

B Einführung in die Programmiersprache C

B.1 C vs. Java

- Java: objektorientierte Sprache
 - zentrale Frage: aus welchen Dingen besteht das Problem
 - Gliederung der Problemlösung in Klassen und Objekte
 - Hierarchiebildung: Vererbung auf Klassen, Teil-Ganze-Beziehungen
 - Ablauf: Interaktion zwischen Objekten
- C: imperative / prozedurale Sprache
 - zentrale Frage: welche Aktivitäten sind zur Lösung des Problems auszuführen
 - Gliederung der Problemlösung in Funktionen
 - Hierarchiebildung: Untergliederung einer Funktion in Teilfunktionen
 - Ablauf: Ausführung von Funktionen

B.1 C vs. Java

1 C hat nicht

- Klassen und Vererbung
- Objekte
- umfangreiche Klassenbibliotheken

2 C hat

- Zeiger und Zeigerarithmetik
- Preprozessor
- Funktionsbibliotheken

B.2 Sprachüberblick

1 Erstes Beispiel

- Die Datei `hello.c` enthält die folgenden Zeilen:

```
/* say "hello, world" */
main()
{
    printf("hello, world\n");
}
```

- Die Datei wird mit dem Kommando `cc` übersetzt:

| | |
|----------------------------|------------------|
| <code>% cc hello.c</code> | (C-Compiler) |
| oder | |
| <code>% gcc hello.c</code> | (GNU-C-Compiler) |

dadurch entsteht eine Datei `a.out`, die das ausführbare Programm enthält.

1 Erstes Beispiel (2)

- Mit der Option `-o` kann der Name der Ausgabedatei auch geändert werden – z. B.

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

- Das Programm wird durch Aufruf der Ausgabedatei ausgeführt:

```
% ./hello
hello, world
%
```

2 Aufbau eines C-Programms

- frei formulierbar - **Zwischenräume** (Leerstellen, Tabulatoren, Newline und Kommentare) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkt benachbarter Worte erforderlich
- **Kommentar** wird durch `/*` und `*/` geklammert
keine Schachtelung möglich
- **Identifizier** (Variablennamen, Marken, Funktionsnamen, ...) sind aus Buchstaben, gefolgt von Ziffern oder Buchstaben aufgebaut
 - `"_"` gilt hierbei auch als Buchstabe
 - Schlüsselwörter wie `if`, `else`, `while`, usw. können nicht als *Identifizier* verwendet werden
 - **Identifizier** müssen vor ihrer ersten Verwendung **deklariert** werden
- Anweisungen werden generell durch `;` abgeschlossen

3 Allgemeine Form eines C-Programms:

```

/* globale Variablen */
...

/* Hauptprogramm */
main(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}

/* Unterprogramm 1 */
function1(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}

/* Unterprogramm n */
functionN(...)
{
    /* lokale Variablen */
    ...
    /* Anweisungen */
    ...
}

```

4 wie ein C-Programm nicht aussehen sollte:

```
#define o define
#o __o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o__ if
#o oo_ 0
#o _o(,_,,)(void)__(,_,ooo(____))
#o __o(o_o_<<((o_o_<<(o_o_<<o_o_)+(o_o_<<o_o_)))
+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_=_oo_,_,_,_[_o];_oo____;____:___=_o-o_
_____:
_o(o_o_,____,___=(_-o_o_<__?_-
o_o:____));o_o(;;_o(o_o_, "\b",o_o_),__--);
_o(o_o_, " ",o_o_);o_(__--)_oo
_____;_o(o_o_, "\n",o_o_);_____:o_(_=_oo_(
oo_,_____,_o))_oo_____;}
```

sieht eher wie Morse-Code aus, ist aber ein **gültiges** C-Programm.

B.3 Datentypen

■ Datentypen

- Konstanten
- Variablen



- ◆ Ganze Zahlen
- ◆ Fließkommazahlen
- ◆ Zeichen
- ◆ Zeichenketten

1 Was ist ein Datentyp?

■ Menge von Werten

+

Menge von Operationen auf den Werten

- ◆ **Konstanten** stellen einen Wert dar
- ◆ **Variablen** sind Namen für Speicherplätze, die einen Wert aufnehmen können
- ➔ Konstanten und Variablen besitzen einen **Typ**

■ Datentypen legen fest:

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

■ Festlegung des Datentyps

- ◆ implizit durch Verwendung und Schreibweise (Zahlen, Zeichen)
- ◆ explizit durch **Deklaration** (Variablen)

SPL

2 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 |

SPL

2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte **Typ-Modifier** verändert werden

short, long

legt für den Datentyp `int` die Darstellungsbreite (i. a. 16 oder 32 Bit) fest.

Das Schlüsselwort `int` kann auch weggelassen werden

long double

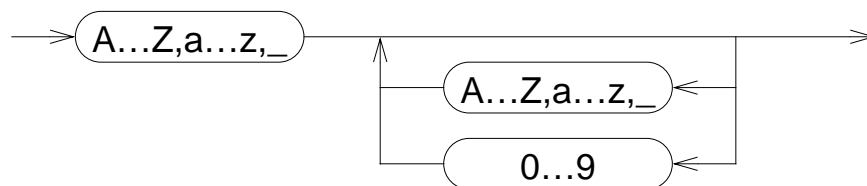
`double`-Wert mit erweiterter Genauigkeit (je nach Implementierung) – mindestens so genau wie `double`

signed, unsigned

legt für die Datentypen `char`, `short`, `long` und `int` fest, ob das erste Bit als Vorzeichenbit interpretiert wird oder nicht

3 Variablen

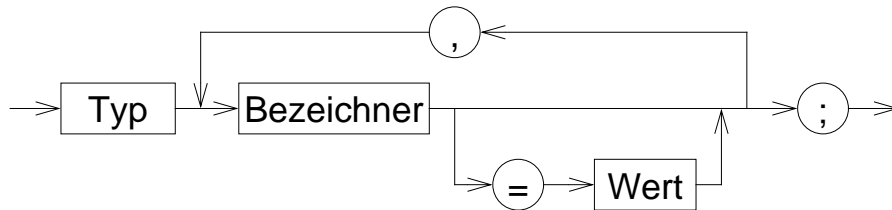
- Variablen besitzen
 - ◆ **Namen** (Bezeichner)
 - ◆ Typ
 - ◆ zugeordneten Speicherbereich für einen Wert des Typs
Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
 - ◆ **Lebensdauer**
- Bezeichner



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

3 Variablen (2)

- Typ und Bezeichner werden durch eine **Variablen-Deklaration** festgelegt
 - ◆ reine Deklarationen werden erst in einem späteren Kapitel benötigt
 - ◆ vorerst beschränken wir uns auf Deklarationen in **Variablen-Definitionen**
- eine **Variablen-Definition** deklariert eine Variable und reserviert den benötigten Speicherbereich



3 Variablen (3)

- Variablen-Definition: Beispiele

```

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

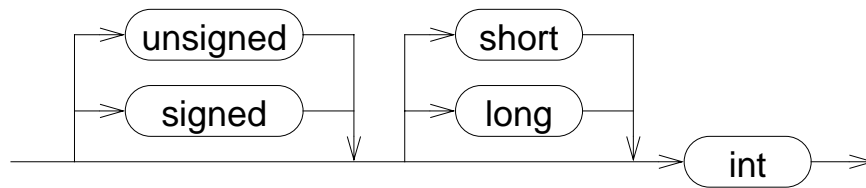
- ◆ Position im Programm:

- nach jeder "{"
- außerhalb von Funktionen

- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus der Programmstruktur

4 Ganze Zahlen

■ Definition



■ $\text{Speicherbedarf}(\text{short int}) \leq \text{Speicherbedarf}(\text{int}) \leq \text{Speicherbedarf}(\text{long int})$

■ $\text{Speicherbedarf}(\text{int})$: heute meist 32 Bit

■ Konstanten (Beispiele):

42, -117
 035 (oktal = 29₁₀)
 0x10 (hexadezimal = 16₁₀)
 0x1d (hexadezimal = 29₁₀)

4 Ganze Zahlen (2)

■ Repräsentation im Speicher (meistens):
 (Motorola, HP, SPARC, ...)

| | | | | |
|----------|----------|-----|-------|-------|
| b_{31} | b_{30} | ... | b_1 | b_0 |
|----------|----------|-----|-------|-------|

bei Intel-Prozessoren:

| | | | | |
|-------|-------|-----|----------|----------|
| b_0 | b_1 | ... | b_{30} | b_{31} |
|-------|-------|-----|----------|----------|

■ Wertebereich:

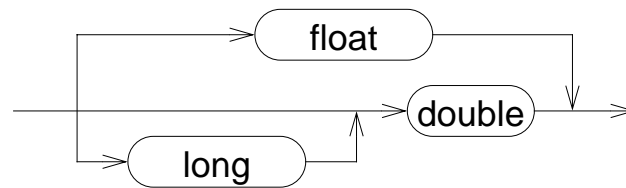
unsigned: $0 \dots 2^{32} - 1$

signed: $-2^{31} \dots 2^{31} - 1$

b_{31} ist Vorzeichenbit

5 Fließkommazahlen

■ Definition



■ Speicherbedarf(float) ≤ Speicherbedarf(double) ≤ Speicherbedarf(long double)

■ Speicherbedarf(float): 32 Bit

■ Konstanten (Beispiele):

◆ normale Dezimalpunkt-Schreibweise

3.14, -2.718, 368.345, 0.003

1.0

aber nicht einfach 1 (wäre eine int-Konstante!)

◆ 10er-Potenz Schreibweise (368.345 = 3.68345 · 10², 0.003 = 3.0 · 10⁻³)

3.68345e2, 3.0e-3

5 Fließkommazahlen (2)

■ Repräsentation: durch Mantisse m und Exponent e

◆ Wert = $m \cdot 2^e$

■ Genauigkeit

◆ hängt von der Anzahl der Bits für die Mantisse ab

■ Wertebereich

◆ hängt von der Anzahl der Bits für den Exponent ab

5 Fließkommazahlen (3)

- Repräsentation abhängig vom jeweiligen Prozessor
(häufig verwendetes Format: IEEE-Standard)

| | Mant. (Bit) | Exp. (Bit) | kleinste darstellbare positive Zahl größte darstellbare Zahl prozentuale Genauigkeit |
|--|----------------|---------------|--|
| float | 24 | 8 | $1.175494351 \cdot 10^{-38}$ $3.402823466 \cdot 10^{+38}$ $1.192092896 \cdot 10^{-07}$ |
| double | 53 | 11 | $2.2250738585072014 \cdot 10^{-308}$ $1.7976931348623157 \cdot 10^{+308}$ $2.2204460492503131 \cdot 10^{-16}$ |
| long double 80 Bit (Intel-Architektur) | 65 | 15 | $3.3621031431120935062627 \cdot 10^{-4932}$ $1.1897314953572317650213 \cdot 10^{+4932}$ $1.0842021724855044340075 \cdot 10^{-19}$ |
| long double 128 Bit | 113 | 15 | $3.36210314311209350626267781732175260 \cdot 10^{-4932}$ $1.189731495357231765085759326628007016 \cdot 10^{+4932}$ $1.925929944387235853055977942584927319 \cdot 10^{-34}$ |

6 Zeichen

- Bezeichnung: **char**
- Speicherbedarf: 1 Byte
- Repräsentation: ASCII-Code
zählt damit zu den ganzen Zahlen
- Konstanten: Zeichen durch ' ' geklammert
 - ◆ Beispiele: 'a', 'x'
 - ◆ Sonderzeichen werden durch **Escape-Sequenzen** beschrieben

| | | |
|----------------|------|-----------------|
| Tabulator: | '\t' | Backslash: '\\' |
| Zeilentrenner: | '\n' | Backspace: '\b' |
| Apostroph: | '\'' | |

6 Zeichen (2)

American Standard Code for Information Interchange (ASCII)

| | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|
| NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL |
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| BS | HT | NL | VT | NP | CR | SO | SI |
| 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| CAN | EM | SUB | ESC | FS | GS | RS | US |
| 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F |
| SP | ! | " | # | \$ | % | & | ' |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| (|) | * | + | , | - | . | / |
| 28 | 29 | 2A | 2B | 2C | 2D | 2E | 2F |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 8 | 9 | : | ; | < | = | > | ? |
| 38 | 39 | 3A | 3B | 3C | 3D | 3E | 3F |
| @ | A | B | C | D | E | F | G |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| H | I | J | K | L | M | N | O |
| 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F |
| P | Q | R | S | T | U | V | W |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| X | Y | Z | [| \ |] | ^ | _ |
| 58 | 59 | 5A | 5B | 5C | 5D | 5E | 5F |
| ` | a | b | c | d | e | f | g |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| h | i | j | k | l | m | n | o |
| 68 | 69 | 6A | 6B | 6C | 6D | 6E | 6F |
| p | q | r | s | t | u | v | w |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |
| x | y | z | { | | } | ~ | DEL |
| 78 | 79 | 7A | 7B | 7C | 7D | 7E | 7F |

7 Zeichenketten

- Bezeichnung: `char *`
- Speicherbedarf: (Länge + 1) Bytes
- Repräsentation: Folge von Einzelzeichen,
letztes Zeichen: 0-Byte (ASCII-Wert 0)
- Werte: alle endlichen Folgen von `char`-Werten
- Konstanten: Zeichenkette durch " " geklammert
 - ◆ Beispiel: `"Dies ist eine Zeichenkette"`
 - ◆ Sonderzeichen wie bei `char`, " wird durch `\` dargestellt
- Beispiel für eine Definition einer Zeichenkette:
`char *Mitteilung = "Dies ist eine Mitteilung\n";`

B.4 Ausdrücke

- Ausdruck = gültige Kombination von
Operatoren, Konstanten und Variablen
- Reihenfolge der Auswertung
 - ◆ Die Vorrangregeln für Operatoren legen die Reihenfolge fest, in der Ausdrücke abgearbeitet werden
 - ◆ Geben die Vorrangregeln keine eindeutige Aussage, ist die Reihenfolge undefiniert
 - ◆ Mit Klammern () können die Vorrangregeln überstimmt werden
 - ◆ Es bleibt dem Compiler freigestellt, Teilausdrücke in möglichst effizienter Folge auszuwerten

B.5 Operatoren

1 Zuweisungsoperator =

➔ Zuweisung eines Werts an eine Variable

■ Beispiel:

