnung

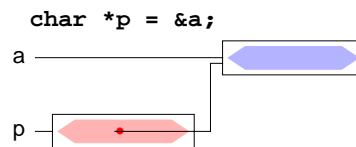
- **Konstante:**
Bezeichnung für einen Wert

'a' ≡ 0110 0001

- **Variable:**
Bezeichnung eines Datenobjekts



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



SP1-Ü

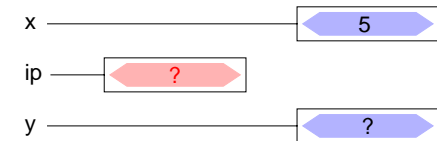
3 Definition von Zeigervariablen

- Syntax:

Typ *Name ;

- ▲ **Beispiele**

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



SP1-Ü

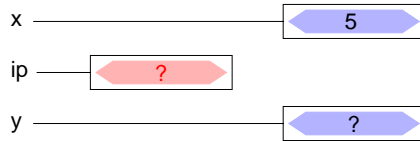
3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

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



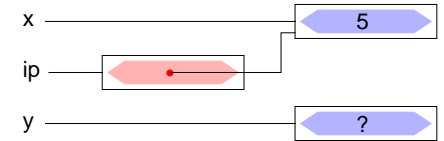
3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

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



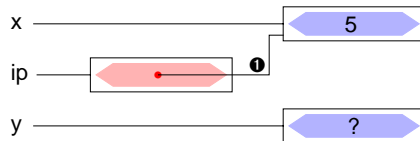
3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

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



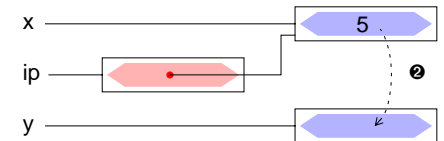
3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

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



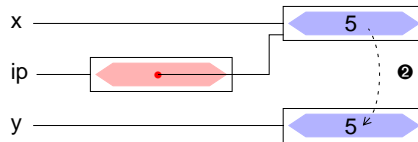
3 Definition von Zeigervariablen

■ 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

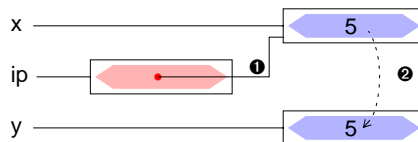
3 Definition von Zeigervariablen

■ Syntax:

```
Typ *Name ;
```

▲ Beispiele

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



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

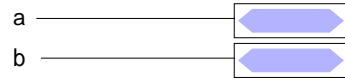
■ über diesen Verweis kann die Funktion jedoch mit Hilfe des *****-Operators auf die zugehörige Variable zugreifen und sie verändern

↳ *call-by-reference*

5 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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



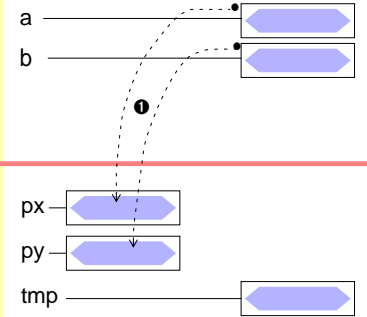
5 ... Zeiger als Funktionsargumente (2)

■ 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;
}
```



SP1-Ü

SP1-Ü

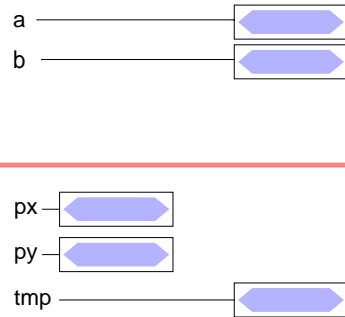
5 ... Zeiger als Funktionsargumente (2)

■ 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;
}
```



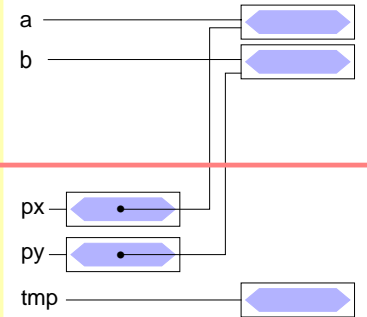
5 ... Zeiger als Funktionsargumente (2)

