Skript zur Vorlesung

Programmierungstechnik I

PPROGRAMMIERUNG I

1. Grundlagen

1.1 Algorithmus

1.1.1 Definition und Eigenschaften

- 820 n.Chr. Al-Khwarizmi
- Algorithmusdefinitionen: Turing, Church, Kleene, Post, Markov

Algorithmus

- Menge von Regeln A zur Überführung eines Systems von einem Anfangszustand P in einen Endzustand R

$$P\{A\}R$$

- beschreibt eine Folge von Operationen, die durch Operatoren auf Operanden ausgeführt werden

$$P\{A_1\}P_1\{A_2\}P_2 \ldots P_{k-1}\{A_k\}R$$

wobei P_i Nachbedingung der Operation A_i und Vorbedingung der Operation A_{i+1} ist.

Eigenschaften

- Minimalkomplexität

  für $A = A_1, \ldots, A_k$ ist $k \geq 2$

- Eindeutigkeit

  eine Operation A_i überführt einen Zustand P_{i-1} in genau einen Zustand P_i

- Vollständigkeit

  es existiert kein $P_j\{A_i\}P_i$ mit $j > i$

- Finitheit der Beschreibung

  der den Algorithmus beschreibende Text muß endlich sein

- Finitheit der Ausführung

  für einen $A = A_1, \ldots, A_k$ ist k endlich

Beispiel
Euklid-Algorithmus zur Berechnung de ggT
Eingangsparameter (P): m, n ganze Zahlen
Ausgangsparameter (R): z ggT
(1) bilde Divisionsrest von m/n und weise ihn r zu (r:=m mod n)
(2) falls r gleich 0, dann weise z den Wert von n (z:=n)zu und Abbruch
(3) weise m den Wert von n zu (m:=n)
(4) weise n den Wert von r zu (n:=r)
(5) weiter mit (1)

Komplexität eines Algorithmus

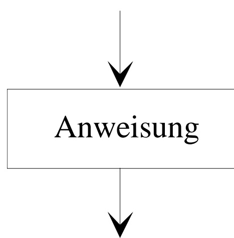
- Zeitkomplexität: Zeitaufwand für die Abarbeitung
- Raumkomplexität: Speicheraufwand für die Implementation

1.1.2 Darstellung von Algorithmen

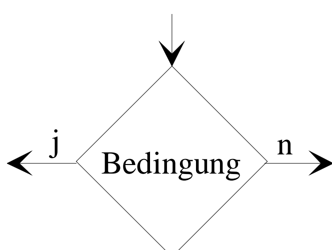
Programmablaufplan

- **Verarbeitungsfolge**

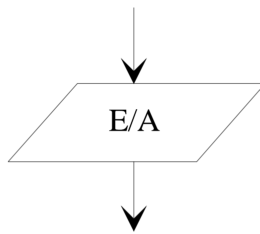
- **Anweisung**



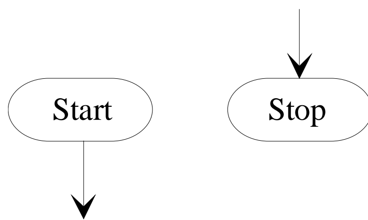
- **Bedingte Anweisung**



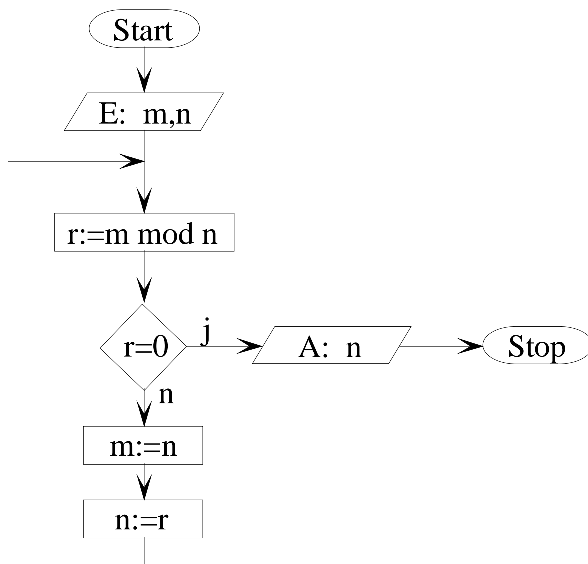
- **Ein-/Ausgabe**



- **Start/Stop**



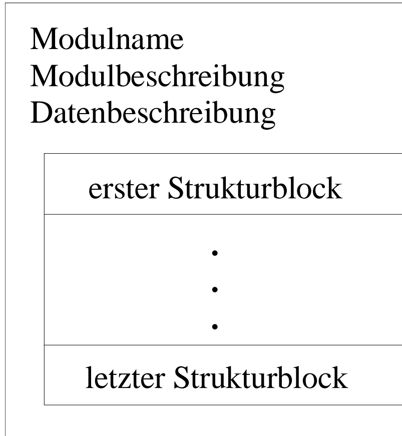
Beispiel
Euklid-Algorithmus



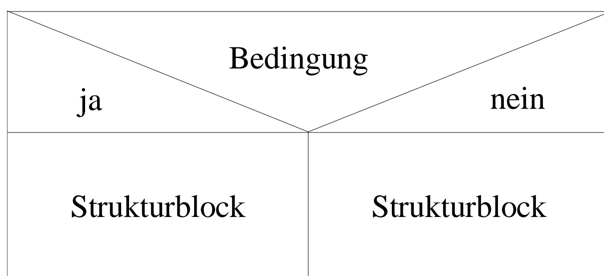
Stuktogramme

a. Elementblock

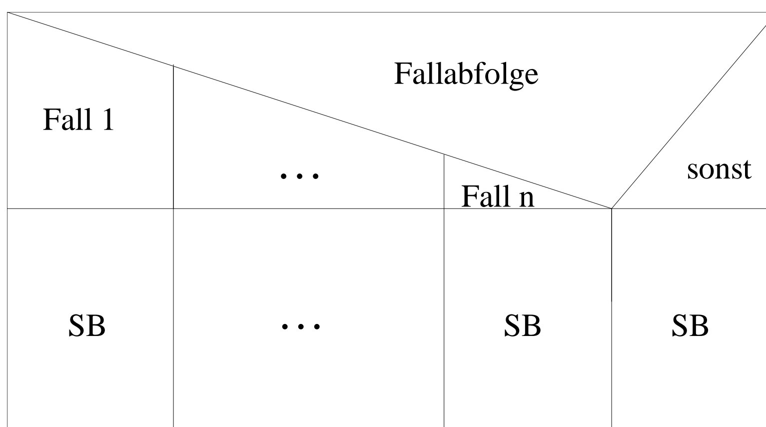
b. Modulblock



c. Selektionsblock
c.1 Bedingte Verzweigung

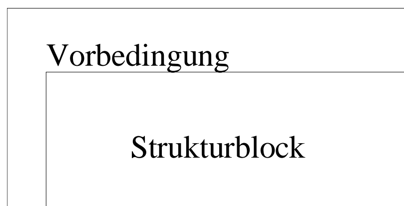


c.2 Fallunterscheidung

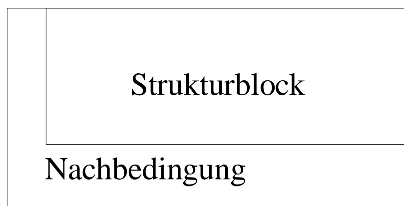


d. Iterationablock

d.1 Wiederholung mit Vorbedingung

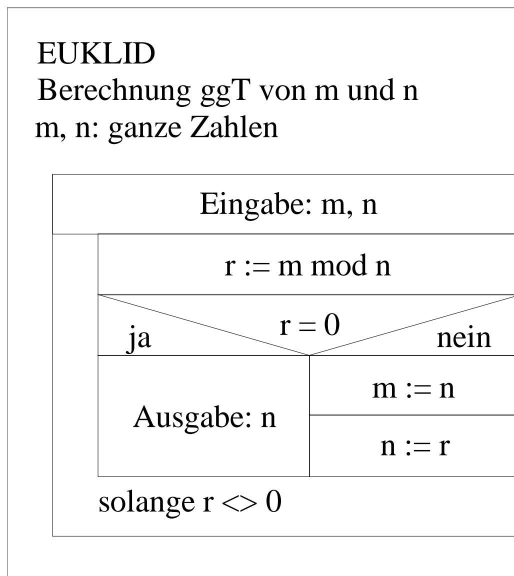


d.2 Wiederholung mit Nachbedingung



Beispiel

Euklid-Algorithmus



1.2 Sprache

- Aspekte einer Sprache

  Syntax
  Gesamtheit zwischen den Beziehungen der Elemente einer Sprache untereinander

  Semantik
  Gesamtheit der Beziehungen zwischen den Elementen einer Sprache und ihrer Bedeutung

  Pragmatik
  Gesamtheit der Beziehungen der Elemente einer Sprache, ihrer Bedeutung und den sprachverarbeitenden Systemen

1.2.1 Grammatik

Definition einer formalen Sprache

- Ein **Alphabet X** ist eine endliche Menge von Symbolen

  $$X = \{A_1, \ldots, A_n\}$$

  Die Menge X^* aller endlichen Sequenzen (Wörter), die aus X gebildet werden können, heißt freie Worthalbgruppe über X.
  Eine **Sprache L über dem Alphabet X** ist eine Teilmenge

  $$L \subseteq X^*$$

- Sind p und q Wörter aus X^*, dann ist deren Zusammensetzung (Konkatenation) pq wieder ein Wort aus X^*.

- Auf eine beliebige Sequenz $z \in X^*$ läßt sich eine Regel $p \to q$ anwenden, wenn p eine Teilzeichenkette von z ist.
  Ist $z = z_1 p z_2$ und wird $p \to q$ auf z angewendet, so erhält man folgende Überführung

  $$z = z_1 p z_2 \xrightarrow{p \to q} z' = z_1 q z_2$$

  die man als **einfache Ableitung** oder Regel bezeichnet.

- Sei P eine Menge von Regeln. Bei einer fortgesetzten Anwendung von Regeln $p \rightarrow q \in P$ ergibt sich dann eine Folge von Ableitungen

  $$z \longrightarrow z' \longrightarrow z'' \longrightarrow \ldots$$

  die abbricht, wenn in $z^{'i}$ keine Teilzeichenkette mehr auftritt, auf die sich noch eine Regel $p \rightarrow q \in P$ anwenden läßt.