```
int a;
a = 20;
```

2 Arithmetische Operatoren

→ für alle `int` und `float` Werte erlaubt

| | |
|-----------------|---|
| <code>+</code> | Addition |
| <code>-</code> | Subtraktion |
| <code>*</code> | Multiplikation |
| <code>/</code> | Division |
| <code>%</code> | Rest bei Division, (modulo) |
| unäres - | negatives Vorzeichen (z. B. <code>-3</code>) |
| unäres + | positives Vorzeichen (z. B. <code>+3</code>) |

■ Beispiel:

```
a = -5 + 7 * 20 - 8;
```

3 spezielle Zuweisungsoperatoren

→ Verkürzte Schreibweise für Operationen auf einer Variablen

`a op= b` \equiv `a = a op b`

mit **op** \in `{ +, -, *, /, %, <<, >>, &, ^, | }`

■ Beispiele:

```
a = -8;
a += 24;      /* -> a: 16 */
a /= 2;       /* -> a: 8  */
```

4 Vergleichsoperatoren

| | |
|----|----------------|
| < | kleiner |
| <= | kleiner gleich |
| > | größer |
| >= | größer gleich |
| == | gleich |
| != | ungleich |

■ **Beachte!** Ergebnistyp `int`: wahr (true) = 1
 falsch (false) = 0

■ Beispiele:

```
a > 3
a <= 5
a == 0
if ( a >= 3 ) { ...
```

5 Logische Operatoren

➔ Verknüpfung von Wahrheitswerten (wahr / falsch)

| "nicht" | | "und" | | | "oder" | | |
|---------|---|-------|---|---|--------|---|---|
| ! | | && | f | w | | f | w |
| f | w | f | f | f | f | f | w |
| w | f | w | f | w | w | w | w |

◆ Wahrheitswerte (Boole'sche Werte) werden in C generell durch `int`-Werte dargestellt:

- Operanden in einem Ausdruck: Operand = 0: falsch
 Operand ≠ 0: wahr
- Ergebnis eines Ausdrucks: falsch: 0
 wahr: 1

5 Logische Operatoren (2)

■ Beispiel:

```
a = 5; b = 3; c = 7;
a > b && a > c
```

$\underbrace{1 \quad \text{und} \quad 0}_{0}$

■ Die Bewertung solcher Ausdrücke wird abgebrochen, sobald das Ergebnis feststeht!

```
(a > c) && ((d=a) > b)
```

$\underbrace{0} \quad \text{wird nicht ausgewertet}$

↓
 Gesamtergebnis=falsch → (d=a) wird nicht ausgeführt

6 Bitweise logische Operatoren

➔ Operation auf jedem Bit einzeln (Bit 1 = wahr, Bit 0 = falsch)

"nicht"

~

"und"

&

"oder"

|

 Antivalenz
 "exklusives oder"

| ^ | f | w |
|---|---|---|
| f | f | w |
| w | w | f |

■ Beispiele:

| | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|
| x | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| ~x | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| x 7 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| x & 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| x ^ 7 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |

7 Logische Shiftoperatoren

➔ Bits werden im Wort verschoben

<< Links-Shift
>> Rechts-Shift

■ Beispiel:

| | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|
| x | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| x << 2 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

7 Inkrement / Dekrement Operatoren

++ inkrement
-- dekrement

■ linksseitiger Operator: ++x bzw. --x

- es wird der Inhalt von x inkrementiert bzw. dekrementiert
- das Resultat wird als Ergebnis geliefert

■ rechtsseitiger Operator: x++ bzw. x--

- es wird der Inhalt von x als Ergebnis geliefert
- anschließend wird x inkrementiert bzw. dekrementiert.

■ Beispiele:

```
a = 10;
b = a++;      /* -> b: 10 und a: 11 */
c = ++a;      /* -> c: 12 und a: 12 */
```


8 Bedingte Bewertung

A ? B : C

- ➔ der Operator dient zur Formulierung von Bedingungen in Ausdrücken
- zuerst wird Ausdruck **A** bewertet
- ist **A ungleich 0**, so hat der gesamte Ausdruck als Wert den Wert des Ausdrucks **B**,
- sonst den Wert des Ausdrucks **C**
- Beispiel:

`c = a > b ? a : b;`

`/* z = max(a,b) */`

besser:

`c = (a > b) ? a : b;`

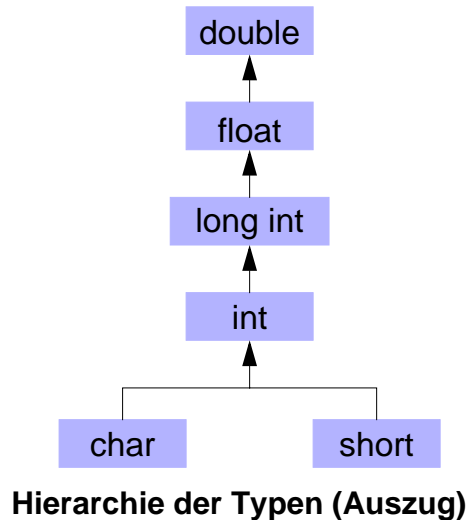
9 Komma-Operator

,

- ➔ der Komma-Operator erlaubt die Aneinanderreihung mehrerer Ausdrücke
- ein so gebildeter Ausdruck hat als Wert den Wert des letzten Teil-Ausdrucks

10 Typumwandlung in Ausdrücken

- Enthält ein Ausdruck Operanden unterschiedlichen Typs, erfolgt eine automatische Umwandlung in den Typ des in der **Hierarchie der Typen** am höchsten stehenden Operanden. (*Arithmetische Umwandlungen*)



11 Vorrangregeln bei Operatoren

| Operatorklasse | Operatoren | Assoziativität |
|--------------------|---------------|-----------------------|
| unär | ! ~ ++ -- + - | von rechts nach links |
| multiplikativ | * / % | von links nach rechts |
| additiv | + - | von links nach rechts |
| shift | << >> | von links nach rechts |
| relational | < <= > >= | von links nach rechts |
| Gleichheit | == != | von links nach rechts |
| bitweise | & | von links nach rechts |
| bitweise | ^ | von links nach rechts |
| bitweise | | von links nach rechts |
| logisch | && | von links nach rechts |
| logisch | | von links nach rechts |
| Bedingte Bewertung | ?: | von rechts nach links |
| Zuweisung | = op= | von rechts nach links |
| Reihung | , | von links nach rechts |

B.6 Einfacher Programmaufbau

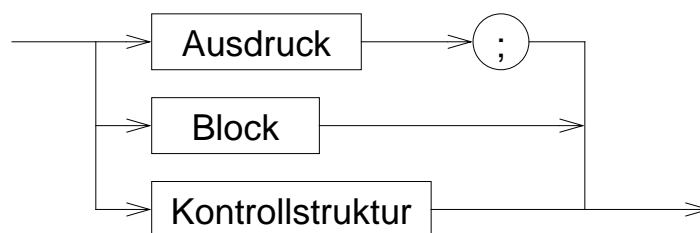
- Struktur eines C-Hauptprogramms
- Anweisungen und Blöcke
- Einfache Ein-/Ausgabe
- C-Präprozessor

1 Struktur eines C-Hauptprogramms

```
main()
{
    Variablendefinitionen
    Anweisungen
}
```

2 Anweisungen

Anweisung:



3 Blöcke

- Zusammenfassung mehrerer Anweisungen
- Lokale Variablendefinitionen → Hilfsvariablen
- Schaffung neuer Sichtbarkeitsbereiche (**Scopes**) für Variablen
 - ◆ bei Namensgleichheit ist immer die Variable des innersten Blocks sichtbar

```

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 Einfache Ein-/Ausgabe

- Jeder Prozeß (jedes laufende Programm) bekommt von der Shell als Voreinstellung drei Ein-/Ausgabekanäle:

| | |
|---------------|---------------------|
| stdin | als Standardeingabe |
| stdout | als Standardausgabe |
| stderr | Fehlerausgabe |

- Die Kanäle **stdin**, **stdout** und **stderr** sind in UNIX auf der Kommandozeile umlenkbar:

```
% prog < EingabeDatei > AusgabeDatei
```

4 Einfache Ein-/Ausgabe (2)

- Für die Sprache C existieren folgende primitive Ein-/Ausgabefunktionen für die Kanäle **stdin** und **stdout**:

| | |
|----------------|----------------------|
| getchar | zeichenweise Eingabe |
| putchar | zeichenweise Ausgabe |
| scanf | formatierte Eingabe |
| printf | formatierte Ausgabe |
- folgende Funktionen ermöglichen Ein-/Ausgabe auf beliebige Kanäle (z. B. auch **stderr**)
getc, putc, fscanf, fprintf