■ 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;
}
```



SP1-Ü

SP1-Ü

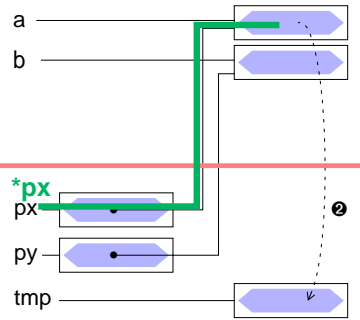
5 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

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



SP1-Ü

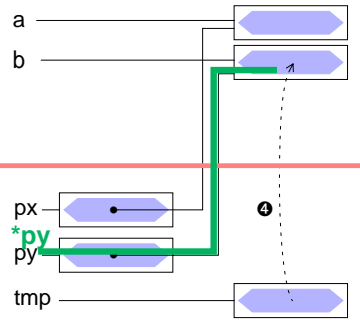
5 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

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



SP1-Ü

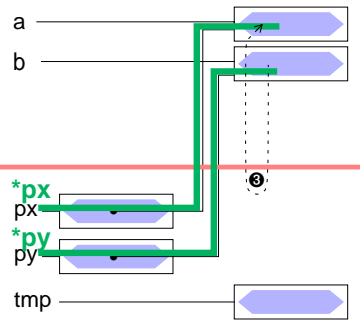
5 ... Zeiger als Funktionsargumente (2)

■ 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; ❸
  *py = tmp;
}
```



SP1-Ü

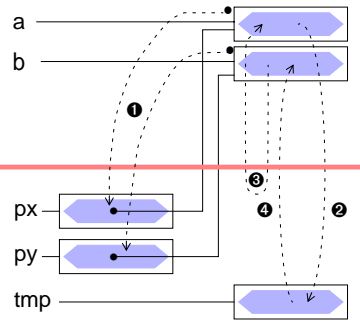
5 ... Zeiger als Funktionsargumente (2)

■ Beispiel:

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

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

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



SP1-Ü

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

SP1-Ü

7 Zusammenfassung

■ Variable

```
int a;
a — 5
```

■ Zeiger

```
int *p = &a;
a — 5
p — • —> a
```

■ Struktur

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

■ Zeiger auf Struktur

```
struct s *sp = &s1;
s1 — 2 — a
sp — • —> s1
```

SP1-Ü

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';
```

SP1-Ü

B.8 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 */
```

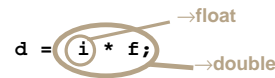
SP1-Ü

B.9 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;
```



- 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
```

SP1-Ü

B.10 Speicherverwaltung

- void *malloc(size_t size): Reservieren eines Speicherbereiches

- void free(void *ptr): Freigeben eines reservierten Bereiches

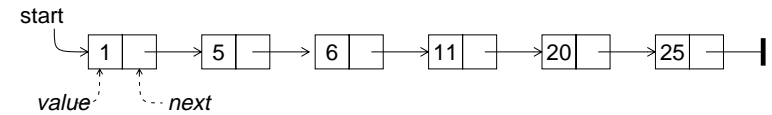
```
struct person *p1 = (struct person *) malloc(sizeof(struct person));
if (p1 == NULL) {
    perror("malloc person p1");
    ...
}
...
free(p1);
```

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

SP1-Ü

1. Aufgabe

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

SP1-Ü

B.11 Felder

1 Eindimensionale Felder

- eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefaßt werden
- bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!
- der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null
- Definition eines Feldes

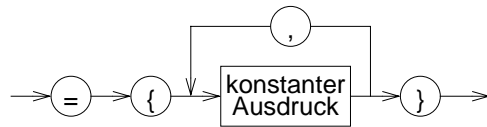


- Beispiele:

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

SP1-Ü

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

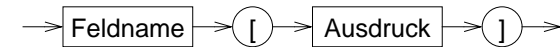
3 ... Initialisierung eines Feldes (2)

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

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

4 Zugriffe auf Feldelemente

- Indizierung:



wobei: $0 \leq \text{Wert}(\text{Ausdruck}) < \text{Feldgröße}$

- Beispiele:

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

5 Mehrdimensionale Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
- Definition eines mehrdimensionalen Feldes

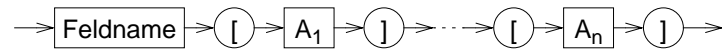


- Beispiel:

```
int matrix[4][4];
```

6 Zugriffe auf Feldelemente bei mehrdim. Feldern

- Indizierung:



wobei: $0 \leq A_i < \text{Größe der Dimension } i \text{ des Feldes}$
 $n = \text{Anzahl der Dimensionen des Feldes}$

- Beispiel:

```
int feld[5][8];
feld[2][3] = 10;
```

- ist äquivalent zu:

```
int feld[5][8];
int *f1;
f1 = (int*)feld;
f1[2*8 + 3] = 10;
```

7 Initialisierung eines mehrdimensionalen Feldes

- ein mehrdimensionales Feld kann - wie ein eindimensionales Feld - durch eine Liste von konstanten Werten, die durch Komma getrennt sind, initialisiert werden
- wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Größe des Feldes
- Beispiel:

```
int feld[3][4] = {
  { 1, 3, 5, 7 }, /* feld[0][0-3] */
  { 2, 4, 6   }, /* feld[1][0-2] */
};
```

`feld[1][3]` und `feld[2][0-3]` werden in dem Beispiel mit 0 initialisiert!