- Ein **Chomsky-Grammatik** (Satzstrukturgrammatik) G ist ein Quadrupel

  $$G = (N,\ T,\ P,\ S)$$

  mit

  N - endliche Menge von Variablen (**Nichtterminale**), die ein Hilfsalphabet bilden
  S - **Startsysmbol** $S \in N$
  T - Menge von **Terminalen** $T \subseteq X^*$
  P - endliche Menge von **Erzeugungsregeln** $p \rightarrow q$

  Eine durch eine Chomski-Grammatik definierte **Sprache L** ist eine Menge

  $$L(G) = \left\{ z \;\middle|\; S \xrightarrow{G} z \;\wedge\; z \in T \right\}$$

  wobei $S \xrightarrow{G} z$ eine Folge von Ableitungen $p \rightarrow q \in P$ ist, die abbricht, wenn $z \in T$.

- eine Grammatik heißt **kontextfrei** (Chomski-2-Grammatik), wenn

  $$\forall\ (p \rightarrow q) \in P: \quad p \in N \;\wedge\; q \in \{N \cup T\}^*$$

  mit $\{N \cup T\}^*$: freie Worthalbgruppe über $T \cup N$.

Beispiel

$L = \{AA,\ AB,\ BA,\ BB\},\ X = \{A, B\}$
$S \rightarrow xx$
$x \rightarrow A$
$x \rightarrow B$

Beispiel

L = { ERGEBNIS=ZAHL,
ERGEBNIS=ZAHL+ZAHL,
ERGEBNIS=ZAHL-ZAHL,
ERGEBNIS=ZAHL+ZAHL+ZAHL,
ERGEBNIS=ZAHL+ZAHL-ZAHL, . . .}

S → ERGEBNIS=ZAHL
S → ERGEBNIS=ZAHL x
x → y ZAHL
x → x x
y → +
y → -

1.2.2 Syntaxdefinitionen

Backus-Naur-Form

- Elemente:

::=	für →
< name >	für Nichtterminal
Name	für Terminal
|	für Alternative

Beispiel

<ausdruck>	::=	Ergebnis = <zahl>
	|	Ergebnis = <zahl> <operation>
<operation>	::=	<operator> <zahl>
	|	<operator> <zahl> <operation>
<operator>	::=	+
	|	-
<zahl>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
	|	<zahl><zahl>

- Erweiterte BNF zusätzliche Elemente:

[sequenz]	für kein-/einmaliges Auftreten der Sequenz (Option)
{ sequenz }	für kein-/mehrmalige Wiederholung der Sequenz

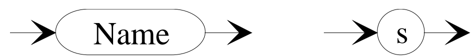
Beispiel

<ausdruck>	::=	Ergebnis = <zahl>{<operator> <zahl>}
<operator>	::=	+ | -
<zahl>	::=	<ziffer>{<ziffer>}
<ziffer>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Syntaxdiagramm

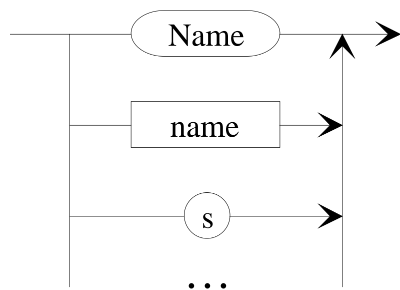
- Produktionsregel
  gerichteter Graph, Kanten sind Pfeile, Knoten sind Terminale oder Nichtterminale,
  Produktion in Richtung der Pfeile

- Terminale

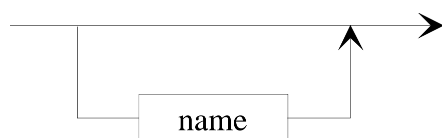


- Nichtterminale

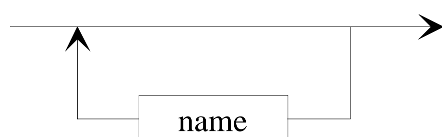
- Alternative



- Option

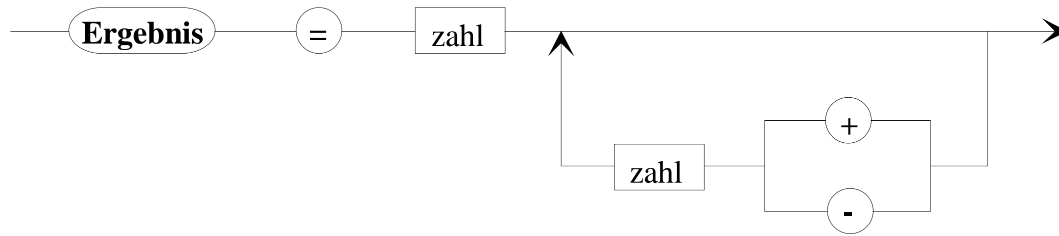


- Widerholung

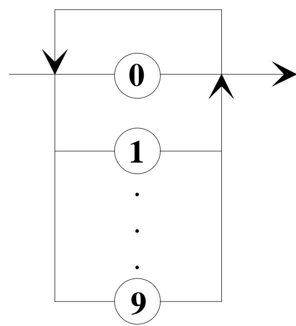


Beispiel

ausdruck



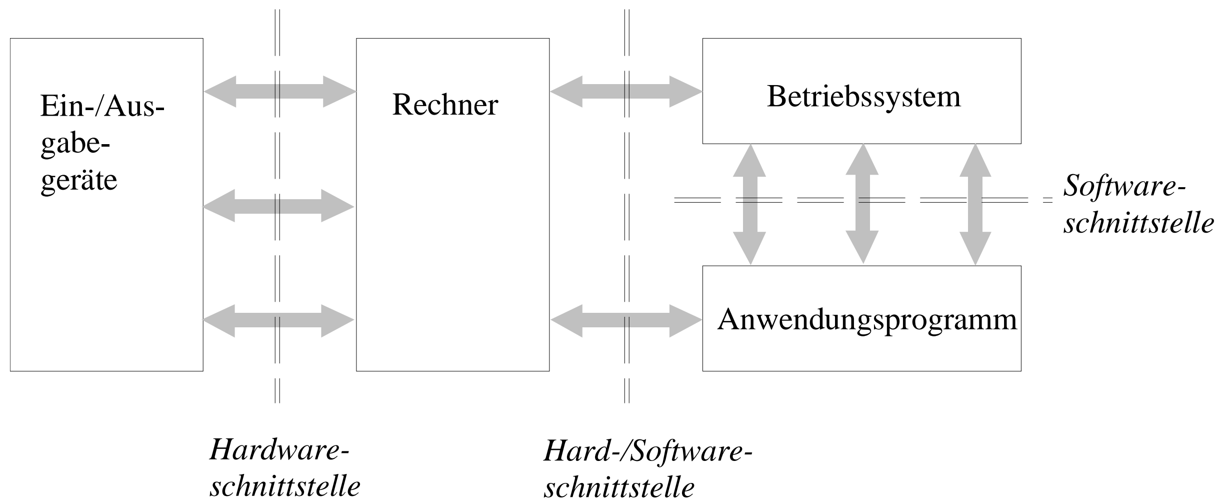
zahl



1.3 Maschine

1.3.1 Rechnerarchitektur

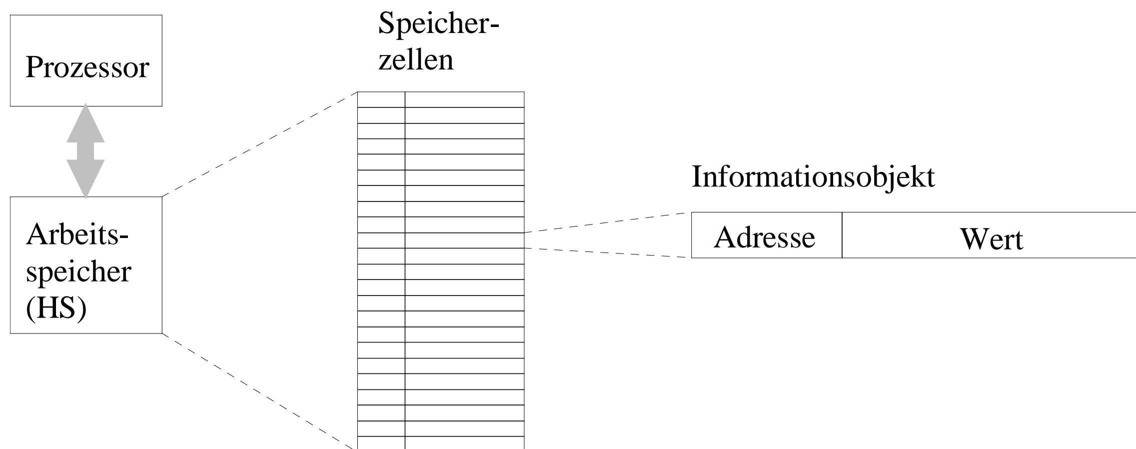
Rechneraufbau und Schnittstellen



- Von-Neumann-Architektur

Informationsstruktur

- Struktur der Informationsobjekte: Von-Neumann-Variable ::= (Adresse, Wert)



- 2 Typen der Werte von Informationsobjekten:
  - Befehl
  - Datum

- Struktur des Wertes eines Befehlsobjektes

Operation	Operand

- Operationen des Prozessors auf Befehlsobjekte:
  - Befehl aus Speicher holen
  - Befehl dekodieren/interpretieren
  - Adresse des Operanden ermitteln

- Operationen des Prozessors auf Datenobjekte:
  - Datum von der ermittelten Adresse aus dem Speicher holen
  - Datum entsprechend Befehl verarbeiten
  - Datum entsprechend Befehl in den Speicher schreiben