5 Einzelzeichen E/A

- **getchar(), getc()** ein Zeichen lesen

◆ Beispiel:

```
int c;
c = getchar();
```

```
int c;
c = getc(stdin);
```

- **putchar(), putc()** ein Zeichen schreiben

◆ Beispiel:

```
char c = 'a';
putchar(c);
```

```
char c = 'a';
putc(c, stdout);
```

- Beispiel:

```
#include <stdio.h>

/*
 * kopiere Eingabe auf Ausgabe
 */
main()
{
    int c;
    while ( (c = getchar()) != EOF )
    {
        putchar(c);
    }
}
```

6 Formatierte Ausgabe

- Aufruf: `printf (format, arg)`
- **printf** konvertiert, formatiert und gibt die **Werte (arg)** unter der Kontrolle des Formatstrings **format** aus
 - ◆ die Anzahl der Werte (*arg*) ist abhängig vom Formatstring
- sowohl für **format**, wie für **arg** sind Ausdrücke zulässig
- **format** ist vom Typ **Zeichenkette (string)**
- **arg** muß dem durch das zugehörige **Formatelement** beschriebenen Typ entsprechen

6 Formatierte Ausgabe (2)

- die Zeichenkette **format** ist aufgebaut aus:
 - ➔ **einfachem Ausgabetext**, der unverändert ausgegeben wird
 - ➔ **Formatelementen**, die Position und Konvertierung der zugeordneten **Werte** beschreiben

- Beispiele für **Formatelemente**:

Zeichenkette: `%[-][min][.max]s`
Zeichen: `%[+][-][n]c`
Ganze Zahl: `%[+][-][n][l]d`
Gleitkommazahl: `%[+][-][n][.n]f`

[] bedeutet optional

- Beispiel:

```
printf("a = %d, b = %d, a+b = %d", a, b, a+b);
```

7 C-Preprozessor — Kurzüberblick

- 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 |

8 C-Preprozessor — 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
```

9 C-Preprozessor — 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!)

B.7 Kontrollstrukturen

B.7 Kontrollstrukturen

Kontrolle des Programmablaufs in Abhängigkeit von dem Ergebnis von Ausdrücken

- Bedingte Anweisung
 - ◆ einfache Verzweigung
 - ◆ mehrfache Verzweigung
- Fallunterscheidung
- Schleifen
 - ◆ abweisende Schleife
 - ◆ nicht abweisende Schleife
 - ◆ Laufanweisung
 - ◆ Schleifensteuerung

1 Bedingte Anweisung

| Bedingung | |
|-----------|------|
| ja | nein |
| Anweisung | |

```
if ( Bedingung )
    Anweisung
```

■ Beispiel:

| Dampftemperatur > 450 Grad | |
|--|------|
| ja | nein |
| Ausgabe: 'Dampftemperatur gefährlich hoch!' | |

```
if (temp >= 450.0)
    printf("Dampftemperatur gefaehrlich hoch!\n");
```

1 Bedingte Anweisung einfache Verzweigung

| Bedingung | |
|-------------|-------------|
| ja | nein |
| Anweisung_1 | Anweisung_2 |

```
if ( Bedingung )
    Anweisung_1
else
    Anweisung_2
```

1 Bedingte Anweisung mehrfache Verzweigung

| Bedingung_1 | | |
|-------------|-------------|-------------|
| ja | nein | |
| Anweisung_1 | Bedingung_2 | |
| | ja | nein |
| | Anweisung_2 | Anweisung_3 |

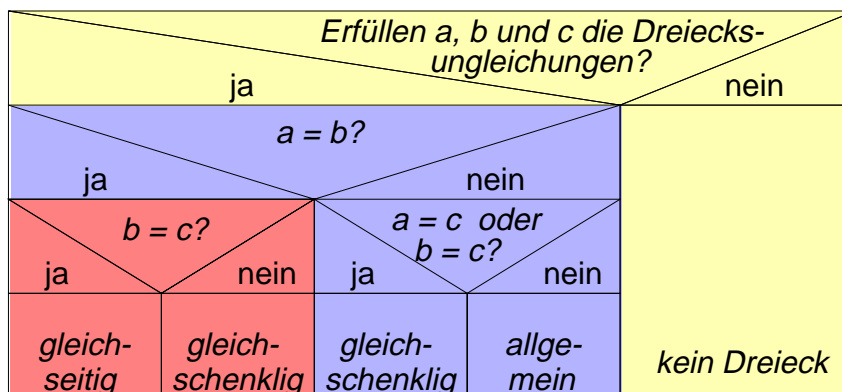
```

if ( Bedingung )
    Anweisung_1
else if ( Bedingung_2 )
    Anweisung_2
else
    Anweisung_3

```

1 Bedingte Anweisung mehrfache Verzweigung (2)

- Beispiel: Eigenschaften von Dreiecken — Struktogramm



1 Bedingte Anweisung mehrfache Verzweigung (3)

- Beispiel: Eigenschaften von Dreiecken — Programm

```
printf("Die Seitenlaengen %f, %f und %f bilden ", a, b, c);

if ( a < b+c && b < a+c && c < a+b )
    if ( a == b )
        if ( b == c )
            printf("ein gleichseitiges");
        else
            printf("ein gleichschenkliges");
        else
            if ( a==c || b == c )
                printf("ein gleichschenkliges");
            else
                printf("ein allgemeines");
    else
        printf("kein");
printf(" Dreieck");
```

2 Fallunterscheidung

- Mehrfachverzweigung = Kaskade von if-Anweisungen
- verschiedene Fälle in Abhängigkeit von einem ganzzahligen Ausdruck

| ganzzahliger Ausdruck = ? | | | | |
|---------------------------|--------|--|--------|--------|
| Wert1 | Wert2 | | | sonst |
| Anw. 1 | Anw. 2 | | Anw. n | Anw. x |

```
switch ( Ausdruck ) {
    case Wert_1:
        Anweisung_1
        break;
    case Wert_2:
        Anweisung_2
        break;
    .. .
    case Wert_n:
        Anweisung_n
        break;
    default:
        Anweisung_x
}
```

2 Fallunterscheidung — Beispiel

```
#include <stdio.h>

main()
{
    char zeichen;
    int i;
    int ziffern, leer, sonstige;

    ziffern = leer = sonstige = 0;

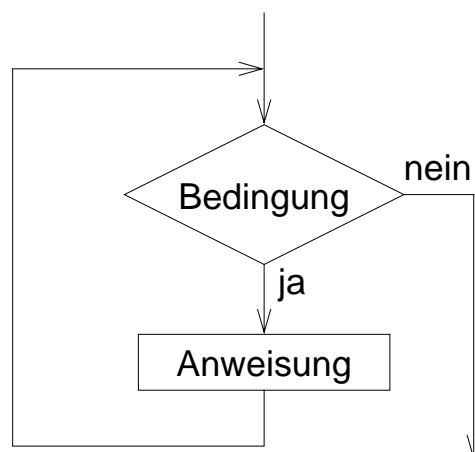
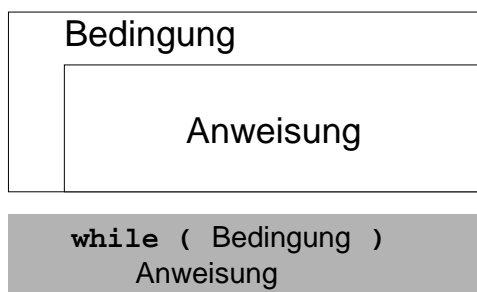
    while ((zeichen = getchar()) != EOF)
        switch (zeichen) {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                ziffern++;
                break;
            case ' ':
            case '\n':
            case '\t':
                leer++;
                break;
            default:
                sonstige++;
        }

    printf("Zahl der Ziffern = %d\n", ziffern);
    printf("Zahl der Leerzeichen = %d\n", leer);
    printf("Zahl sonstiger Zeichen = %d\n", sonstige);
}
```

3 Schleifen

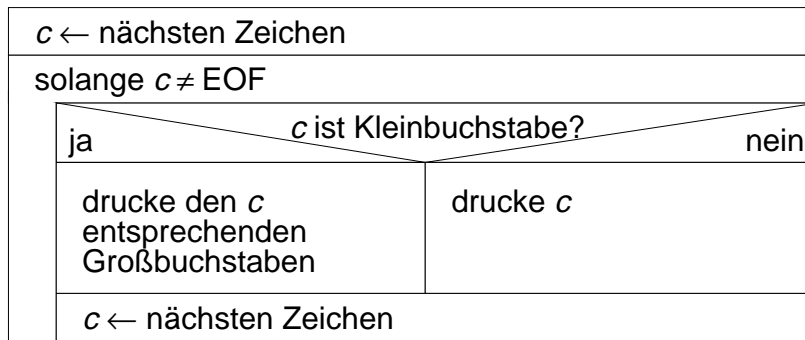
- Wiederholte Ausführung von Anweisungen in Abhängigkeit von dem Ergebnis eines Ausdrucks

4 abweisende Schleife



4 abweisende Schleife (2)

■ Beispiel: Umwandlung von Klein- in Großbuchstaben

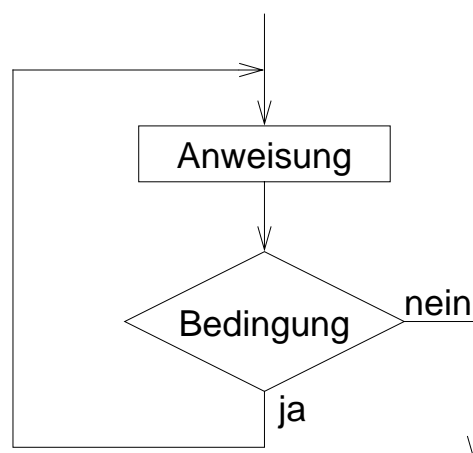
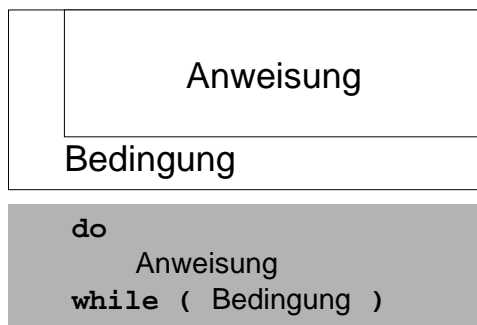


```
char c;
c = getchar();
while ( c != EOF ) {
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
    c = getchar();
}
```

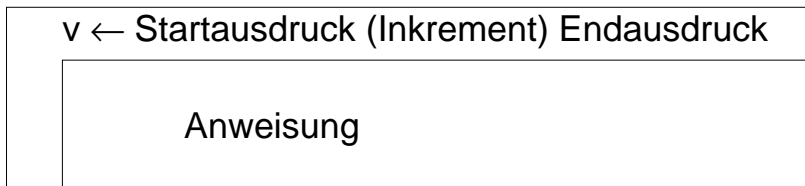
► abgekürzte Schreibweise

```
while ( (c = getchar()) != EOF )
    if ( c >= 'a' && c <= 'z' )
        putchar(c+'A'-'a');
    else
        putchar(c);
```

5 nicht-abweisende Schleife



6 Laufanweisung



```
for (v = Startausdruck; v <= Endausdruck; v += Inkrement)
    Anweisung
```

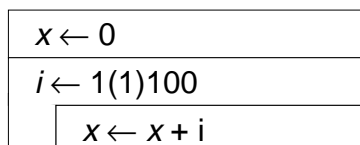
allgemein:

```
for (Ausdruck_1; Ausdruck_2; Ausdruck_3)
    Anweisung
```

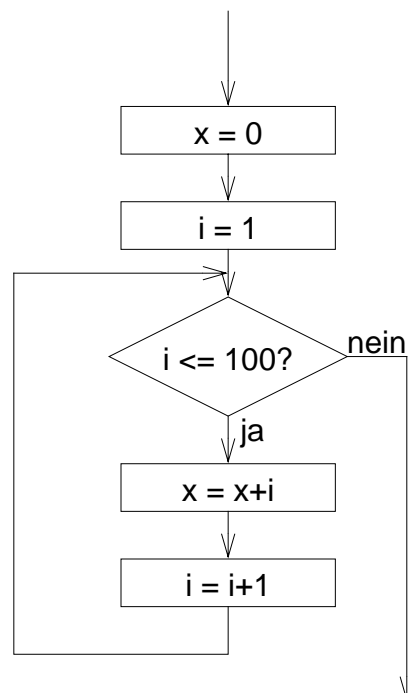
```
Ausdruck_1;
while (Ausdruck_2) {
    Anweisung
    Ausdruck_3;
}
```

6 Laufanweisung (2)

■ Beispiel: Berechne $x = \sum_{i=1}^{100} i$



```
x = 0;
for ( i=1; i<=100; i++)
    x += i;
```



7 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.8 Funktionen

1 Überblick

■ Funktion =

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

■ Funktionen sind die elementaren Bausteine für Programme

- ➔ gliedern umfangreiche, schwer überblickbare Aufgaben in kleine Komponenten
- ➔ erlauben die Wiederverwendung von Programmkomponenten
- ➔ verbergen Implementierungsdetails vor anderen Programmteilen (**Black-Box-Prinzip**)

1 Überblick (2)

- ➔ Funktionen dienen der Abstraktion
- Name und Parameter abstrahieren
 - vom tatsächlichen Programmstück
 - von der Darstellung und Verwendung von Daten
- Verwendung
 - ◆ mehrmals benötigte Programmstücke können durch Angabe des Funktionsnamens aufgerufen werden
 - ◆ Schrittweise Abstraktion
(**Top-Down**- und **Bottom-Up**-Entwurf)

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);
}
```

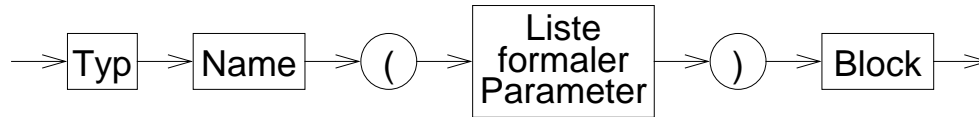
```
main()
{
    double wert;

    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 Funktionsdefinition



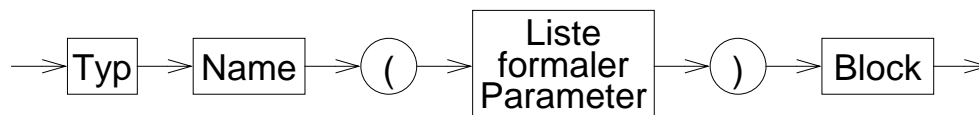
■ Typ

- ◆ Typ des Werts, der am Ende der Funktion als Wert zurückgegeben wird
- ◆ beliebiger Typ
- ◆ `void` = kein Rückgabewert

■ Name

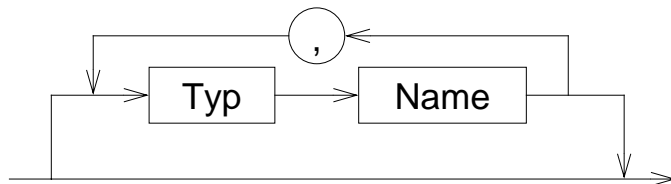
- ◆ beliebiger Bezeichner, kein Schlüsselwort

3 Funktionsdefinition (2)

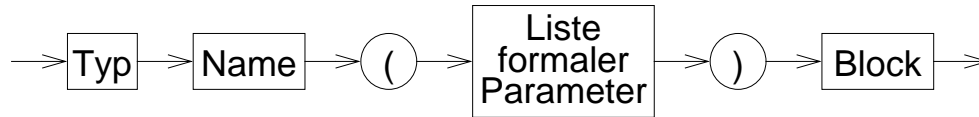


■ Liste formaler Parameter

- ◆ **Typ**: beliebiger Typ
- ◆ **Name**:
beliebiger Bezeichner
- ◆ die formalen Parameter stehen für die Werte, die beim Aufruf an die Funktion übergeben wurden (= **aktuelle Parameter**)
- ◆ die formalen Parameter verhalten sich wie Variablen, die im **Funktionsrumpf** definiert sind und mit den aktuellen Parametern vorbelegt werden



3 Funktionsdefinition (3)



■ Block

- ◆ beliebiger Block
- ◆ zusätzliche Anweisung

```
return ( Ausdruck );
```

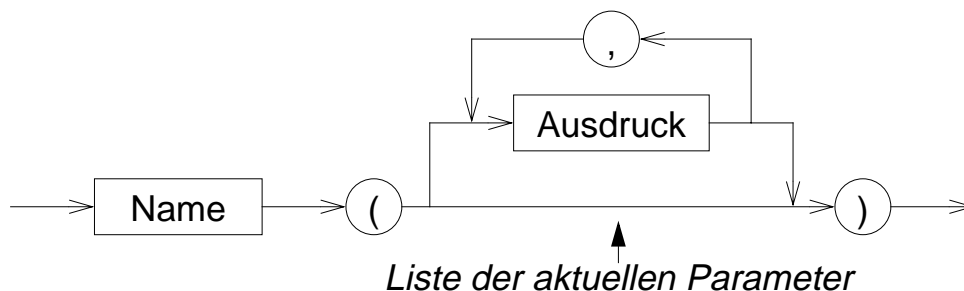
oder

```
return;
```

bei void-Funktionen

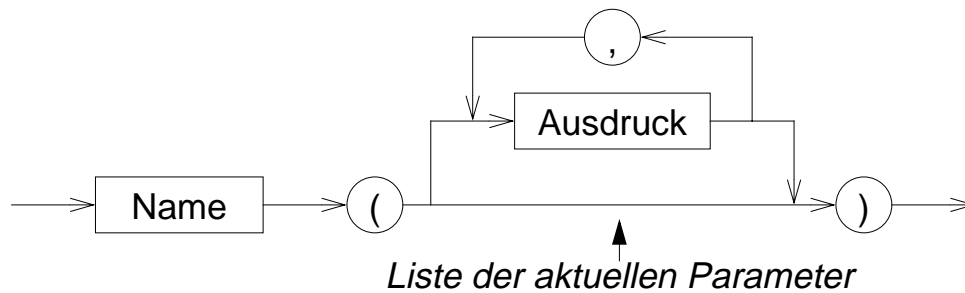
- Rückkehr aus der Funktion: das Programm wird nach dem Funktionsaufruf fortgesetzt
- der Typ des Ausdrucks muß mit dem Typ der Funktion übereinstimmen
- die Klammern können auch weggelassen werden

4 Funktionsaufruf



- Jeder Funktionsaufruf ist ein Ausdruck
- void-Funktionen können keine Teilausdrücke sein
 - ◆ wie Prozeduren in anderen Sprachen (z. B. Pascal)

4 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 Beispiel

```
float power (float b, int e)
{
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    return(prod);
}
```

```
float x, y;

y = power(2+x,4)+3;
```

≡

```
float x, y, power;
{
    float b = 2+x;
    int e = 4;
    float prod = 1.0;
    int i;

    for (i=1; i <= e; i++)
        prod *= b;
    power = prod;
}
y=power+3;
```

6 Regeln

- Funktionen werden global definiert
 - ↳ keine lokalen Funktionen/Prozeduren wie z. B. in Pascal
- `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)

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

6 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**

6 Regeln (2)

■ Funktionsdeklaration

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

- ◆ Syntax:

Typ Name (Liste formaler Parameter);

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

- ◆ Beispiel:

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

7 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);
}
```

8 Parameterübergabe an Funktionen

- allgemein in Programmiersprachen vor allem zwei Varianten:
 - call by value
 - call by reference

call by value

- Normalfall in C
- 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

8 Parameterübergabe an Funktionen (2)

call by reference

- In C nur indirekt mit Hilfe von Zeigern realisierbar
- Der Übergabeparameter ist eine Variable und die aufgerufene Funktion erhält die Speicheradresse dieser Variablen
 - die Funktion kann den Übergabeparameter durch Zugriff auf den formalen Parameter lesen
 - wenn die Funktion den Wert des formalen Parameters verändert, ändert sie den Inhalt der Speicherzelle des aktuellen Parameters
 - auch der Wert der Variablen (aktueller Parameter) beim Aufrufer der Funktion ändert sich dadurch

B.9 Programmstruktur & Module

1 Softwaredesign

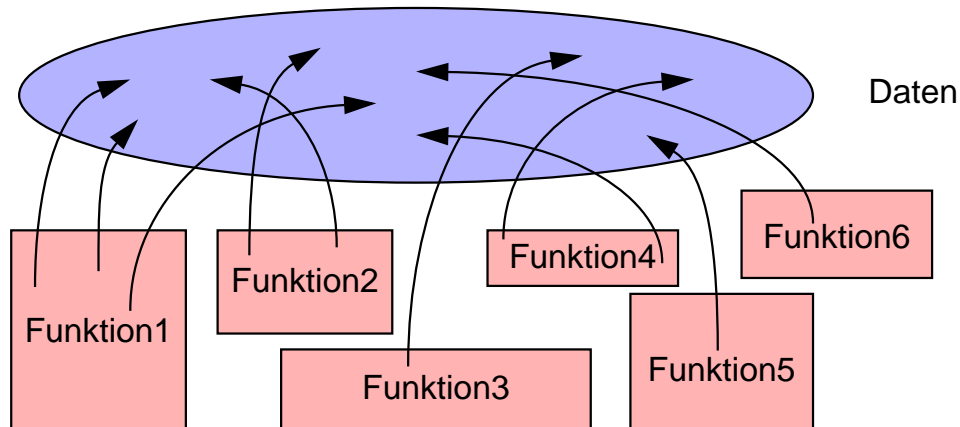
- Grundsätzliche Überlegungen über die Struktur eines Programms **vor** Beginn der Programmierung
- Verschiedene Design-Methoden
 - ◆ Top-down Entwurf / Prozedurale Programmierung
 - traditionelle Methode
 - bis Mitte der 80er Jahre fast ausschließlich verwendet
 - an Programmiersprachen wie Fortran, Cobol, Pascal oder C orientiert
 - ◆ Objekt-orientierter Entwurf
 - moderne, sehr aktuelle Methode
 - Ziel: Bewältigung sehr komplexer Probleme
 - auf Programmiersprachen wie C++, Smalltalk oder Java ausgerichtet

2 Top-down Entwurf

- Zentrale Fragestellung
 - ◆ was ist zu tun?
 - ◆ in welche Teilaufgaben läßt sich die Aufgabe untergliedern?
 - Beispiel: Rechnung für Kunden ausgeben
 - Rechnungspositionen zusammenstellen
 - Lieferungsposten einlesen
 - Preis für Produkt ermitteln
 - Mehrwertsteuer ermitteln
 - Rechnungspositionen addieren
 - Positionen formatiert ausdrucken