B.12 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht by-value** übergeben werden
- wird einer Funktion der Feldname als Parameter übergeben, kann sie in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
 - die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
 - ggf. ist die Feldgröße über einen weiteren `int`-Parameter der Funktion explizit mitzuteilen
 - die Länge von Zeichenketten in `char`-Feldern kann normalerweise durch Suche nach dem `\0`-Zeichen bestimmt werden
- wird ein Feldparameter als `const` deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden

1 Beispiele

- Bestimmung der Länge einer Zeichenkette (*String*)

```
int strlen(const char string[])
{
  int i=0;
  while (string[i] != '\0') ++i;
  return(i);
}
```

1 Beispiele (2)

- Konkateniere Strings

```
void strcat(char to[], const char from[])
{
    int i=0, j=0;
    while (to[i] != '\0') i++;
    while ( (to[i++] = from[j++]) != '\0' )
        ;
}
```

- Funktionsaufruf mit Feld-Parametern

► als aktueller Parameter beim Funktionsaufruf wird einfach der Feldname angegeben

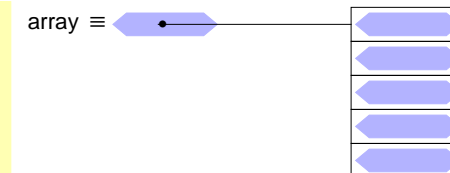
```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2); /* → s1= "text1text2" */
strcat(s1, "text3"); /* → s1= "text1text2text3" */
```

SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
```



SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

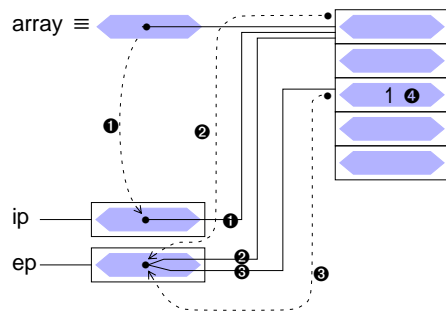
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



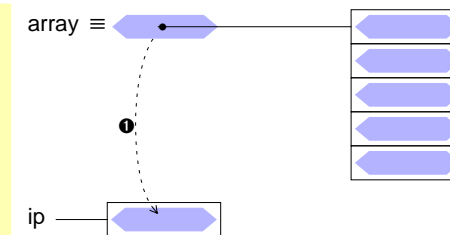
SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];

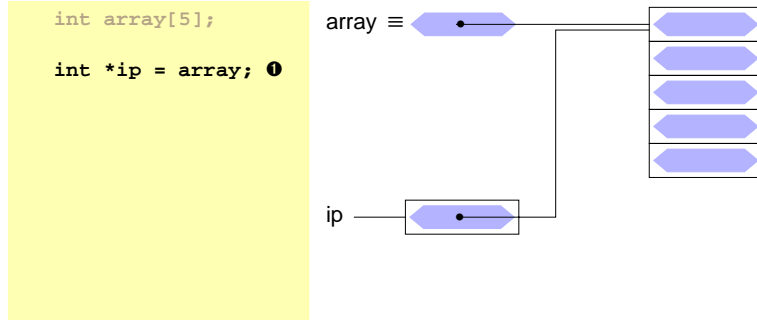
int *ip = array; ❶
```



SP1-Ü

B.13 Zeiger und Felder

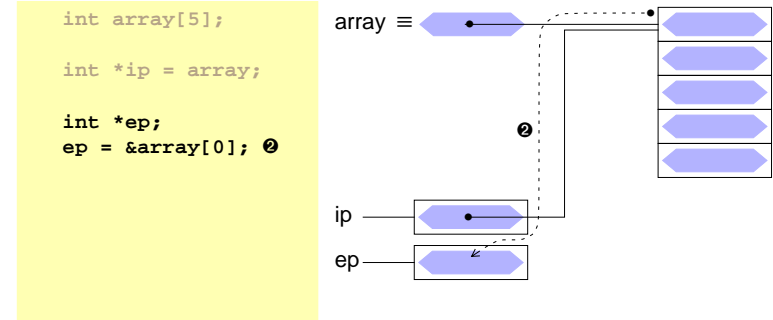
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