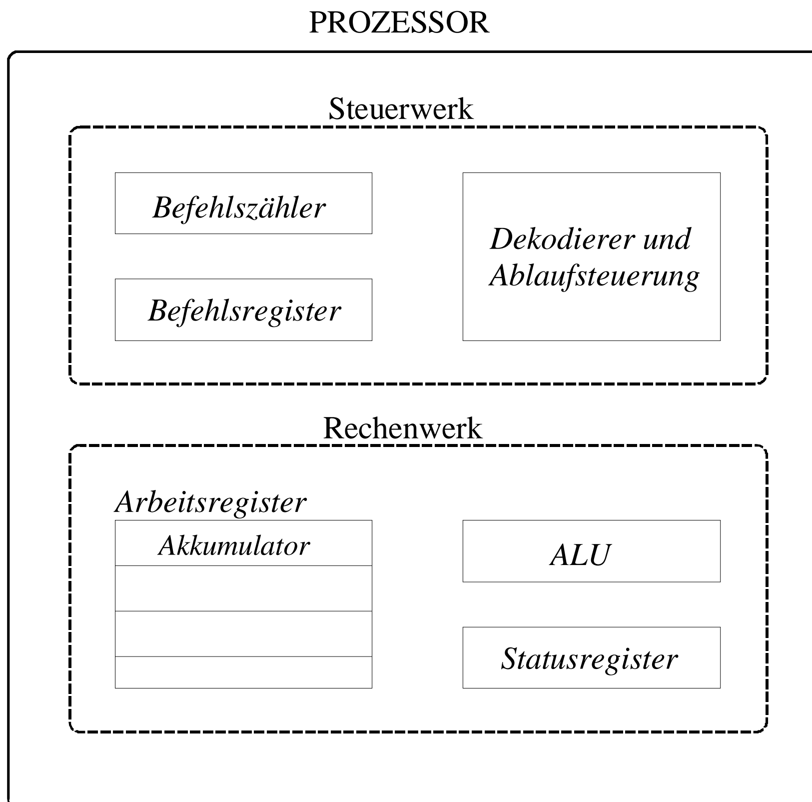
Kontrollstruktur

- Alle Informationsobjekte durch 1/0 codiert
  ⇒ Informationsobjekte der Von-Neumann-Maschine sind nicht selbstdarstellend!

- **Befehlszustand**
  das aus dem HS geholte Informationsobjekt wird als Befehl interpretiert
  Datumzustand
  das aus dem HS geholte Informationsobjekt wird als Datum interpretiert

Hardwarearchitektur

- Funktionseinheiten einer von-Neumann-Maschine:
  - E/A-Prozessor
  - Arbeitsspeicher
  - CPU (Central Processing Unit)
  - Busse (Daten-/Adreß-/Kontrollbus)



- **Steuerwerk**
  interne Ablaufsteuerung, Taktung

  – **Befehlszähler**
    enthält die Adresse des jeweils nächsten Befehls im HS
  – **Befehlsregister**
    enthält den aktuellen auszuführenden Befehl
  – **Ablaufsteuerung**
    Steuert/Taktet das Einlesen/Verarbeiten/Ausgeben von Informationsobjekten, insbes. die Befehlsinterpretation

- **Rechenwerk**
  Befehlsausführung, Verarbeitung der Operanden eines Befehls

  – **ALU (Arithmetic Logical Unit)**
    Verknüpfungslogik zwischen Befehlsoperanden (Arithmetik, Transferop., Schiebeop. ..)
  – **Arbeitsregister**
    schneller lokaler Operandenspeicher
    Akkumulator: nimmt Operanden einstelliger Befehle auf
  – **Statusregister**
    zeigt Zustand des Rechenwerks nach Operationsausführung an (Z-Bit, C-Bit, ..)

Ausführung einer Programmanweisung durch den Prozessor

- 2 Befehlstypen
  - Befehle zur Datenmanipulation: (Arithmetik, Verschiebebefehle, ..)

  - Befehle zur Programmablaufsteuerung: (Sprungbefehle, ..)

Befehlszustand

- Steuerwerk holt die Adresse des nächsten auszuführenden Befehls aus dem Befehlszähler
- Adresse wird an den HS übermittelt
- Informationsobjekt an dieser Adresse aus dem HS in das Befehlsregister eingelesen (da als Befehl interpretiert)
- Befehlszahler um 1 (nächste Adresse im HS) erhöht
- Befehl im Steuerwerk analysiert

Datumszustand

Manipulationsbefehl

- Operand des Befehls (Datumsadresse) wird an den HS übermittelt
- Informationsobjekt an dieser Adresse aus dem HS in das Rechenwerk eingelesen (da als Datum interpretiert)
- Unter Kontrolle des Steuerwerks (Befehlsinhalt) führt das Rechenwerk die Verarbeitung des Datums durch (z.B. Verknüpfung mit Akkumulatorinhalt, Ergebnis in den Akkumulator)
- Flags im Statusregister in Abhängigkeit vom Ergebnis gesetzt

Steuerbefehl

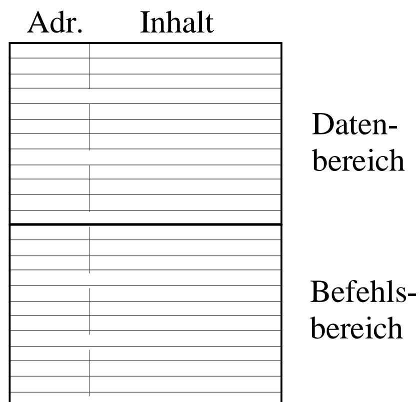
- Operand des Befehls (Befehlsadresse) wird in den Befehlszähler geladen

- weiter mit Befehlszustand

1.3.2 Maschinensprache und Assembler

- Arten von Maschinenbefehlen
  - Datentransferbefehle (LOAD, STORE, ...)
  - Steuerbefehle (JUMP, JZ, JS, ...)
  - Arithmetische/Logische Befehle (ADD, SUB, AND, ...)
  - Verschiebe-Befehle (RAL, RAR, ...)
  - Unterbrechungsbefehle (INT, ...)

- HS-Aufbau



- Befehlsbereich wird sequentiell (BZr+1) abgearbeitet, solange keine Steuerbefehle auftreten

Fiktiver Befehlssatz eines Prozessors

Code	**Mnemo**	**Semantik**
0000	STOP	Halt des Programms
0001	JUMP	Stellt Befehlszähler auf die im Operanden angegebene Speicheradresse
0010	JZ	Stellt Befehlszähler auf die im Operanden angegebene Speicheradresse, wenn letzter ALU-Befehl Ergebnis 0 hatte (Zero-Flag)
0011	JS	Stellt Befehlszähler auf die im Operanden angegebene Speicheradresse, wenn letzter ALU-Befehl ein negatives Ergebnis hatte (Sign-Flag)
0100	ADD	Addiert den Inhalt der im Operanden adressierten Speicherzelle zum Akkumulatorinhalt (Ergebnis im Akkumulator)
0101	SUB	Subtrahiert den Inhalt der im Operanden adressierten Speicherzelle vom Akkumulatorinhalt (Ergebnis im Akkumulator)
0110	LOAD	Lädt den Inhalt der im Operanden adressierten Speicherzelle in den Akkumulator
0111	STORE	Speicher den Akkumulatorinhalt in der im Operanden adressierten Speicherzelle
1000	INPUT	Eingabe eines Datums in die im Operanden adressierten Speicherzelle von einem Eingabegerät
1001	OUTPUT	Ausgabe eines Datums aus der im Operanden adressierten Speicherzelle auf ein Ausgabegerät

Beispiel

Programm, das 2 Zahlen (je 12 Bit) von einem Eingabegerät einliest und die größere auf ein Ausgabegerät ausgibt

Hauptspeicher

Adresse	Inhalt	Mnemo
0000 0000		
0000 0001		
0000 0010		
. . .		
0111 1111		
1000 0000	1000 0000 0000	INPUT 0000 0000
1000 0001	1000 0000 0001	INPUT 0000 0001
1000 0010	0110 0000 0000	LOAD 0000 0000
1000 0011	0101 0000 0001	SUB 0000 0001
1000 0100	0011 1000 0111	JS 1000 0111
1000 0101	1001 0000 0000	OUTPUT 0000 0000
1000 0110	0001 1000 1000	JUMP 1000 1000
1000 0111	1001 0000 0001	OUTPUT 0000 0001
1000 1000	0000	STOP
1000 1001		
. . .		
1111 1111		
	Op. Od.	

Assembler

- Symbolische Namen für Adressen für
  - Variablen
  - Konstanten
  - Programmzeilen
- Ausdrücke
- Makros (Programmtextersatz)
- allg. Befehlsaufbau

  [<marke>:] <operation-mnemo> [<operanden>] [<kommentar>]

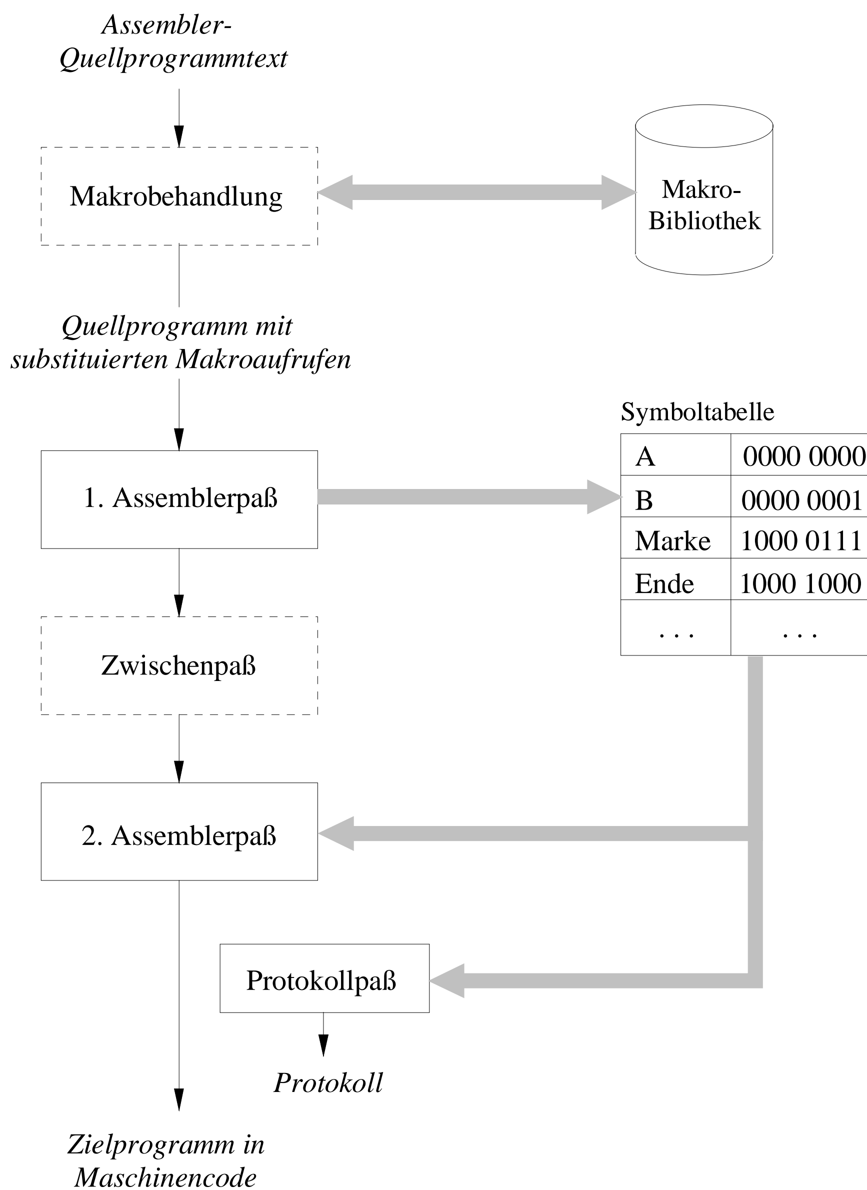
Beispiel

INPUT A

```
        INPUT B
        LOAD A
        SUB B
        JS Marke
        OUTPUT A
        JUMP Ende
Marke:  OUTPUT B
Ende:   STOP
```