2 Top-down Entwurf (2)

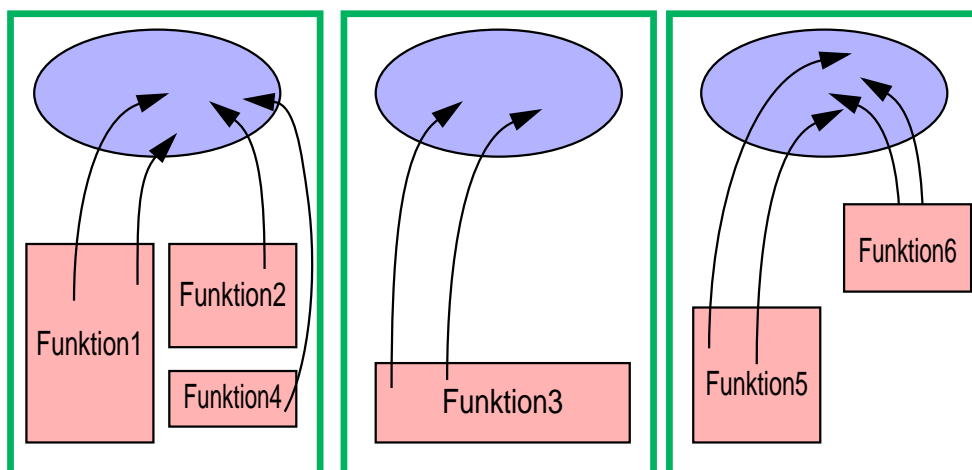
- Problem:
Gliederung betrifft nur die Aktivitäten, nicht die Struktur der Daten
- Gefahr:
Sehr viele Funktionen arbeiten "wild" auf einer Unmenge schlecht strukturierter Daten



2 Top-down Entwurf (3) Modul-Bildung

- Lösung:
Gliederung von Datenbeständen zusammen mit Funktionen, die darauf operieren

➔ **Modul**



3 Module in C

- Teile eines C-Programms können auf mehrere .c-Dateien (C-Quelldateien) verteilt werden
- Logisch zusammengehörende Daten und die darauf operierenden Funktionen sollten jeweils zusammengefaßt werden

➔ **Modul**

- Jede C-Quelldatei kann separat übersetzt werden (Option `-c`)
 - Zwischenergebnis der Übersetzung wird in einer .o-Datei abgelegt

```
% cc -c main.c           (erzeugt Datei main.o)
% cc -c f1.c             (erzeugt Datei f1.o)
% cc -c f2.c f3.c        (erzeugt f2.o und f3.o)
```

- Das Kommando `cc` kann mehrere .c-Dateien übersetzen und das Ergebnis — zusammen mit .o-Dateien — binden:

```
% cc -o prog main.o f1.o f2.o f3.o f4.c f5.c
```

3 Module in C

!!! .c-Quelldateien auf keinen Fall mit Hilfe der `#include`-Anweisung in andere Quelldateien einkopieren

- Bevor eine Funktion aus einem anderen Modul aufgerufen werden kann, muß sie **deklariert** werden
 - Parameter und Rückgabewerte müssen bekannt gemacht werden
- Makrodefinitionen und Deklarationen, die in mehreren Quelldateien eines Programms benötigt werden, werden zu **Header-Dateien** zusammengefaßt
 - ◆ Header-Dateien werden mit der `#include`-Anweisung des Preprozessors in C-Quelldateien einkopiert
 - ◆ der Name einer Header-Datei endet immer auf `.h`

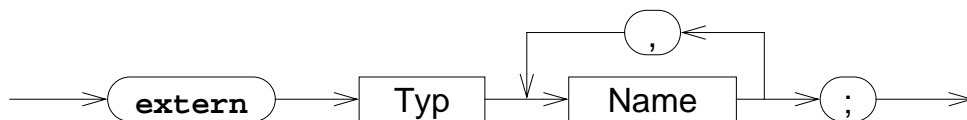
4 Gültigkeit von Namen

- Gültigkeitsregeln legen fest, welche Namen (Variablen und Funktionen) wo im Programm bekannt sind
- Mehrere Stufen
 1. Global im gesamten Programm
(über Modul- und Funktionsgrenzen hinweg)
 2. Global in einem Modul
(auch über Funktionsgrenzen hinweg)
 3. Lokal innerhalb einer Funktion
 4. Lokal innerhalb eines Blocks
- Überdeckung bei Namensgleichheit
 - eine lokale Variable innerhalb einer Funktion überdeckt gleichnamige globale Variablen
 - eine lokale Variable innerhalb eines Blocks überdeckt gleichnamige globale Variablen und gleichnamige lokale Variablen in umgebenden Blöcken

5 Globale Variablen

Gültig im gesamten Programm

- Globale Variablen werden außerhalb von Funktionen definiert
- Globale Variablen sind ab der Definition in der gesamten Datei zugreifbar
- Globale Variablen, die in anderen Modulen **definiert** wurden, müssen vor dem ersten Zugriff bekanntgemacht werden
(**extern-Deklaration** = Typ und Name bekanntmachen)



- Beispiele:

```
extern int a, b;
extern char c;
```

5 Globale Variablen (2)

■ Probleme mit globalen Variablen

- ◆ Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
- ◆ Funktionen können Variablen ändern, ohne daß der Aufrufer dies erwartet (Seiteneffekte)
- ◆ Programme sind schwer zu pflegen, weil bei Änderungen der Variablen erst alle Programmteile, die sie nutzen gesucht werden müssen

➔ **globale Variablen möglichst vermeiden!!!**

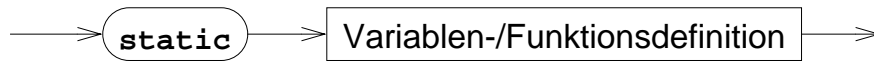
5 Globale Funktionen

- Funktionen sind generell global (es sei denn, die Erreichbarkeit wird explizit auf das Modul begrenzt)
- Funktionen aus anderen Modulen müssen ebenfalls vor dem ersten Aufruf **deklariert** werden (= Typ, Name und Parametertypen bekanntmachen)
- Das Schlüsselwort **extern** ist bei einer Funktionsdeklaration nicht notwendig
- Beispiele:


```
double sinus(double);
float power(float, int);
```
- Globale Funktionen (und soweit vorhanden die globalen Daten) bilden die äußere Schnittstelle eines Moduls
 - "vertragliche" Zusicherung an den Benutzer des Moduls

6 Einschränkung der Gültigkeit auf ein Modul

- Zugriff auf eine globale Variable oder Funktion kann auf das Modul (= die Datei) beschränkt werden, in der sie definiert wurde
 - Schlüsselwort **static** vor die Definition setzen

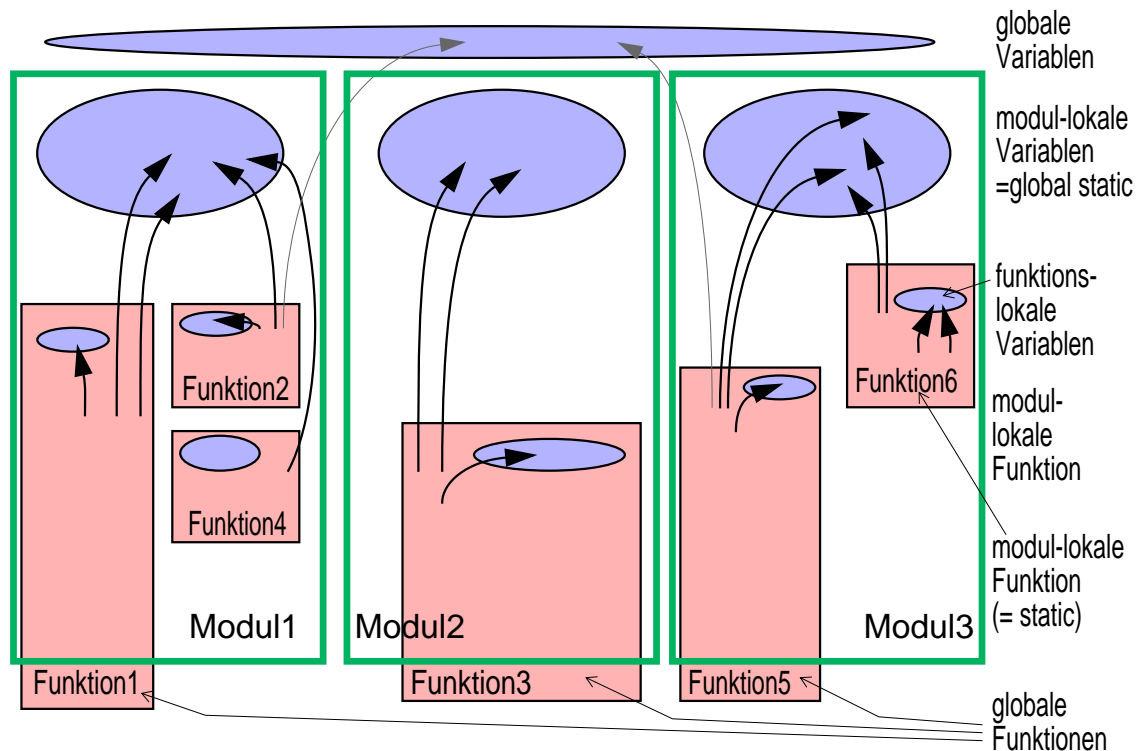


- **extern**-Deklarationen in anderen Modulen sind nicht möglich
- Die **static**-Variablen bilden zusammen den Zustand eines Moduls, die Funktionen des Moduls operieren auf diesem Zustand
- Hilfsfunktionen innerhalb eines Moduls, die nur von den Modulfunktionen benötigt werden, sollten immer static definiert werden
 - sie werden dadurch nicht Bestandteil der Modulschnittstelle (= des "Vertrags" mit den Modulbenutzern)
- !!! das Schlüsselwort **static** gibt es auch bei lokalen Variablen (mit anderer Bedeutung!)

7 Lokale Variablen

- Variablen, die innerhalb einer Funktion oder eines Blocks definiert werden, sind lokale Variablen
- bei Namensgleichheit zu globalen Variablen oder lokalen Variablen eines umgebenden Blocks gilt die jeweils letzte Definition
- lokale Variablen sind außerhalb des Blocks, in dem sie definiert wurden, nicht zugreifbar und haben dort keinen Einfluß auf die Zugreifbarkeit von Variablen

8 Gültigkeitsbereiche — Übersicht



9 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
 - ◆ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
 - statische (**static**) Variablen
 - ◆ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
 - dynamische (**automatic**) Variablen

9 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
 - der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
 - ➔ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
 - die Initialisierung wird bei jedem Eintritt in den Block wiederholt
- !!! **wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert (= irgendwas)**

9 Lebensdauer von Variablen (2)

static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort **static** eine **Lebensdauer über die gesamte Programmausführung** hinweg
 - ➔ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
- !!! Das Schlüsselwort **static** hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können durch beliebige konstante Ausdrücke initialisiert werden
 - die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
 - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

10 Wertaustausch 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 vielen 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
 - Modul suchen, dem die Variable zugeordnet werden kann!!!
- ◆ Variablen, deren Lebensdauer nicht beschränkt sein darf, die aber nicht in `main()` deklariert werden sollen
 - in zugehöriger Funktion lokal-static definieren

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;;
        ...
    }
}
```

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));
}

...