SP1-Ü

B.13 Zeiger und Felder

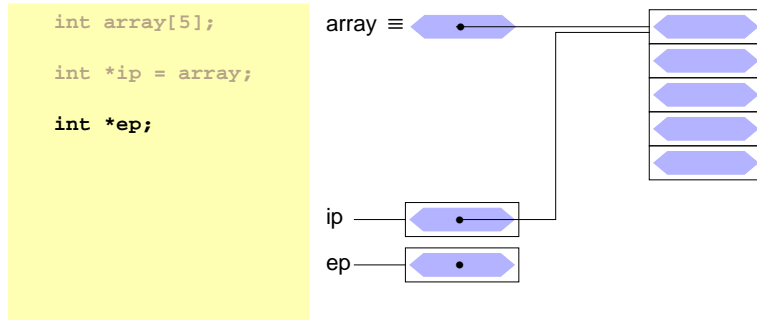
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



SP1-Ü

B.13 Zeiger und Felder

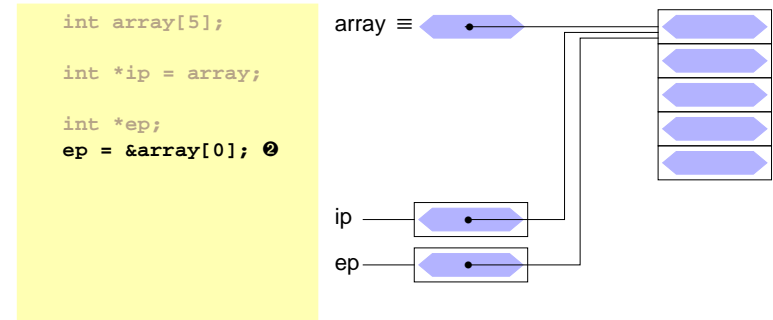
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



SP1-Ü

B.13 Zeiger und Felder

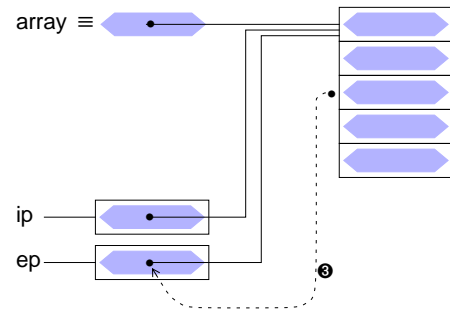
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2]; ③
```



SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

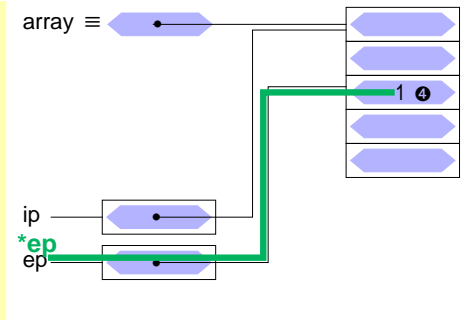
```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2];

*ep = 1; ④
```



SP1-Ü

B.13 Zeiger und Felder

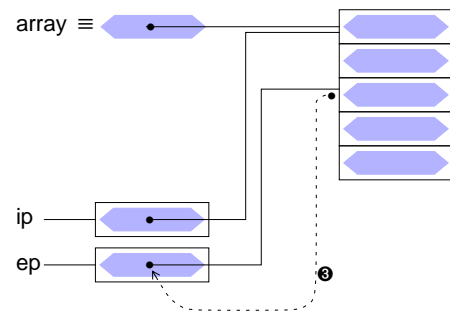
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];

int *ip = array;

int *ep;
ep = &array[0];

ep = &array[2]; ③
```



SP1-Ü

B.13 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

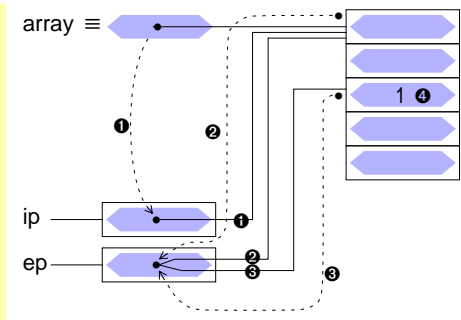
```
int array[5];

int *ip = array; ①

int *ep;
ep = &array[0]; ②

ep = &array[2]; ③

*ep = 1; ④
```