- Assembler: Programm, das einen in Assemblersprache geschriebenen Quellcode in ausführbaren Maschinencode überführt (assembliert)

- 2-Paß-Assembler



2. Einführung in die C-Programmierung (C++)

2.1 C-Programme

2.1.1 Aufbau eines C-Programms

- Aufbau einer Funktion

```
[<rückgabewert>] <funktionsname> ( [<parameter>] )
{
    [<anweisungen>]
}
```

- Haupt-Funktion (Hauptprogramm): **main()**

Beispiel
Programm, das die Zeichenkette „Hallo world“ auf den BS ausgibt
Datei: hallo.c/hallo.cpp

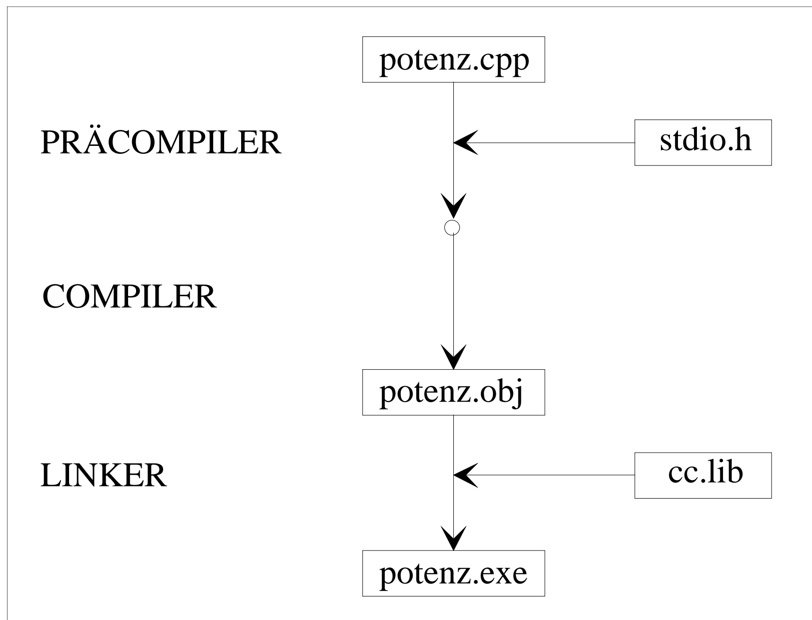
```
#include <stdio.h>
int main()
{
    printf("Hallo world");
    return 0;
}
```

Beispiel
Programm, das eine Zahl potenziert und das Ergebnis auf BS ausgibt
Datei: potenz.cpp

```
/*  Potenz der Zahl 8
    und Ausgabe
*/
#include <stdio.h>
//  Funktion
int hoch_2(int a)
{
    int b;
    b=a*a;
    return b;
}
main()
{
    int i,j;
    i=8;
    j=hoch_2(i);  // Funktionsaufruf
    printf("Ergebnis: %d", j);
    return 0;
}
```

- Konvention für Bezeichner (Namen) in C (Variablen, Funktionen ...)
  - bestehen Aus Buchstaben, Ziffern und _
  - beginnen mit Buchstaben oder _
  - Signifikante Länge 32 (Einstellung in <Source Options>)
  - Unterscheidung zwischen Groß- und Kleinbuchstaben

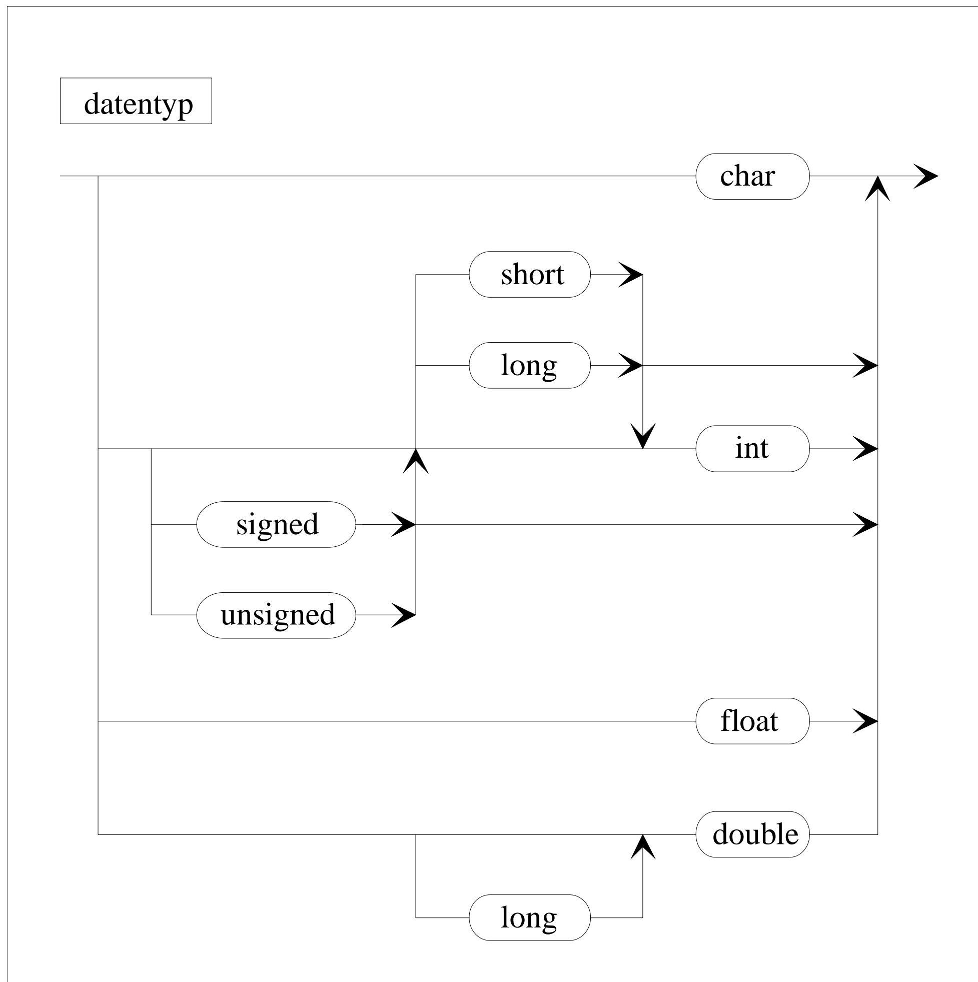
2.1.2 Übersetzungsprozeß



2.2 Einfache Datentypen

2.2.1 Datentypen und Konstanten

Datentypen



Typ	Bit	Bereich	Beschreibung
char	8	-128 ... 127	ASCII-Zeichen
unsigned int, unsigned	16	0 ... 65535	ganze Zahl
int, signed, signed int, short, short int, signed short int	16	-32768 ... 32767	ganze Zahl
unsigned long, unsigned long int	32	0 ... 4294967295	ganze Zahl
long, signed long, long int	32	-2147483648 ... 2147483647	ganze Zahl
float	32	$\pm 3.4*10^{-38}$... $3.4*10^{38}$	Gleitkommazahl, 7 Stellen
double	64	$\pm 1.7*10^{-308}$... $1.7*10^{308}$	Gleitkommazahl, 15 Stellen
long double	80	$\pm 3.4*10^{-4932}$... $3.4*10^{4932}$	Gleitkommazahl, 19 Stellen

Konstanten

- **Integerkonstanten**
  - Dezimalzahlen
  - Oktalzahlen: führende 0
  - Hexadezimalzahlen: führendes 0x oder 0X

- **Gleitkommakonstanten**
  - [<vorkommastellen>] [**.**] [<nachkommastellen>] [**e**/**E** [+/**-**] <integerexponent>]

- **Zeichenkonstanten**
  - einzelnes in Hochkomma eingeschlossenes Zeichen
  - nicht druckbare Zeichen durch vorangestellten Backslash
  - Escape-Sequenzen
    oktal: \ddd
    hexa: \xdd

- **Zeichenkettenkonstanten**
  - Folge von ASCII-Zeichen in Anführungszeichen
  - vom Compiler selbständig immer mit NULL-Byte ('\0') abgeschlossen

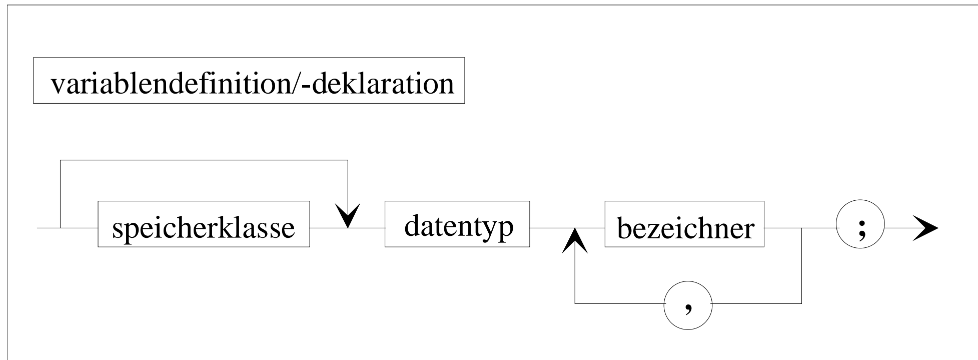
2.2.2 Variablen- und Typdefinition

- Eigenschaften:
  - Bezeichner (Name)
  - Datentyp
  - Wert (veränderlich)
  - Speicherklassenattribut

Variablendefinition

a) definierende Deklaration → **Definition**

b) nicht-definierende Deklaration → **Deklaration**



- Eigenschaften/Regeln:
  - eine Variable ist von der Stelle, an der sie definiert/deklariert worden ist bis zum Ende des Blockes gültig (lokale Variable).
  - eine Variable muß an der Stelle ihrer Verwendung gültig sein.