```


Les



B-Einfuehrung.fm 2004-04-20 09.24

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

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

B.10 Felder

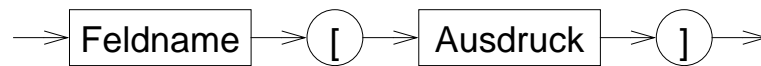
- ```
char name1[5] = "Otto";
char name2[] = "Otto";
```

B-Einfuehrung.fm 2004-04-20 09.24

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

### 3 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'
```

#### ■ Beispiel Vektoraddition:

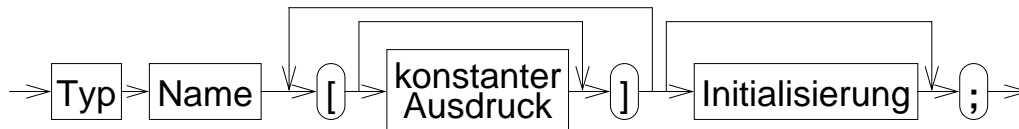
```
float v1[4], v2[4], sum[4];
int i;
...
for (i=0; i < 4; i++)
 sum[i] = v1[i] + v2[i];
for (i=0; i < 4; i++)
 printf("sum[%d] = %f\n", i, sum[i]);
```

### 4 Mehrdimensionale Felder

#### ■ neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren

◆ ihre Bedeutung in C ist gering

#### ■ Definition eines mehrdimensionalen Feldes

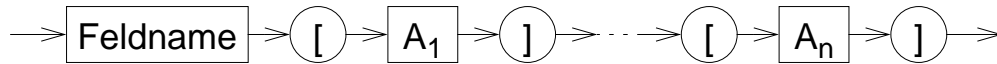


#### ■ Beispiel:

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

## 5 Mehrdimensionale Felde — Zugriffe auf Feldelemente

- Indizierung:



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

- Beispiel:

```
feld[2][3] = 10;
```

## 6 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 nicht initialisiert!

## 7 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 (ANSI)

## 8 Beispiele

- Bestimmung der Länge einer Zeichenkette (*String*)

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

## 8 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" */
```

## B.11 Strukturen

### 1 Motivation

#### ■ Felder fassen Daten eines einheitlichen Typs zusammen

- ungeeignet für gemeinsame Handhabung von Daten unterschiedlichen Typs

#### ■ Beispiel: Daten eines Studenten

|                       |                                 |
|-----------------------|---------------------------------|
| – Nachname            | <code>char nachname[25];</code> |
| – Vorname             | <code>char vorname[25];</code>  |
| – Geburtsdatum        | <code>char gebdatum[11];</code> |
| – Matrikelnummer      | <code>int matrnr;</code>        |
| – Übungsgruppennummer | <code>short gruppe;</code>      |
| – Schein bestanden    | <code>char best;</code>         |

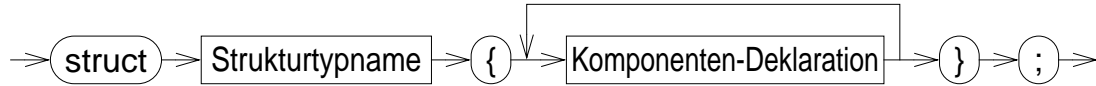
#### ■ Möglichkeiten der Repräsentation in einem Programm

- ◆ einzelne Variablen → sehr umständlich, keine "Abstraktion"
- ◆ Datenstrukturen

## 2 Deklaration eines Strukturtyps

- Durch eine Strukturtyp-Deklaration wird dem Compiler der Aufbau einer Datenstruktur unter einem Namen bekanntgemacht
  - ➔ deklariert einen neuen Datentyp (wie `int` oder `float`)

- Syntax



- **Strukturtypname**

- ◆ beliebiger Bezeichner, kein Schlüsselwort
- ◆ kann in nachfolgenden Struktur-Definitionen verwendet werden

- **Komponenten-Deklaration**

- ◆ entspricht normaler Variablen-Definition, aber keine Initialisierung!
- ◆ in verschiedenen Strukturen dürfen die gleichen Komponentennamen verwendet werden (eigener Namensraum pro Strukturtyp)

## 2 Deklaration eines Strukturtyps (2)

- Beispiele

```
struct student {
 char nachname[25];
 char vorname[25];
 char gebdatum[11];
 int matrnr;
 short gruppe;
 char best;
};
```

```
struct komplex {
 double re;
 double im;
};
```

### 3 Definition einer Struktur

- Die Definition einer Struktur entspricht einer Variablen-Definition
  - ◆ Name der Struktur zusammen mit dem Datentyp bekanntmachen
  - ◆ Speicherplatz anlegen
- Eine Struktur ist eine Variable, die ihre Komponentenvariablen umfasst
- Syntax



- Beispiele

```
struct student stud1, stud2;
struct komplex c1, c2, c3;
```

- Strukturdeklaration und -definition können auch in einem Schritt vorgenommen werden

### 4 Zugriff auf Strukturkomponenten

- **.-Operator**
  - **x.y** ≡ Zugriff auf die Komponente **y** der Struktur **x**
  - **x.y** verhält sich wie eine normale Variable vom Typ der Strukturkomponenten **y** der Struktur **x**

- Beispiele

```
struct komplex c1, c2, c3;
...
c3.re = c1.re + c2.re;
c3.im = c1.im + c2.im;

struct student stud1;
...
if (stud1.matrnr < 1500000) {
 stud1.best = 'y';
}
```



## 5 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 Position**

➔ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

## 6 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);
}
```

## 7 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].matrnr < 1500000) {
 gruppe8[i].best = 'y';
 } else {
 gruppe8[i].best = 'n';
 }
}
```

## 8 Strukturen in Strukturen

- Die Komponenten einer Struktur können wieder Strukturen sein
- Beispiel

```
struct name {
 char nachname[25];
 char vorname[25];
};

struct student {
 struct name name;
 char gebdatum[11];
 int matrnr;
 short gruppe;
 char best;
}

struct prof {
 struct name pname;
 char gebdatum[11];
 int lehrstuhlNr;
}

struct student stud1;
strcpy(stud1.name.nachname, "Meier");
if (stud1.name.nachname[0] == 'M') {
 ...
}
```

## 9 Compilerabhängige Eigenschaften

### ■ Funktionsparameter

- ◆ alte C-Compiler erlauben die Übergabe ganzer Strukturen als Funktionsparameter oder -rückgabewert nicht
  - dann nur Zeiger auf Strukturen möglich

### ■ Zuweisungen

- ◆ moderne C-Compiler erlauben die Zuweisung kompletter Strukturen
- ◆ Beispiel

```
struct komplex c1, c2;
c2 = c1;
```

- ◆ wenn der Compiler dies nicht erlaubt, komponentenweise zuweisen

```
c2.re = c1.re; c2.im = c1.im;
```

### ■ Namen von Strukturkomponenten

- ◆ alte C-Compiler erlauben nicht gleiche Komponentennamen in unterschiedlichen Strukturen

## B.12 Zeiger(-Variablen)

### 1 Einordnung

#### ■ Konstante:

Bezeichnung für einen Wert

'a' ≡ 0110 0001

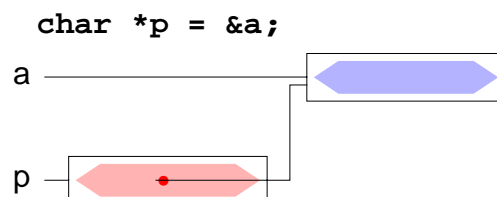
#### ■ Variable:

Bezeichnung eines Datenobjekts



#### ■ Zeiger-Variable (Pointer):

Bezeichnung einer Referenz auf ein Datenobjekt



## 2 Überblick

- Eine Zeigervariable (**pointer**) enthält als Wert einen Verweis auf den Inhalt einer anderen Variablen
  - ➔ *der Zeiger verweist auf die Variable*
- Über diese Adresse kann man **indirekt** auf die andere 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 ;
```

## 4 Beispiele

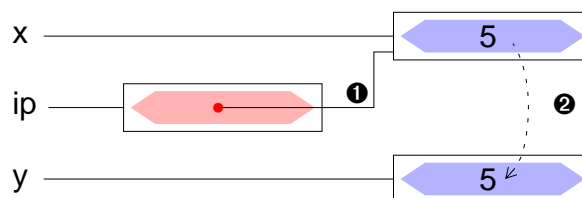
```
int x = 5;

int *ip;

int y;

ip = &x; ①

y = *ip; ②
```



## 5 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

## 6 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*

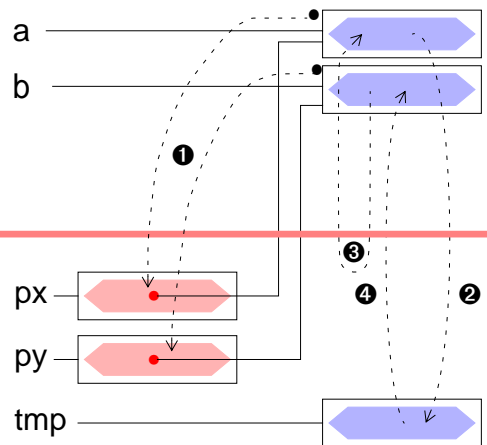
## 6 ... Zeiger als Funktionsargumente (2)

### ■ 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
}
```



## 7 Zeiger und Felder (1)

- 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:

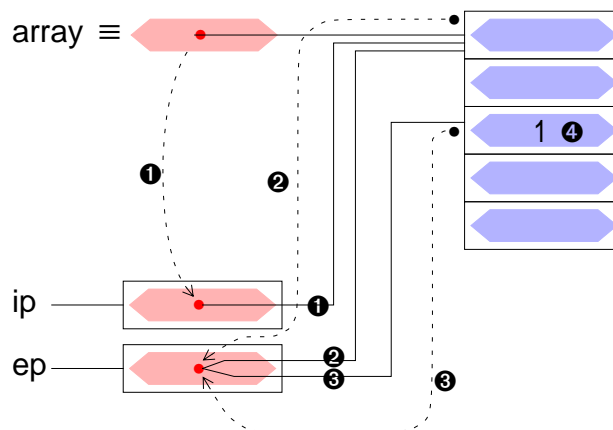
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```

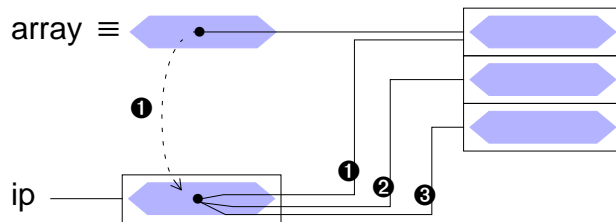


## 8 Arithmetik mit Adressen

- ++ -Operator: Inkrement = nächstes Objekt

```
int array[3];
int *ip = array; ❶

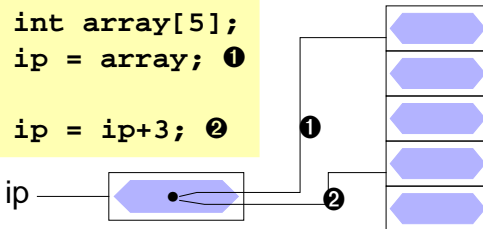
ip++; ❷
ip++; ❸
```



- -- -Operator: Dekrement = vorheriges Objekt

- +, -  
Addition und Subtraktion von  
Zeigern und ganzzahligen Werten.

Dabei wird immer die Größe des  
Objektyps berücksichtigt!



!!! **Achtung:** Assoziativität der Operatoren beachten !!