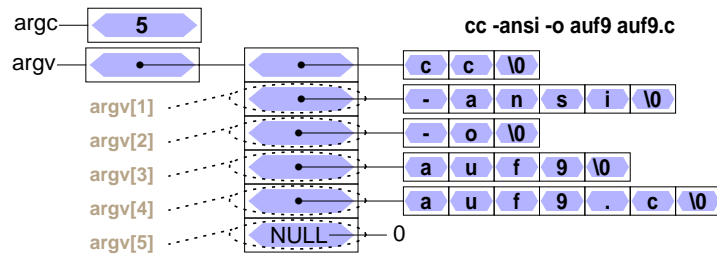


SP1-Ü

B.14 Kommandozeilenparameter

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int main (int argc, char *argv[]) {
    int i;
    for ( i=1; i<argc; i++) {
        printf("%s%c", argv[i],
            (i < argc-1) ? ' ': '\n' );
    }
    ...
}
```



1 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden

- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! Vorsicht

bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,

bei der Initialisierung jedoch nur durch die Postion

- potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

B.15 Strukturen

- Initialisierung
- Strukturen als Funktionsparameter
- Felder von Strukturen
- Zeiger auf Strukturen

2 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
 - Übergabesemantik: **call by value**
 - Funktion erhält eine Kopie der Struktur
 - auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
 - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
 - Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```

3 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden

- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    ...
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnur < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```

5 Zusammenfassung

- Variable

```
int a;
a — 5
```

- Zeiger

```
int *p = &a;
a — 5
p — • —> 5
```

- Feld

```
int a[3];
a ≡ • —> [ ] [ ] [ ]
```

- Feld von Zeigern

```
int *p[3];
p ≡ • —> [ • ] [ • ] [ • ]
```

- Struktur

```
struct s { int a; char c; };
struct s s1 = { 2, 'a' };
s1 —> [ 2 ] [ a ]
```

- Zeiger auf Struktur

```
struct s *sp = &s1;
s1 —> [ 2 ] [ a ]
sp —> • —> [ 2 ] [ a ]
```

- Feld von Strukturen

```
sa ≡ • —> [ ] [ ] [ ] [ ] [ ]
struct s sa[3];
```

4 Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!

- Beispiel

```
struct student gruppe8[35];
struct student *gp1, *gp2;

gp1 = gruppe8; /* gp1 zeigt auf erstes Element des Arrays */
printf("Nachname des ersten Studenten: %s", gp1->nachname);

gp2 = gp1 + 1; /* gp2 zeigt auf zweite Element des Arrays */
printf("Nachname des zweiten Studenten: %s", gp2->nachname);

printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

B.16 Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion

- ◆ Variablendef.: <Rückgabety> (*<Variablenname>) (<Parameter>);

```
int (*fptr)(int, char*);
```

```
int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }
```

```
fptr = test1;
```

```
fptr(42, "hallo");
```

```
fptr = test2;
```

```
fptr(42, "hallo");
```

B.17 Ein-/Ausgabe

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
 - Bestandteil der Standard-Funktionsbibliothek
 - einfache Programmierschnittstelle
 - effizient
 - portabel
 - betriebssystemnah
- Funktionsumfang
 - Öffnen/Schließen von Dateien
 - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
 - Formatierte Ein-/Ausgabe

1 Standard Ein-/Ausgabe

- Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:
 - ◆ `stdin` Standardeingabe
 - normalerweise mit der Tastatur verbunden
 - Dateiende (**EOF**) wird durch Eingabe von **CTRL-D** am Zeilenanfang signalisiert
 - bei Programmaufruf in der Shell auf Datei umlenkbar


```
prog <eingabedatei
```

 (bei Erreichen des Dateiendes wird **EOF** signalisiert)
 - ◆ `stdout` Standardausgabe
 - normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
 - bei Programmaufruf in der Shell auf Datei umlenkbar


```
prog >ausgabedatei
```
 - ◆ `stderr` Ausgabekanal für Fehlermeldungen
 - normalerweise ebenfalls mit Bildschirm verbunden

1 Standard Ein-/Ausgabe (2)

- Pipes
 - ◆ die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden
 - Aufruf


```
prog1 | prog2
```
- ! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar
- automatische Pufferung
 - ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen (`'\n'`) an das Programm übergeben!

2 Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen
 - Zugriff auf Dateien
- Öffnen eines E/A-Kanals
 - Funktion `fopen`:


```
#include <stdio.h>
FILE *fopen(char *name, char *mode);
```

name	Pfadname der zu öffnenden Datei
mode	Art, wie die Datei geöffnet werden soll
"r"	zum Lesen
"w"	zum Schreiben
"a"	append: Öffnen zum Schreiben am Dateiende
"rw"	zum Lesen und Schreiben
 - Ergebnis von `fopen`:
Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt
im Fehlerfall wird ein `NULL`-Zeiger geliefert

2 Öffnen und Schließen von Dateien (2)

B.17 Ein-/Ausgabe

■ Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *eingabe;

    if (argv[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1);
        /* Programm abbrechen */
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]);
        /* Fehlermeldung ausgeben */
        exit(1);
        /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
}
```

■ Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

- schließt E/A-Kanal `fp`

Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

B-Einfuehrung_in_C.fm 2003-10-29 10.55

B.83

SP1-U

3 Zeichenweise Lesen und Schreiben

B.17 Ein-/Ausgabe

■ Lesen eines einzelnen Zeichens

- ◆ von der Standardeingabe