Speicherklassen

- **auto**
  - voreingestellte Klasse (default)
  - keine Initialisierung mit Werten
  - Speicherplatz wird nach Verlassen des Blockes freigegeben

Beispiel

```
main()
{
    int i;
    i=12;
    printf("%d", i);
    {
        int i;
        i=46;
        printf("%d", i);
    }
    printf("%d", i);
}
```

- **static**
  - vom Compiler mit 0 initialisiert
  - Speicher nach Verlassen des Blockes nicht freigegeben: Werte bleiben erhalten

Beispiel

```
while(...)
{
```

```
    ...
    {
        static int k;
        printf("%d", k);
        k=k+1;
    }
    ...
}
```

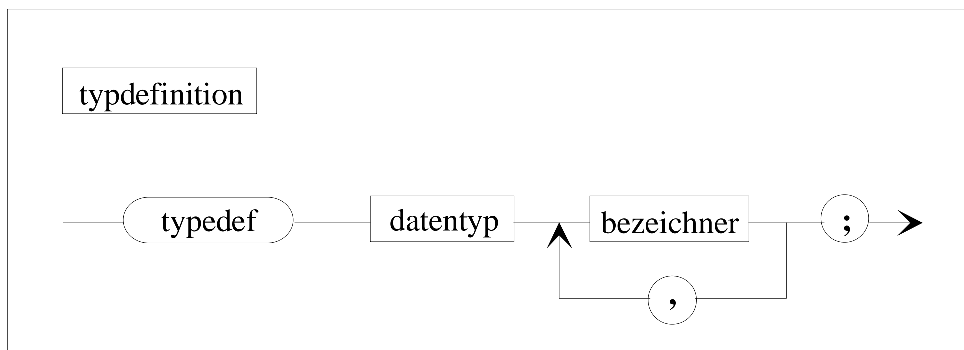
- **register**
  - wird vom Compiler nach Möglichkeit im Prozessorregister gehalten
  - impliziert auto

- **extern**
  - Variablendeklaration, keine Speicherplatzzuweisung
  - wird benötigt um globale Variablen (aber auch Funktionen) an Stellen gültig zu machen, an denen sie sonst nicht gültig wären

Beispiel

```
#include <stdio.h>
f()
{
    extern int k;
    printf("%d", k);
}
int k;
main()
{
    k=123;
    f();
}
```

Typdefinition

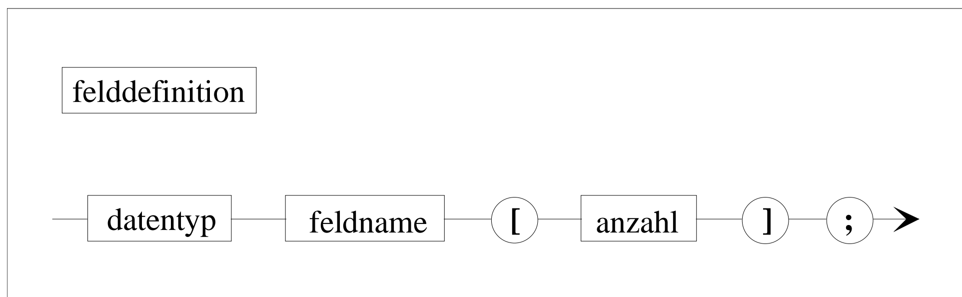


Beispiel

```
typedef char zeichen;
typedef float zeit, strecke;
```

```
...
zeichen c1, c2;
c1='X';
zeit t;
t=22.897
```

Einfache Felddefinition



Beispiel

```
char f[10];
static float preis[20];
f[0] = 'A';
f[1] = 'B';
f[2] = 'C';
```

2.2.3 Wertezuweisung

- einfachste Form des Zuweisungsausdrucks:

  <lvalue> = <ausdruck>

Beispiel

Mehrfachzuweisung

```
int i, j, k;
i = j = k = 10;
```

Initialisierung

- Expizite Wertezuweisung bei der Variablendefinition

- **einfache Variable**
  int i=12;

```
float pi=3.14, tau=44.3;
```

- **Felder**

```
int f[10]={1,2,3,4};
int f[ ]={1,2,3,4};
```

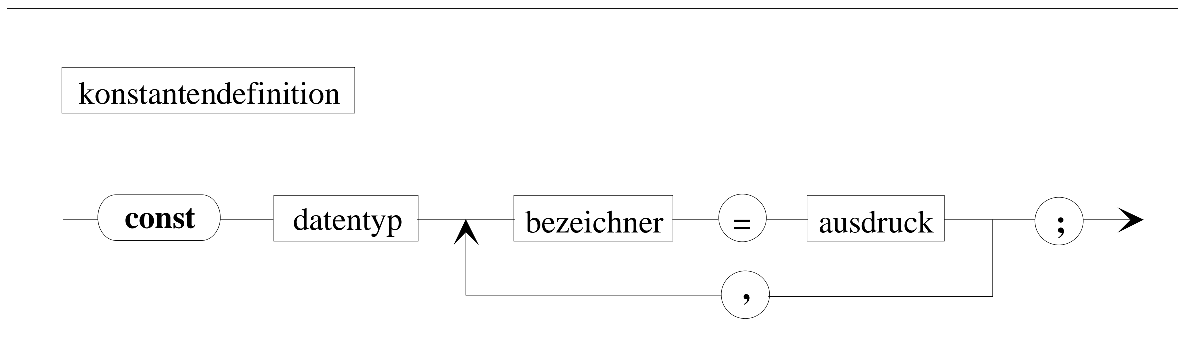
- **Zeichenketten**

```
char zk[10]={'H','a','l','l','o','\0'};
char zk[10]="Hallo";
char zk[ ]="Hallo";
```

Beispiel

```
while(...)
{
    static int i=7;
    i=i-1;
    printf("%d", i);
}
```

2.2.4 Konstantendefinition



Beispiel

```
const float pi=3.14;
const float piquadrat=9.87;
const char Gruss[ ]="Hallo";
pi=3.1415;
```

2.3 Ein-/Ausgabe

- Standard-E/A-Funktionen, Konstante und Variable in stdio.h deklariert

2.3.1 Formatierte Ein-/Ausgabe

Ausgabefunktion

- **int printf(** <formatzeichenkette> **[,** <argumentliste>] **)**

<formatzeichenkette> ::= **"** <zeichenfolge> **"**

<zeichenfolge> ::= <normales zeichen>
| <konvertierungsanweisung>
| <zeichenfolge><zeichenfolge>

<konvertierungsanweisung> ::= **%**[<breite>][**.**<präzision>]<typ>

<argumentliste> ::= <ausdruck>
| <ausdruck> **,** <argumentliste>

- <breite> bestimmt die minimale Anzahl auszugebender Zeichen für ein Argument
- <präzision> bestimmt die maximale Anzahl von Stellen nach dem Dezimalpunkt
- <typ> bestimmt den auszugebenden Zeichentyp:

Typ	**Argument**	**Ausgabe**
d	Integer	als Dezimalzahl
o	Integer	als Oktalzahl
x	Integer	als Hexadezimalzahl
f	Gleitkomma	als Gleitkommazahl ohne Exponent
e, g	Gleitkomma	als Gleitkommazahl mit Exponent
c	Zeichen	als einzelnes Zeichen
s	Zeichenkette	als Zeichenkette bis '\0'

- printf liefert bei fehlerfreier Ausführung die Anzahl der ausgegebenen Bytes, sonst EOF

Beispiel

```
char s[ ] = ", wie geht\nes Dir?\n\n";
double x = 123.45678;
int i = 40;
printf("Hallo");
printf("%s",s);
printf("%f\n%15.8f\n%e", x, x, x);
printf("\n\nDezimal:  %d    Oktal:  %o    Hexa:  %x", i+5, i+5, i+5);
```

Eingabefunktion

- **int scanf(** <formatzeichenkette> [**,** <adressenliste>] **)**

<formatzeichenkette> ::= **"** <zeichenfolge> **"**

<zeichenfolge> ::= <normales zeichen>
| <formatierungsanweisung>
| <zeichenfolge><zeichenfolge>

<formatierungsanweisung> ::= **%**[<breite>]<typ>

<adressenliste> ::= <adressausdruck>
| <adressausdruck> **,** <adressenliste>

- <breite> bestimmt die maximal in die zugehörige Variable einzulesende Zeichenzahl.
- <typ> bestimmt, welche Art von Eingabe erwartet wird

Typ	**Adresse**	**Eingabe**
d	Zeiger auf int	als Dezimalzahl
o	Zeiger auf int	als Oktalzahl
x	Zeiger auf int	als Hexadezimalzahl
f, e, g	Zeiger auf float	als Gleitkommazahl mit/ohne Exponent
c	Zeiger auf char	als einzelnes Zeichen
s	Zeiger auf char-Feld	als Zeichenkette bis whitespace, abschließendes '\0' wird vom System vergeben

- scanf liefert bei fehlerhafter Ausführung oder beim Lesen über ein Datei-/Stringende EOF.