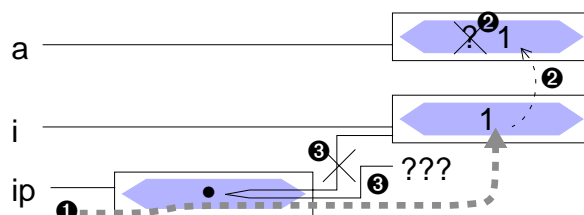
## 9 Vorrangregeln bei Operatoren

| Operatorklasse | Operatoren                 | Assoziativität        |
|----------------|----------------------------|-----------------------|
| primär         | ( ) Funktionsaufruf<br>[ ] | von links nach rechts |
| unär           | ! ~ ++ -- + - * &          | von rechts nach links |
| multiplikativ  | * / %                      | von links nach rechts |
| ...            |                            |                       |

## 10 Beispiele

```
int a, i, *ip;
i = 1;
ip = &i;
```

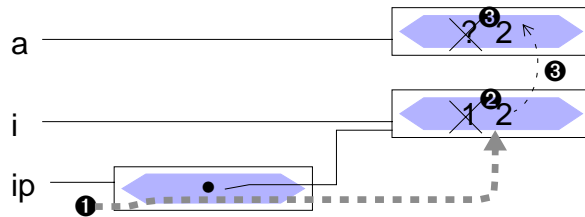
```
a = *ip++;
> (1) a = *ip++;
> (2) a = *ip++;
```



## 10 Beispiele (2)

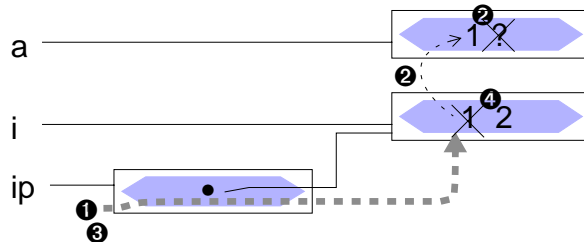
```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = ++*ip;
(1) a = ++*ip;
 ① *ip
 ② ++ → *ip
(2) a = ++*ip;
 ③
```



```
int a, i, *ip;
i = 1;
ip = &i;
```

```
a = (*ip)++;
(1) a = (*ip)++;
 ①
 ②
(2) a = (*ip)++;
 ③
 ④ ++ → *ip
```



## 11 Zeigerarithmetik und Felder

- Ein Feldname ist eine Konstante, für die Adresse des Feldanfangs
  - Feldname ist ein ganz normaler Zeiger
    - Operatoren für Zeiger anwendbar ( \*, [ ] )
  - aber keine Variable ➔ keine Modifikationen erlaubt
    - keine Zuweisung, kein ++, --, +=, ...

- es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array; /* ≡ int *ip = &array[0] */
int *ep;
```

```
/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
(ip+i) = 1; / Vorrang ! */
*(array+i) = 1;
```

```
ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```



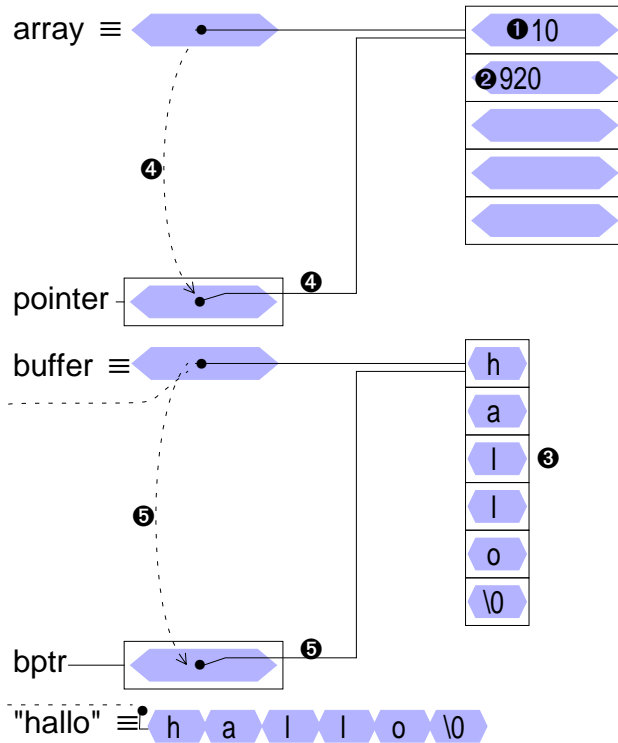
# 11 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```

```
① array[0] = 10;
② array[1] = 920;
③ strcpy(buffer, "hallo");
④ pointer = array;
⑤ bptr = buffer;
```

Formale Parameter  
der Funktion strcpy

```
ziel — []
quelle — []
```



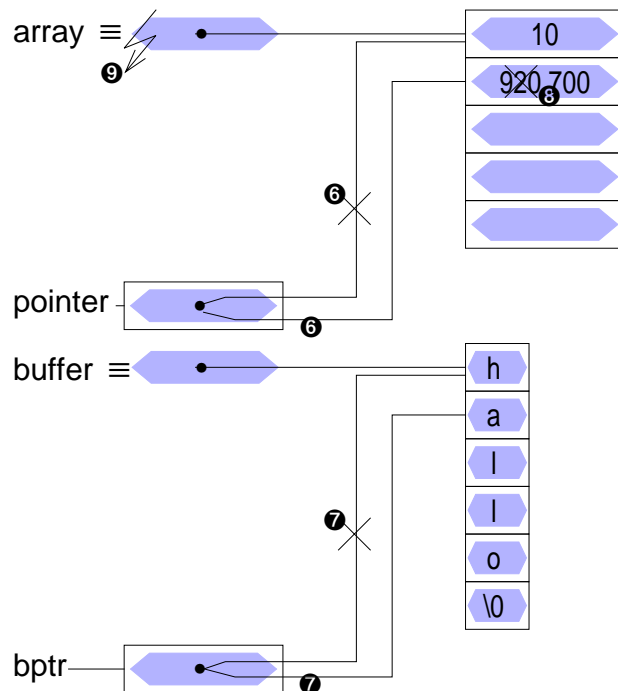
# 11 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;
```

```
① array[0] = 10;
② array[1] = 920;
③ strcpy(buffer, "hallo");
④ pointer = array;
⑤ bptr = buffer;
```

```
⑥ pointer++;
⑦ bptr++;
⑧ *pointer = 700;
```

```
⑨ array++;
```



## 12 Vergleichsoperatoren und Adressen

- Neben den arithmetischen Operatoren lassen sich auch die Vergleichsoperatoren auf Zeiger (allgemein: Adressen) anwenden:

|    |                |
|----|----------------|
| <  | kleiner        |
| <= | kleiner gleich |
| >  | größer         |
| >= | größer gleich  |
| == | gleich         |
| != | ungleich       |

## B.13 Felder als Funktionsparameter

- Funktionsaufruf und Deklaration der formalen Parameter am Beispiel eines `int`-Feldes:

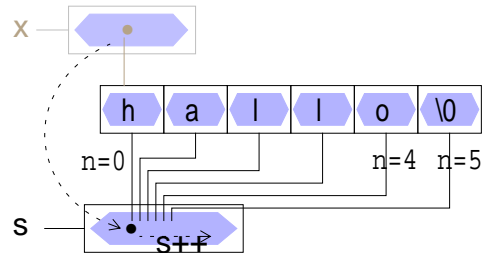
```
int a, b;
int feld[20];
func(a, feld, b);
...
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
```

- Der Feldname ist eine Konstante Adresse, die als Argument eines Funktionsaufrufs *by-value* übergeben wird (= Funktion erhält Kopie!)
- Diese Kopie kann von der aufgerufenen Funktion – unabhängig von der Deklaration des formalen Parameters – wie eine Zeigervariable verwendet werden
- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!

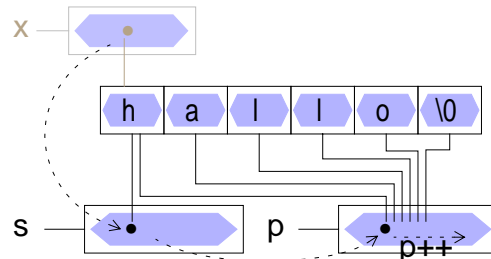
## B.14 Zeiger, Felder und Zeichenketten

- Zeichenketten sind Felder von Einzelzeichen (**char**), die in der internen Darstellung durch ein `'\0'`-Zeichen abgeschlossen sind
- Beispiel: Länge eines Strings ermitteln — Aufruf `strlen(x);`

```
/* 1. Version */
int strlen(char *s)
{
 int n;
 for (n=0; *s != '\0'; s++)
 n++;
 return(n);
}
```



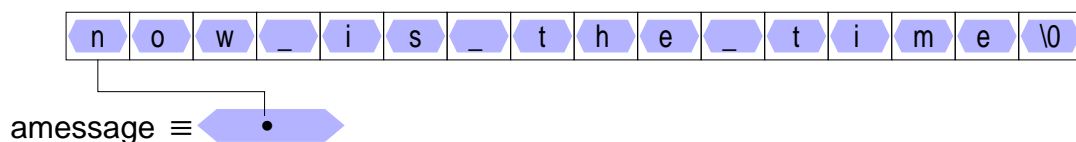
```
/* 2. Version */
int strlen(char *s)
{
 char *p = s;
 while (*p != '\0')
 p++;
 return(p-s);
}
```



## B.14... Zeiger, Felder und Zeichenketten (2)

- wird eine Zeichenkette zur Initialisierung eines char-Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

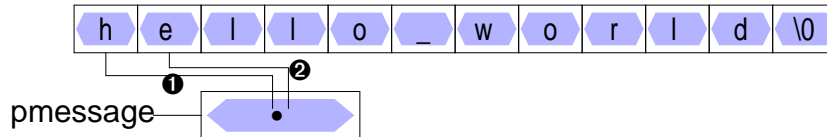
```
char amessage[] = "now is the time";
```



## B.14... Zeiger, Felder und Zeichenketten (3)

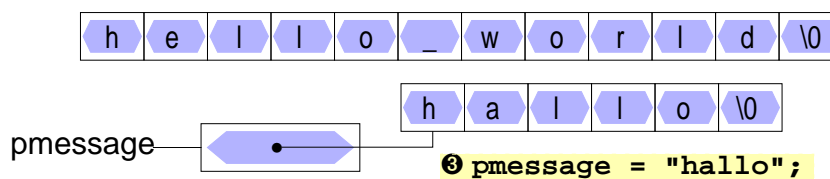
- wird eine Zeichenkette zur Initialisierung eines `char`-Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```



```
pmessage++; ②
printf("%s", pmessage); /* gibt "ello world" aus */
```

- ➔ wird dieser Zeiger überschrieben, ist die Zeichenkette nicht mehr adressierbar!



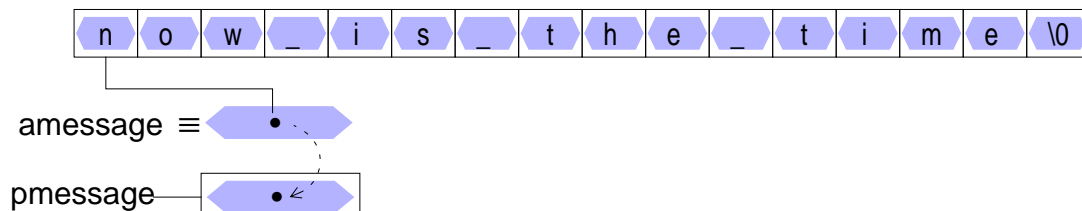
```
③ pmessage = "hallo";
```

## B.14... Zeiger, Felder und Zeichenketten (4)

- die Zuweisung eines `char`-Zeigers oder einer Zeichenkette an einen `char`-Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger `pmessage` lediglich die Adresse der Zeichenkette `"now is the time"` zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

## B.14... Zeiger, Felder und Zeichenketten (5)

### ■ Zeichenketten kopieren

```

/* 1. Version */
void strcpy(char s[], t[])
{
 int i=0;
 while ((s[i] = t[i]) != '\0')
 i++;
}

/* 2. Version */
void strcpy(char *s, *t)
{
 while ((*s = *t) != '\0')
 s++, t++;
}

/* 3. Version */
void strcpy(char *s, *t)
{
 while (*s++ = *t++)
 ;
}

```

## B.14... Zeiger, Felder und Zeichenketten (6)

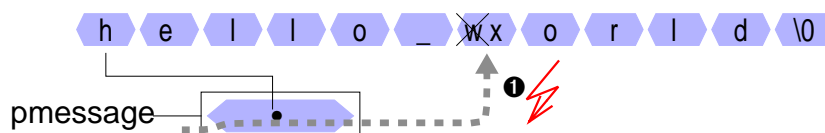
- in ANSI-C können Zeichenketten in nicht-modifizierbaren Speicherbereichen angelegt werden (je nach Compiler)

- ➔ Schreiben in Zeichenketten  
(Zuweisungen über dereferenzierte Zeiger)  
kann zu Programmabstürzen führen!

- Beispiel:

```
strcpy("zu ueberschreiben", "reinschreiben");
```

```
char *pmessage = "hello world";
```



```
pmessage[6] = 'x'; ❶
```

aber!