```
int getchar( )
```

- lesen das nächste Zeichen
- geben das gelesene Zeichen als `int`-Wert zurück
- geben bei Eingabe von `CTRL-D` bzw. am Ende der Datei `EOF` als Ergebnis zurück

- ◆ von einem Dateikanal

```
int getc(FILE *fp )
```

■ Schreiben eines einzelnen Zeichens

- ◆ auf die Standardausgabe

```
int putchar(int c)
```

- schreiben das im Parameter `c` übergeben Zeichen
- geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

- ◆ auf einen Dateikanal

```
int putc(int c, FILE *fp )
```

Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

B-Einfuehrung_in_C.fm 2003-10-29 10.55

B.84

SP1-U

3 Zeichenweise Lesen und Schreiben (2)

B.17 Ein-/Ausgabe

■ Beispiel: copy-Programm, Aufruf: `copy Quelldatei Zieldatei`

```
#include <stdio.h>
// Teil 1: Aufrufargumente auswerten
int main(int argc, char *argv[]) {
    FILE *quelle, *ziel;
    int c;
    /* gerade kopiertes Zeichen */

    if (argc < 3) { /* Fehlermeldung, Abbruch */ }

    if ((quelle = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(EXIT_FAILURE); /* Programm abbrechen */
    }

    if ((ziel = fopen(argv[2], "w")) == NULL) {
        /* Fehlermeldung, Abbruch */
    }

    while ( ( c = getc(quelle) ) != EOF ) {
        putchar(c, ziel);
    }

    fclose(quelle);
    fclose(ziel);
}
```

Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

B-Einfuehrung_in_C.fm 2003-10-29 10.55

B.85

SP1-U

3 Zeilenweise Lesen und Schreiben (3)

B.17 Ein-/Ausgabe

■ Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

- liest Zeichen von Dateikanal `fp` in das Feld `s` bis entweder `n-1` Zeichen gelesen wurden oder `'\n'` oder `EOF` gelesen wurde
- `s` wird mit `'\0'` abgeschlossen (`'\n'` wird nicht entfernt)
- gibt bei `EOF` oder Fehler `NULL` zurück, sonst `s`
- für `fp` kann `stdin` eingesetzt werden, um von der Standardeingabe zu lesen

■ Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

- schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
- für `fp` kann auch `stdout` oder `stderr` eingesetzt werden
- als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

Übungen zu Systemprogrammierung I

© Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2003

B-Einfuehrung_in_C.fm 2003-10-29 10.55

B.86

SP1-U

4 Formatierte Ausgabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );
int fprintf(FILE *fp, char *format, /* Parameter */ ... );
int sprintf(char *s, char *format, /* Parameter */ ... );
int snprintf(char *s, int n, char *format, /* Parameter */ ... );
```

■ Die statt ... angegebenen Parameter werden entsprechend der Angaben im `format`-String ausgegeben

- ▶ bei `printf` auf der Standardausgabe
- ▶ bei `fprintf` auf dem Dateikanal `fp` (für `fp` kann auch `stdout` oder `stderr` eingesetzt werden)
- ▶ `sprintf` schreibt die Ausgabe in das `char`-Feld `s` (achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
- ▶ `snprintf` arbeitet analog, schreibt aber maximal nur `n` Zeichen (`n` sollte natürlich nicht größer als die Feldgröße sein)

4 Formatierte Ausgabe (2)

■ Zeichen im `format`-String können verschiedene Bedeutung haben

- ▶ normale Zeichen: werden einfach auf die Ausgabe kopiert
- ▶ Escape-Zeichen: z. B. `\n` oder `\t`, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
- ▶ Format-Anweisungen: beginnen mit `%`-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem `format`-String aufbereitet werden soll