Beispiel

```
char f1[10], f2[10];
scanf("%s%s", f1, f2);
printf("%s\n%s\n",f1, f2);
```

Beispiel

```
char f1[10], f2[10];
```

```
scanf("%4s%6s", f1, f2);
printf("%s\n%s\n",f1, f2);
```

Beispiel

```
char c;
float x, y;
scanf("%c%f%f", &c, &x, &y);
printf("\n%c\n%e\n%f\n", c, x, y);
```

Beispiel

```
int t, m, j;
printf("Datum:  ");
scanf("%d/%d/%d", &t, &m, &j);
printf("Heute ist der %d.%d.200%d\n", t, m, j);
```

2.3.2 Zeichenweise Ein-/Ausgabe

- **int putchar(int c)**
  - schreibt ein Zeichen in den Ausgabestrom stdout
  - liefert bei fehlerfreier Ausführung das Zeichen, sonst EOF zurück

- **int getchar( )**
  - liest ein Zeichen vom Eingabestrom stdin
  - liefert bei fehlerfreier Ausführung das Zeichen, bei Fehler oder Dateiende EOF zurück

Beispiel

Programm KOPIE, das ein Zeichen aus dem Eingabestrom in den Ausgabestrom kopiert

```
#include <stdio.h>
main()
{
    char c;
    c = getchar( );
    putchar(c);
}
```

Beispiel

Programm KOPIE, das bis zum EOF Zeichen vom Eingabe- in den Ausgabestrom kopiert

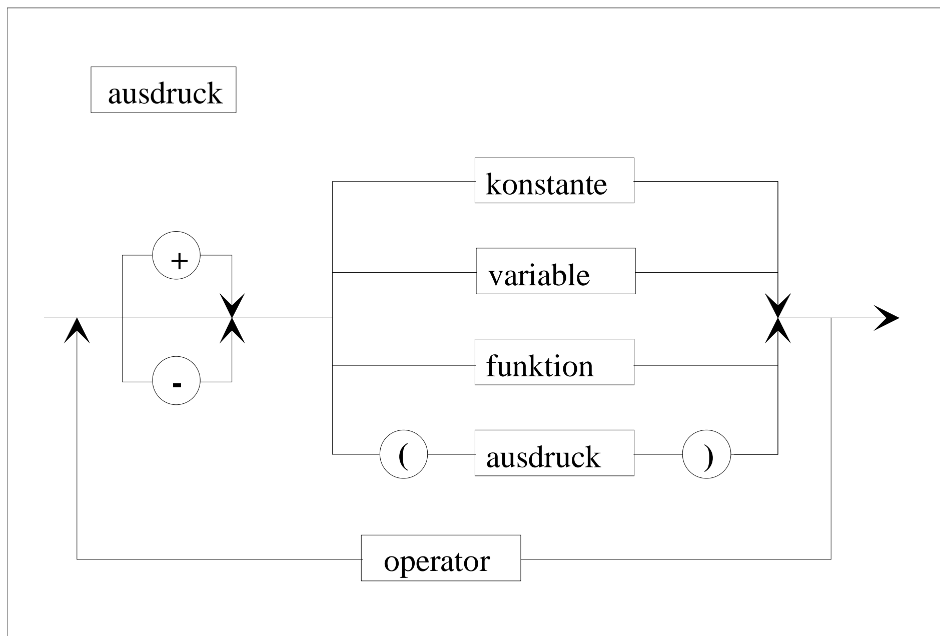
```
#include <stdio.h>
main()
{
    char c;
    c = getchar( );
    while(c!=EOF)
    {
        putchar(c);
        c=getchar( );
    }
}
```

- Möglichkeit der Verwendung des KOPIE-Programms auf Betriebssystemebene (DOS):
  - Ein-/Ausgabe-Umlenkung: **c:>**program <inputfile >ouputfile
  - Pipelining: **c:>**program1 | program2

Beispiel

```
c:> KOPIE <datei1.txt
c:> KOPIE >datei2.txt
c:> KOPIE <datei1.txt >datei2.txt
c:> type datei1.txt | KOPIE > datei2.txt
```

2.4 Operatoren und Ausdrücke



- Ein Ausdruck <ausdruck><operator><ausdruck> erhält im Ergebnis der Operation als Ganzes den Ergebniswert (siehe Mehrfachzuweisung)

a) **Unäre Operatoren:** + **-**

b) **Arithmetische Operatoren:** + **-** * / **%**

c) **Vergleichsoperatoren:** < <= > >= = = **!=**

– ein Vergleichsausdruck hat im Ergebnis ausschließlich den integer-Zahlenwert 1 (log. wahr) oder 0 (log. falsch)

d) **Logische Operatoren:**

! - nicht (unär)
|| - oder
&& - und

e) **Bitoperatoren:**

~	- Komplement (unär)
&	- und
|	- oder
^	- exklusives oder
>>n	- Rechtsverschiebung um n bit
<<n	- Linksverschiebung um n bit

– n ist integer-Ausdruck
– bei Verschiebeoperationen wird das freiwerdende Bit mit 0 aufgefüllt

Beispiel

Programm: Es ist eine Zahl x von der Bit-Position p an um n Stellen (nach Rechts) zu komplementieren. Das äußerste rechte Bit hat die Position 0.

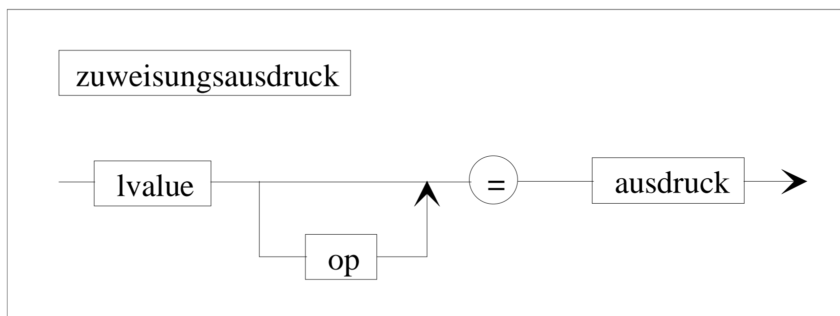
z.B. x = 0123
p = 4
n = 2

Position	7.	6.	5.	4.	3.	2.	1.	0.
x =	0	1	0	1	0	0	1	1
$x^{Ergebnis}$ =	0	1	0	0	1	0	1	1

```
#include <stdio-h>
main()
{
    unsigned x=0123;
    int p=4, n=2;
    x = x ^ ( ~ ( ~ 0 << n ) << ( p + 1 - n ) );
    printf("%o", x);
}
```

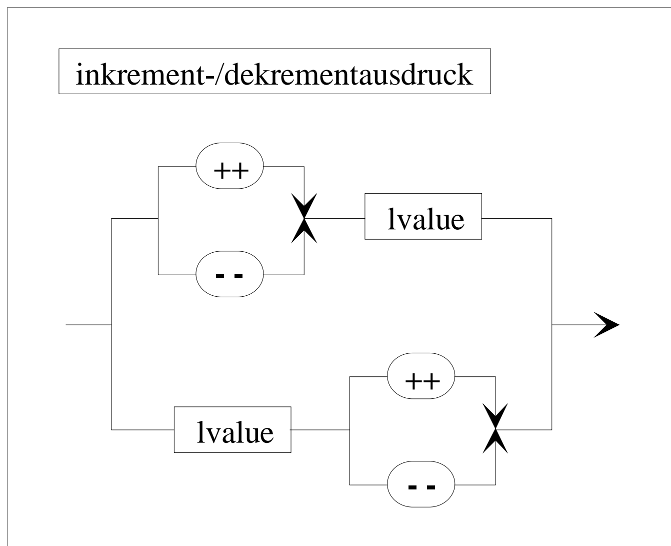
f) **Zuweisungsoperator:** =

- allg. Form des Zuweisungsausdrucks:



– Semantik: <lvalue> <op> = <ausdruck> ⇔ <lvalue> = <lvalue> <op> <ausdruck>
– mögliche **op:** + - / * % & | ^ >>n <<n

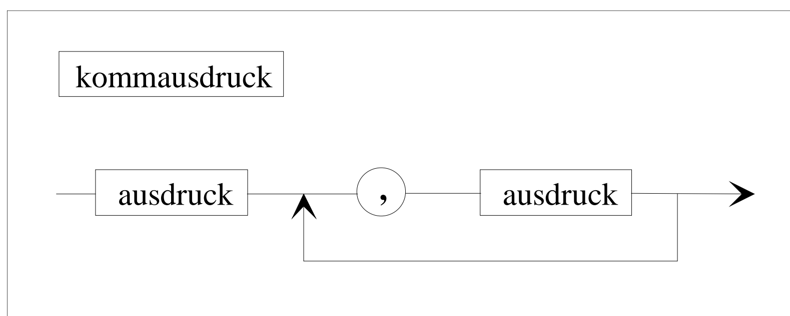
g) **Inkrement-/Dekrementoperator:** ++ - -



- Semantik: ++/-- <lvalue> ⇔ <lvalue> ++/-- ⇔ <lvalue> = <lvalue> +/- 1

- Präfixnotation ++/-- <lvalue>: Inkrementierung/Dekrementierung bevor der Wert von lvalue weiterverwendet wird
- Postfixnotation <lvalue> ++/--: Inkrementierung/Dekrementierung nachdem der Wert von lvalue weiterverwendet wird

h) **Kommaoperator:** ,



- die einzelnen Ausdrücke werden von links nach rechts abgearbeitet
- der gesamte Kommaausdruck erhält den Wert des des rechten Ausdrucks

i) **sizeof-Operator:** **sizeof (** <datentyp> **)**

- der sizeof-Ausdruck erhält als Wert den für den angegebenen Datentyp erforderlichen Speicherplatz in Byte

j) **typecast-Operator:** **(** <datentyp> **)** <ausdruck>

– Der Datentyp des Wertes des Ausdrucks wird in den angegebenen Datentyp umgewandelt (**explizite Typumwandlung**)

Implizite Typumwandlung

- Vom Compiler werden vor der Auswertung eines arithmetischen Ausdrucks
  1. alle darin vorkommenden char-, short- und enum-Datentypen in den int-Typ umgewandelt.
  2. alle darin vorkommenden float-Datentypen in den double-Typ umgewandelt
  3. alle Datentypen in den im Ausdruck auftretenden höchstwertigen Typ umgewandelt.