```
char amessage[] = "hello world";
amessage[6] = 'x';
```

ok!

## B.15 Beispiel: dynamische Speicherverwaltung

- die folgenden zwei Funktionen verwalten einen globalen, zusammenhängenden Speicherbereich,
  - aus dem freier Speicher angefordert werden kann (**alloc**)
  - und in den freigegebener Speicher wieder integriert wird (**free**)
- diese Version einer dynamischen Speicherverwaltung ist rudimentär und soll lediglich einen Eindruck von einer möglichen Realisierung geben
- in der Regel werden für diese Aufgabenstellung die Funktionen **malloc()** und **free()** aus der C-Bibliothek verwendet

## B.15... Beispiel: dynamische Speicherverwaltung (2)

- Globale Definitionen

```
#define NULL (char *)0
#define ALLOCSIZE 1000
static char allocbuf[ALLOCSIZE]
static char *allocp = allocbuf;
```

- Anfordern von Speicher

```
char *alloc(int n)
{
 if (allocp+n <= allocbuf+ALLOCSIZE) {
 allocp += n;
 return (allocp - n);
 } else
 return (NULL);
}
```



## B.15... Beispiel: dynamische Speicherverwaltung (3)

### ■ Freigabe von Speicher

```
void free(char *p)
{
 if (p >= allocbuf &&
 p < allocbuf+ALLOCSIZE)
 allocp = p;
}
```

## B.16 Felder von Zeigern

## B.16 Felder von Zeigern

### ■ Auch von Zeigern können Felder gebildet werden

### ■ Deklaration

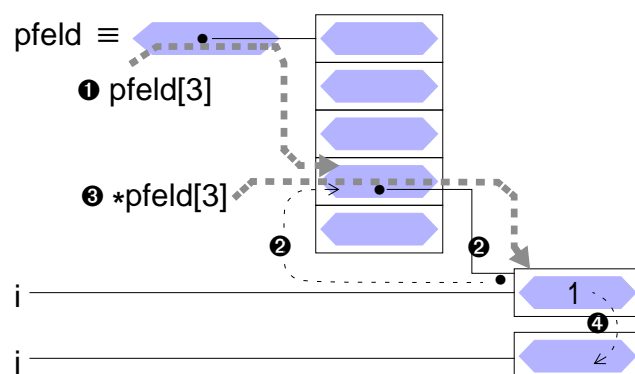
```
int *pfeld[5];
int i = 1;
int j;
```

### ■ Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ②
```

### ■ Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

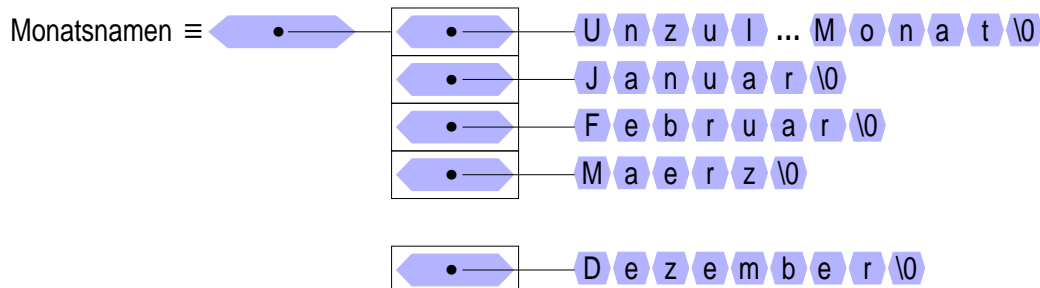
```
j = *pfeld[3]; ④
```



## B.16 Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
 static char *Monatsnamen[] = {
 "Unzulaessiger Monat",
 "Januar",
 ...
 "Dezember"
 };
 return ((n<0 || n>12) ?
 Monatsnamen[0] : Monatsnamen[n]);
}
```



## B.17 Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion **main()** durch zwei Aufrufparameter ermöglicht:

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

oder

```
int
main (int argc, char **argv)
{
 ...
}
```

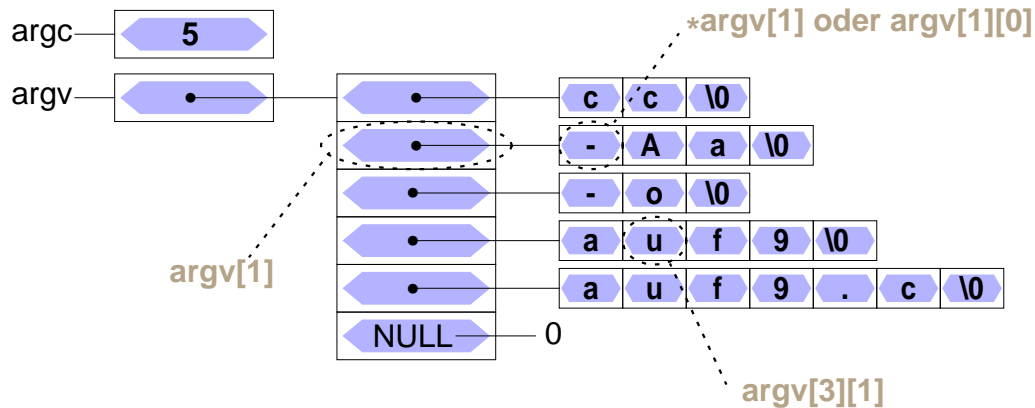
- der Parameter **argc** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **argv** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**argv[0]**)



# 1 Datenaufbau

Kommando: `cc -Aa -o auf9 auf9.c`

Datei cc.c:   
`...`  
`main(int argc, char *argv[]) {`  
`...`



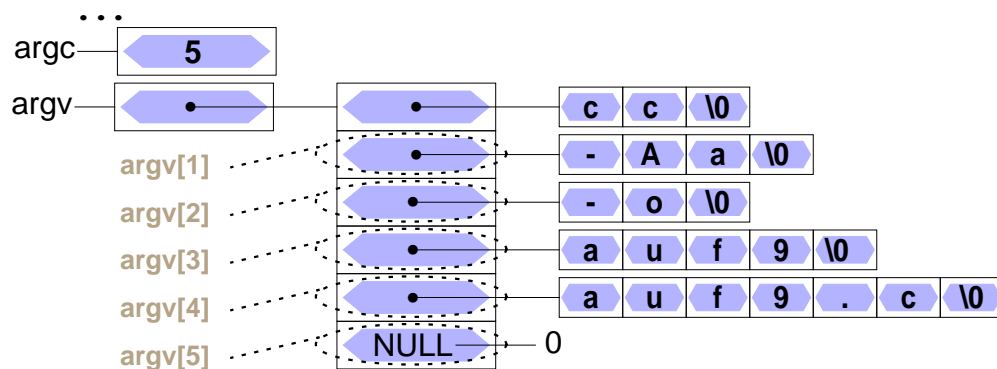
## 2 Zugriff

### Beispiel: Ausgeben aller Argumente (1)

- 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. Version



## 2 Zugriff

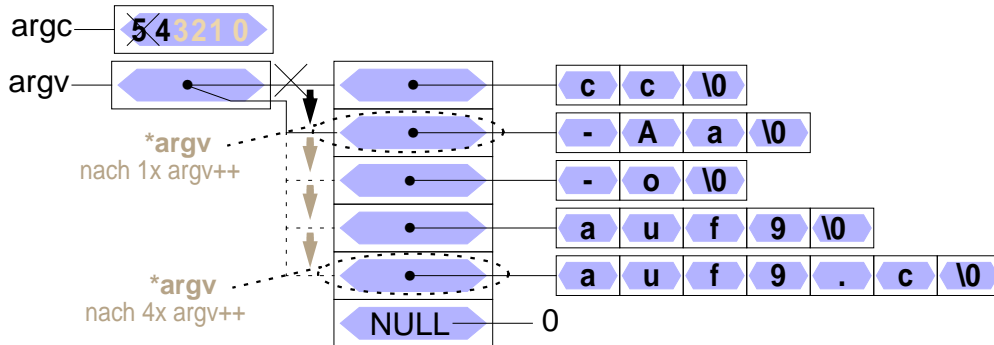
### Beispiel: Ausgeben aller Argumente (2)

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

```
int
main (int argc, char **argv)
{
 while (--argc > 0) {
 argv++;
 printf("%s%c", *argv, (argc>1) ? ' ' : '\n');
 }
 ...
}
```

linksseitiger Operator:  
erst dekrementieren,  
dann while-Bedingung prüfen  
→ Schleife läuft für argc=4,3,2,1

2. Version

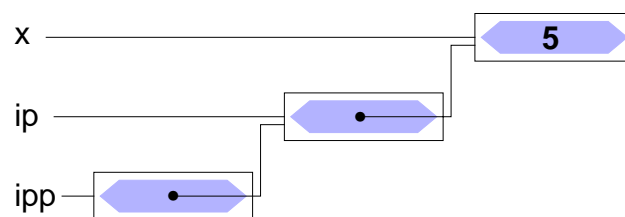


## B.18 Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist

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

int **ipp = &ip;
/* → **ipp = 5 */
```



- wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muß (z. B. swap-Funktion für Zeiger)

## B.19 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` ( $\equiv$  `int`)  
`(#include <stddef.h>!)`

- Beispiel:

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

## B.20 Explizite Typumwandlung — Cast-Operator

- C enthält Regeln für eine automatische Konvertierung unterschiedlicher Typen in einem Ausdruck (vgl. Abschnitt B.5.10)

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

◆ Beispiel:

```
char *malloc(int); /* Funktion zum dynamischen Anfordern von Speicher */
int *array; /* Zeiger auf Anfang eines dyn. anzufordernden Feldes */
int i, n = 20;

array = (int *)malloc(n * sizeof(int)); /* Feld mit n Integer-Werten */
/* dynamisch anfordern */
for (i=0; i<n; i++) array[i] = 1; /* alle Feld-Werte auf 1 setzen */
```

## B.21 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 gruppe8[35];
struct student *pstud;
pstud = &stud1; /* => pstud -> stud1 */
pstud = gruppe8; /* => pstud -> gruppe8[0] */
pstud++; /* => pstud -> gruppe8[1] */
pstud += 12; /* => pstud -> gruppe8[13] */
```

- Besondere Bedeutung zum Aufbau
  - ↳ rekursiver Strukturen

## B.21 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';`