■ Format-Anweisungen

`%d`, `%i` `int` Parameter als Dezimalzahl ausgeben

`%f` `float` Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben

`%e` `float` Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben

`%c` `char`-Parameter wird als einzelnes Zeichen ausgegeben

`%s` `char`-Feld wird ausgegeben, bis `'\0'` erreicht ist

5 Formatierte Eingabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ... );
int fscanf(FILE *fp, char *format, /* Parameter */ ... );
int sscanf(char *s, const char *format, /* Parameter */ ... );
```

■ Die Funktionen lesen Zeichen von `stdin` (`scanf`), `fp` (`fscanf`) bzw. aus dem `char`-Feld `s`.

■ `format` gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen

■ Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. `char`-Felder bei Format `%s`), in die die Resultate eingetragen werden

■ relativ komplexe Funktionalität, hier nur Kurzüberblick für Details siehe Manual-Seiten

5 Formatierte Eingabe (2)

■ *White space* (Space, Tabulator oder Newline `\n`) bildet jeweils die Grenze zwischen Daten, die interpretiert werden

- ▶ *white space* wird in beliebiger Menge einfach überlesen
- ▶ Ausnahme: bei Format-Anweisung `%c` wird auch *white space* eingelesen

■ Alle anderen Daten in der Eingabe müssen zum `format`-String passen oder die Interpretation der Eingabe wird abgebrochen

- ▶ wenn im `format`-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
- ▶ wenn im `format`-String eine Format-Anweisung (`%...`) angegeben ist, muß in der Eingabe etwas hierauf passendes auftauchen
 - ↳ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen

■ Die `scanf`-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

5 Formatierte Eingabe (3)

<code>%d</code>	int
<code>%hd</code>	short
<code>%ld</code>	long int
<code>%lld</code>	long long int
<code>%f</code>	float
<code>%lf</code>	double
<code>%Lf</code>	long double
analog auch <code>%e</code> oder <code>%g</code>	
<code>%c</code>	char
<code>%s</code>	String, wird automatisch mit <code>'\0'</code> abgeschl.

- nach `%` kann eine Zahl folgen, die die maximale Feldbreite angibt
 - `%3d` = 3 Ziffern lesen
 - `%5c` = 5 char lesen (Parameter muß dann Zeiger auf char-Feld sein)
 - ▶ `%5c` überträgt exakt 5 char (hängt aber kein `'\0'` an!)
 - ▶ `%5s` liest max. 5 char (bis white space) und hängt `'\0'` an

- Beispiele:

```
int a, b, c, d, n;
char s1[20]="XXXXXX", s2[20];
n = scanf("%d %d %3d %5c %s %d",
         &a, &b, &c, s1, s2, &d);
Eingabe: 12 1234567 sowas hmm
Ergebnis: n=5, a=12, b=12, c=345
          s1="67 soX", s2="was"
```

B.18 Dynamische Speicherverwaltung

- Erzeugen von Feldern der Länge `n`:

- ◆ mittels: `void *malloc(size_t size)`

```
struct person *personen;
personen = (struct person *)malloc(sizeof(struct person)*n);
if(personen == NULL) ...
```

- ◆ mittels: `void *calloc(size_t nelem, size_t elsize)`

```
struct person *personen;
personen = (struct person *)calloc(n, sizeof(struct person));
if(person == NULL) ...
```

- ◆ `calloc` initialisiert den Speicher mit 0

- ◆ `malloc` initialisiert den Speicher nicht

- ◆ explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(personen, 0, sizeof(struct person)*n);
```

6 Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
 - ◆ Fehlerbehandlung unumgänglich!
- Vorgehensweise:
 - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
 - ◆ Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable `errno`
- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

B.18 Dynamische Speicherverwaltung (2)

- Verlängern von Felder, die durch `malloc` bzw. `realloc` erzeugt wurden:

```
void *realloc(void *ptr, size_t size)
```

```
neu = (struct person *)realloc(personen,
                               (n+10) * sizeof(struct person));
if(neu == NULL) ...
```


B.19 Portable Programme

- 1. Verwenden der standardisierten Programmiersprache ANSI-C

- ◆ gcc-Aufrufoptionen

```
-ansi -pedantic
```

- 2. Verwenden einer standardisierten Betriebssystemschnittstelle, z.B. POSIX

- ◆ gcc-Aufrufoption

```
-D_POSIX_SOURCE
```

- ◆ oder `#define` im Programmtext

```
#define _POSIX_SOURCE
```

- Programme sollte sich mit folgenden gcc-Aufruf compilieren lassen

```
gcc -ansi -pedantic-errors -D_POSIX_SOURCE -Wall -Werror
```

2 Header-Files: ANSI und POSIX

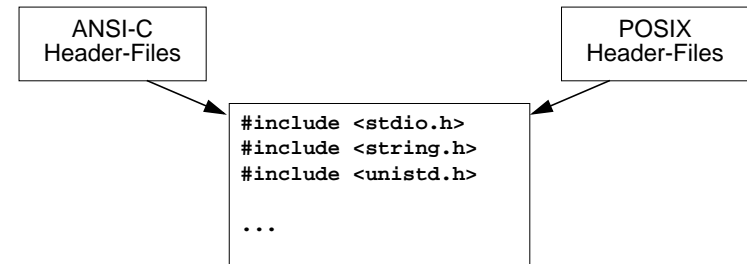
- In den Standards ANSI-C und POSIX.1 sind Header-Files definiert, mit

- ◆ Funktionsprototypen

- ◆ typedefs

- ◆ Makros und Defines

- ◆ Wenn in der Aufgabenstellung nicht anders angegeben, sollen ausschließlich diese Header-Files verwendet werden.