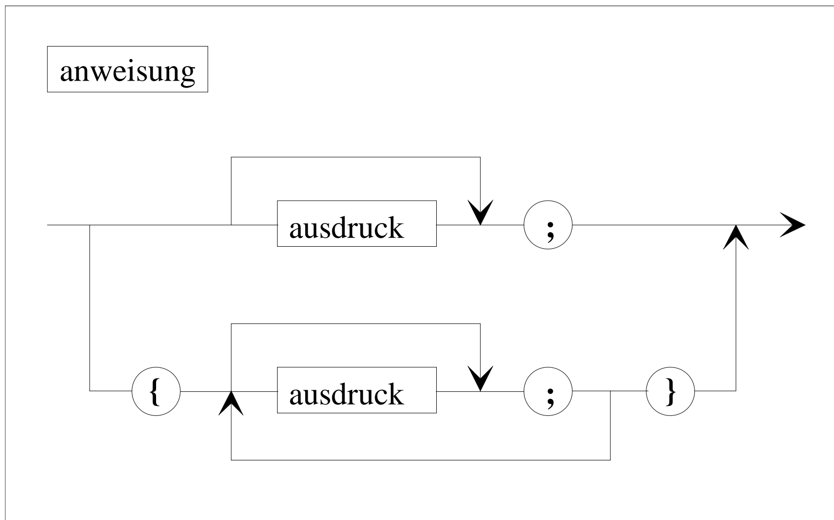
- Wertigkeit der Datentypen (größerer Wertebereich = höherwertig)
  – long double
  – double
  – unsigned long int
  – long int
  – unsigned int

- Explizite Typumwandlung durch gezielte Anwendung des typecast-Operators

Bindungsstärke und Assoziativität von Operatoren

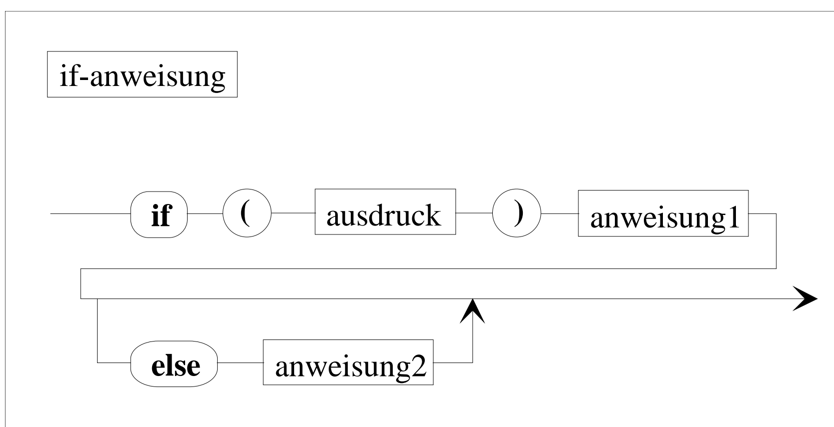
<table>
<tr><th>Operator</th><th>Assoziativität</th></tr>
<tr><td>( ) [ ] -> .</td><td>links nach rechts</td></tr>
<tr><td>! ~ ++ -- + - * & (typ) sizeof new delete</td><td>rechts nach links</td></tr>
<tr><td>* / %</td><td>links nach rechts</td></tr>
<tr><td>+ -</td><td>links nach rechts</td></tr>
<tr><td><< >></td><td>links nach rechts</td></tr>
<tr><td>< <= > >=</td><td>links nach rechts</td></tr>
<tr><td>== !=</td><td>links nach rechts</td></tr>
<tr><td>&</td><td>links nach rechts</td></tr>
<tr><td>^</td><td>links nach rechts</td></tr>
<tr><td>|</td><td>links nach rechts</td></tr>
<tr><td>&&</td><td>links nach rechts</td></tr>
<tr><td>||</td><td>links nach rechts</td></tr>
<tr><td>?:</td><td>rechts nach links</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>rechts nach links</td></tr>
<tr><td>,</td><td>links nach rechts</td></tr>
</table>

2.5 Steueranweisungen



2.5.1 Selektionen

if-Anweisung



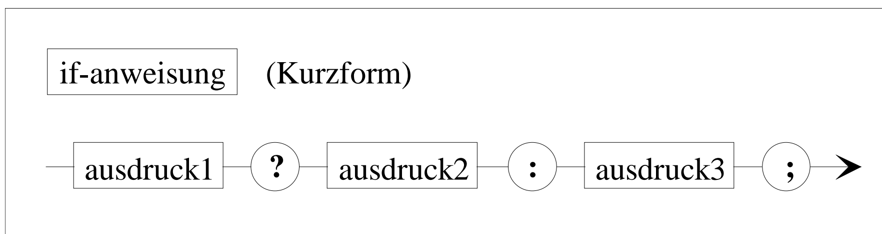
- ist der ausdruck wahr, wird nur anweisung1 abgearbeitet
- ist ausdruck falsch, wird anweisung2 abgearbeitet (falls vorhanden)

Beispiel

```
int x=2;
if(x>0)
    printf("%d", x);
if(x)
    printf("%d", x);

int y=5;
```

```
if(x-y)
    if(x>y)
        printf("Maximum: x");
    else
        printf("Maximum: y");
else
    printf("Gleiche Werte");

int i=1;
if(i--)
    printf("i ungleich 0: %d", i);
else
    printf("i gleich 0: %d", i);
```

- Besondere Kurzform der if-Anweisung durch Operator **?:**

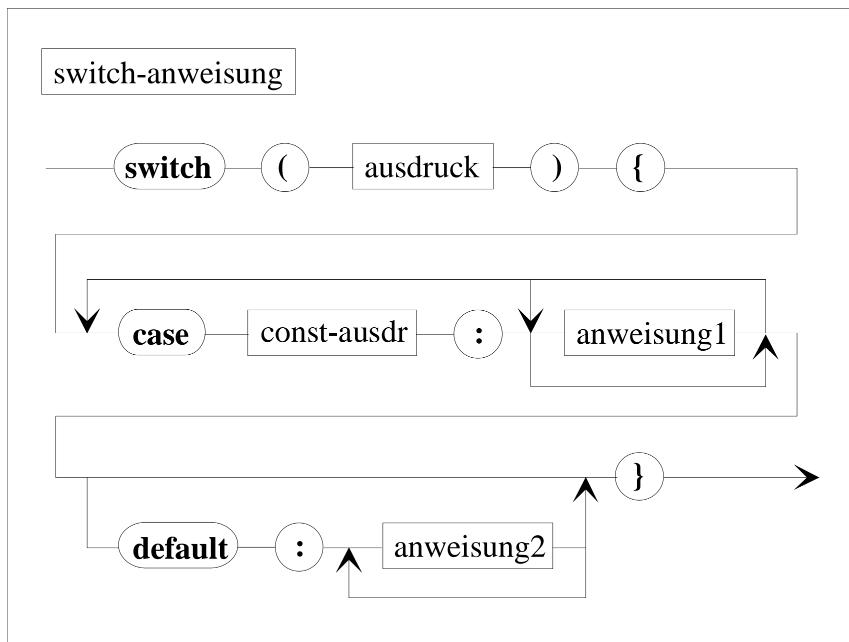


- Semantik: **if(** <ausdruck1>)
  <ausdruck2>**;**
  else
  <ausdruck3>**;**

Beispiel

```
int i=3, j=8;
(i>j) ? printf("%d", i) : printf("%d", j);
int max = (i>j) ? i : j;
int abs;
(i>j) ? (abs=i-j, printf("%d", abs)) : (abs=j-i, printf("%d", abs));
```

switch-Anweisung



- switch vergleicht den Wert von ausdruck der Reihenfolge nach mit jedem const-ausdr der case-Zweige
- sobald ein Wert übereinstimmt werden alle zu dem case-Zweig gehörenden Anweisungen und alle Anweisungen der darauffolgenden case-Zeige (einschließlich des möglichen default-Zweiges) abgearbeitet
- stimmt kein Wert überein, werden die Anweisungen des default-Zweiges (falls vorhanden) abgearbeitet

Beispiel

```
char c=getchar();
switch(c)
{
    case 'a'  :  printf("A");
    case 'b'  :  printf("B");
    default   :  printf("C");
}
```

break-Anweisung

break ;

- darf nur innerhalb der switch-Anweisung oder den while-, do-while- und for-Schleifen verwendet werden.
- break beendet die einschließende switch-Anweisung bzw. die (innerste) einschließende Schleife

Beispiel

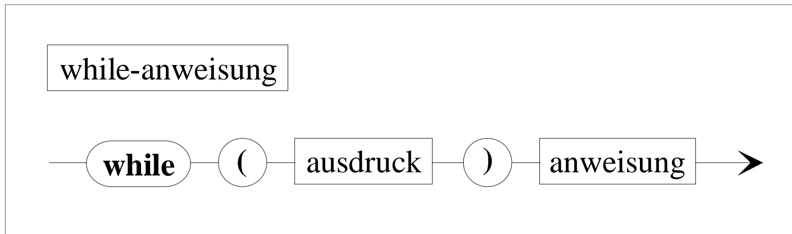
```
char c=getchar();
switch(c)
{
    case 'a'   :  printf("A");
                  break;
    case 'b'   :  printf("B");
                  break;
    default    :  printf("C");
}
```

Beispiel

```
char c=getchar();
switch(c)
{
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':   printf("Ziffer eingegeben"); break;
    default                                         :   printf("Zeichen eingegeben");
}
```

2.5.2 Iteration

while-Anweisung



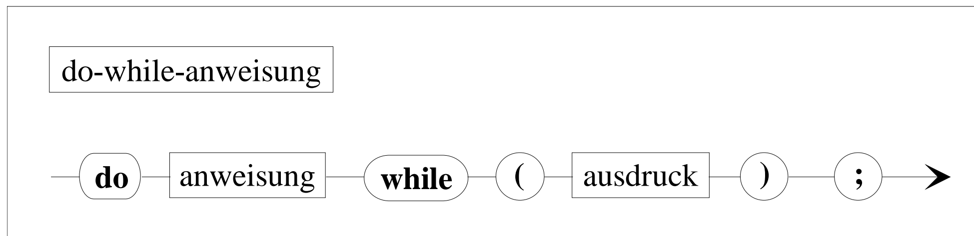
- solange ausdruck wahr ist, wird anweisung wiederholt abgearbeitet
- vor der ersten Abarbeitung wird ausdruck überprüft