1 POSIX

- Standardisierung der Betriebssystemschnittstelle: Portable Operating System Interface (IEEE Standard 1003.1)

- POSIX.1 wird von verschiedenen Betriebssystemen implementiert:

- ◆ SUN Solaris 2.6
- ◆ SGI Irix 6.2/6.4
- ◆ DIGITAL Unix 4.0
- ◆ Linux (größtenteils POSIX, zertifizierte Version von Fa. Unifix)
- ◆ Windows NT (Posix Subsystem)
- ◆ ...

3 ANSI-C Header-Files

- **assert.h**: assert()-Makro
- **ctype.h**: Makros und Funktionen für Characters (z.B. tolower(), isalpha())
- **errno.h**: Fehlerauswertung (z.B. errno-Variablen)
- **float.h**: Makros für Fließkommazahlen
- **limits.h**: Enthält Definitionen für Systemschranken
- **locale.h**: Funktion setlocale()
- **math.h**: Mathematische Funktionen für double
- **setjmp.h**: Funktionen setjmp(), longjmp()
- **signal.h**: Signalbehandlung
- **stdarg.h**: Funktionen und Makros für variable Argumentlisten
- **stddef.h**: Def. von ptrdiff_t, NULL, size_t, wchar_t, offsetof, errno
- **stdio.h**: I/O Funktionen (z.B. printf(), scanf(), fgets())
- **stdlib.h**: Hilfsfunktionen (z.B. malloc(), getenv(), rand())
- **string.h**: Stringmanipulation (z.B. strcpy())
- **time.h**: Zeitmanipulation (z.B. time(), ctime(), strftime())

4 POSIX Header-Files

- **dirent.h:** opendir(), readdir(), rewinddir(), closedir()
- **fcntl.h:** open(), creat(), fcntl()
- **grp.h:** getgrgid(), getgrnam()
- **pwd.h:** getpwuid(), getpwnam()
- **setjmp.h:** sigsetjmp(), siglongjmp()
- **signal.h:** kill(), sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), sigaction, sigprocmask(), sigpending(), sigsuspend()
- **stdio.h:** ctermid(), fileno(), fdopen()
- **sys/stat.h:** umask(), mkdir(), mkfifo(), stat(), fstat(), chmod()
- **sys/times.h:** times()
- **sys/types.h:** enthält betriebssystemabhängige Typdefinitionen
- **sys/utsname.h:** uname()
- **sys/wait.h:** wait(), waitpid()
- **termios.h:** cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed(), tcgetattr(), tcsetattr(), tcsetattr(), tcsetattr(), tcflush(), tcflow()
- **time.h:** time(), tzset()
- **utime.h:** utime()
- **unistd.h:** alle POSIX-Funktionen, die nicht in den obigen Header-Files definiert sind (z.B. fork(), read())

6 XOPEN / UNIX 98 /UNIX 03

- Die Open Group
 - ◆ Eigentümer des Markenzeichens "UNIX"
 - ◆ Erstellen Spezifikationen (Systemaufruf-Schnittstellen, Programme, ...)
- Hersteller können für ihr Betriebssystem ein "Branding" erwerben
- The Single UNIX® Specification (UNIX 95)
 - ◆ enthält STREAMS, Sockets, XTI, POSIX.1, BSD und SVID Schnittstellen
 - ◆ Solaris 2.5 and 2.5.1, HP-UX 10.10, IBM AIX 4.2, Digital Unix 4
 - ◆ aktualisiert: Version 2 (UNIX 98) und Version 3 (UNIX 03)
- The Single UNIX® Specification, Version 2 (Unix98)
 - ◆ <http://www.opengroup.org/onlinepubs/007908799/>
 - ◆ http://www4/Lehre/WS03/V_SP1/Uebung/xopen/susv2/
 - ◆ Unix98-Erweiterungen mit: `#define _XOPEN_SOURCE 500`

5 POSIX Datentypen

- Betriebssystemabhängige Typen aus `<sys/types.h>`:

- **dev_t:** Gerätenummer
- **gid_t:** Gruppen-ID
- **ino_t:** Seriennummer von Dateien (Inodenummer)
- **mode_t:** Dateiattribute (Typ, Zugriffsrechte)
- **nlink_t:** Hardlink-Zähler
- **off_t:** Dateigrößen
- **pid_t:** Prozess-ID
- **size_t:** entspricht dem ANSI-C `size_t`
- **ssize_t:** Anzahl von Bytes oder -1
- **uid_t:** User-ID

B.20 Systemaufrufe vs. Bibliotheksaufrufe