Beispiel

Berechnung der Fakultät einer eingegebenen Zahl

```
#include <stdio.h>
main()
{
    int n, erg=1;
    printf("\nn = ");
    scanf("%d", &n);
    while(n)
        erg*=n--;
    printf("n!= %d",erg);
    return 0;
}
```

do-while-Anweisung



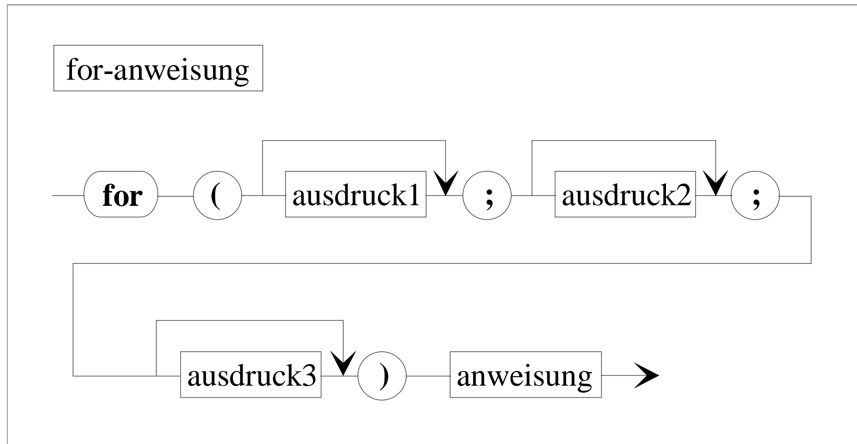
- solange ausdruck wahr ist, wird anweisung wiederholt abgearbeitet
- vor der ersten Überprüfung von ausdruck wird anweisung einmal abgearbeitet

Beispiel

Berechnung des ggT nach EUKLID

```
#include <stdio.h>
main()
{
    int m,n,rest;
    printf("\nm = ");
    scanf("%d", &m);
    printf("\nn = ");
    scanf("%d", &n);
    do
    {
        if(!(rest=m%n))
            printf("\n\nggT ist %d",n);
        m=n;
        n=rest;
    }
    while(rest);
    return 0;
}
```

for-Anweisung



- ausdruck1: Anfangszustand
- Semantik:

  <ausdruck1>**;**
  while(<ausdruck2>)
  {
      <anweisung>
      <ausdruck3>**;**
  }

Beispiel

```
for(i=0; i<10; i++)
    scanf("%d", f[i]);
```

continue-Anweisung

continue ;

- bei while- und do-while-Schleifen: sofort weiter mit Überprüfung der Bedingung
- bei for-Schleife: sofort weiter mit Zustandsänderung (ausdruck3)

Beispiel

Berechnung des Mittelwertes aller nichtnegativen Zahlen eines int-Feldes

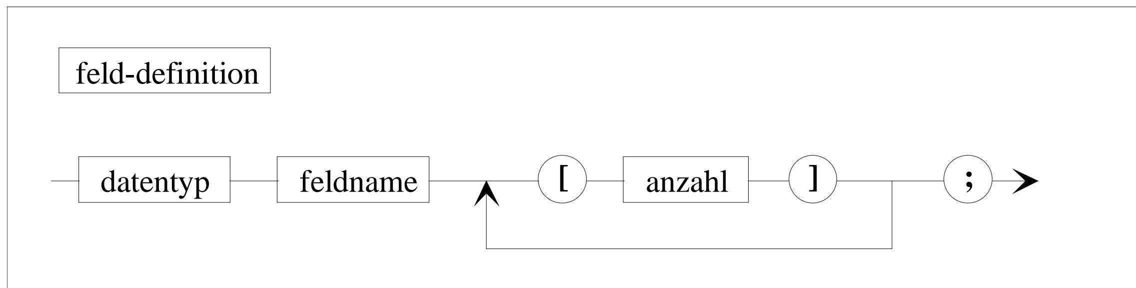
```
#include <stdio.h>
main()
{
    int f[10]={2, -5, 17, 3, -22, 37, 9, -81, 52, -39};
    int summe=0, anzahl=0;
    for(int i=0; i<10; i++)
    {
        if(f[i]<0) continue;
        summe+=f[i];
        anzahl++;
    }
    if(anzahl)
        printf("%f", (float)summe/anzahl);
}
```

2.6 Komplexe Datentypen

2.6.1 Felder

- Felder: Zusammenfassungen von Elementen gleichen Typs

Felddefinition



Zugriff auf Feldelemente durch Index-Notation:

<feldname> [<index1>][<index2>][<index3>]...

- bei 2-dimensionalen Feldern (Matrix):

<feldname> [<zeile>][<spalte>]

- Initialisierung erfolgt zeilenweise

- Initialisierung kann durch Klammerung gesteuert werden

Beispiel

Sortieren eines 10-elementigen Feldes von Integer-Zahlen ("British Museum Method")

```
#include <stdio.h>
main()
{
    const int laenge=10;
    int i, j, tmp, feld[laenge];
    for(i=0;i<laenge;i++)
        scanf("%d", feld[i]);
```

```
    for(i=0;i<laenge;i++)
        for(j=i;j<laenge;j++)
            if(feld[i]<feld[j])
            {
                tmp=feld[i];
                feld[i]=feld[j];
                feld[j]=tmp;
            }

    for(i=0;i<laenge;i++)
        printf("\n%d", feld[i]);
}
```

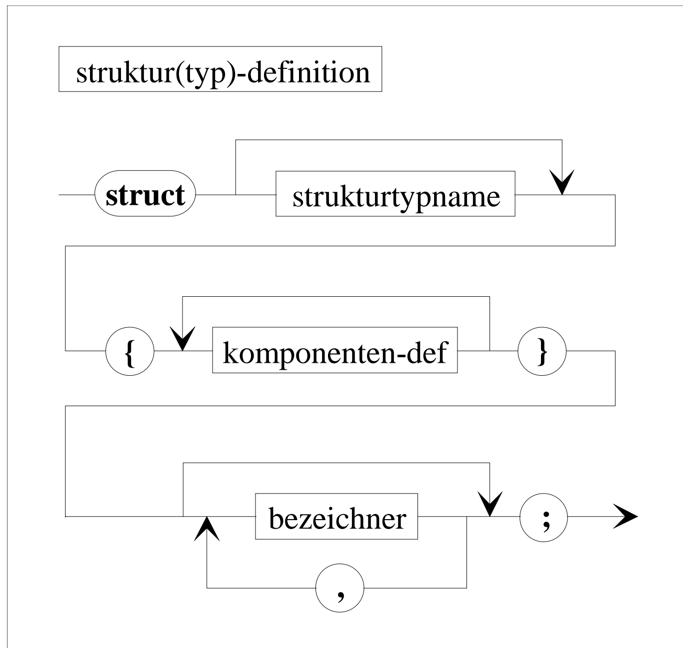
Beispiel
Erzeugen eines Feldes, dessen Elemente sich aus der Summe der Element einer jeden Zeile einer Matrix ergeben: $f_i = \sum_{j=0}^{jmax} m_{ij}$

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int imax=5, jmax=7;
    int i, j, m[imax][jmax], f[imax];
    for(i=0;i<imax;i++)
        for(j=0;j<jmax;j++)
            m[i][j]=random(100);

    for(i=0;i<imax;i++)
    {
        f[i]=0;
        for(j=0;j<jmax;j++)
            f[i]+=m[i][j];
    }

    for(i=0;i<imax;i++)
        printf("\n%d", f[i]);
}
```

2.6.2 Strukturen und Unions

Strukturen

- Zusammenfassungen von Elementen (Komponenten) unterschiedlichen Typs



Strukturtypdefinition (Strukturvereinbarung)

```
struct <typname>
{
    <komponentendefinitionen>
};
```

- es wird ein neuer, komplexer Datentyp vereinbart, keine Speicherallocation

```
typedef struct
{
    <komponentendefinitionen>
} <typname>;
```

Definition von Strukturvariablen

- es wird Speicherplatz alloziert, der sich aus der Summe der Speicheranforderungen für die einzelnen Komponenten ergibt

- **außerhalb einer Strukturdefinition**

  <strukturtypname> <bezeichner> [**,** <bezeichner>]

  oder

  struct <strukturtypname> <bezeichner> [**,** <bezeichner>]   (alt)

- **innerhalb einer Strukturdefinition**

  struct
  {
      <komponentendefinitionen>
  } <bezeichner>**;**

Zugriff auf Komponenten (Komponentenbezeichner)

<strukturvariable> **.** <komponente> [**.** <komponeten ...]

Beispiel

```
struct
{
    char Name[7];
    int Alter;
} Mitarbeiter[3];
for(int i=0; i<3; i++)
    scanf("%s%d", Mitarbeiter[i].Name, &Mitarbeiter[i].Alter);
```

Beispiel

```
struct
{
    char Name[7];
    struct
    {
        char Ort[10];
        char Straße[7];
        int Nr;
    } Wohnung;
    int Alter;
} Person;
scanf("%s%d%s%s%d", Person.Name, &Person.Alter, Person.Wohnung.Ort,
            Person.Wohnung.Straße, &Person.Wohnung.Nr);
```

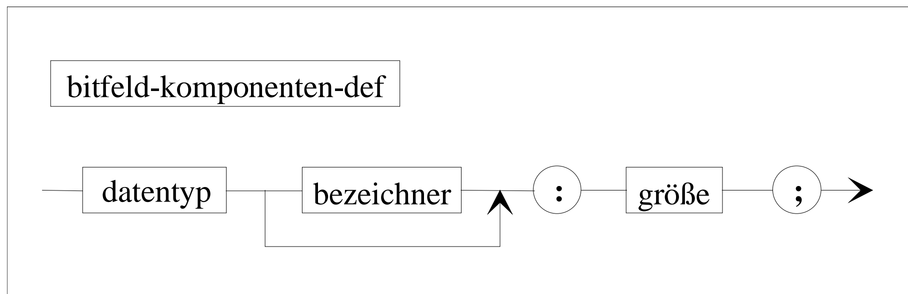
analog

```
struct wohnort
{
    char Ort[10];
    char Straße[7];
    int Nr;
};
struct
{
    char Name[7];
    wohnort Wohnung;
    int Alter;
} Person;
```

- sind Strukturvariablen vom gleichen Typ, können die Komponenten insgesamt zugewiesen werden

Bitfelder

- Strukturen, auf deren Komponenten bitweise zugegriffen werden kann



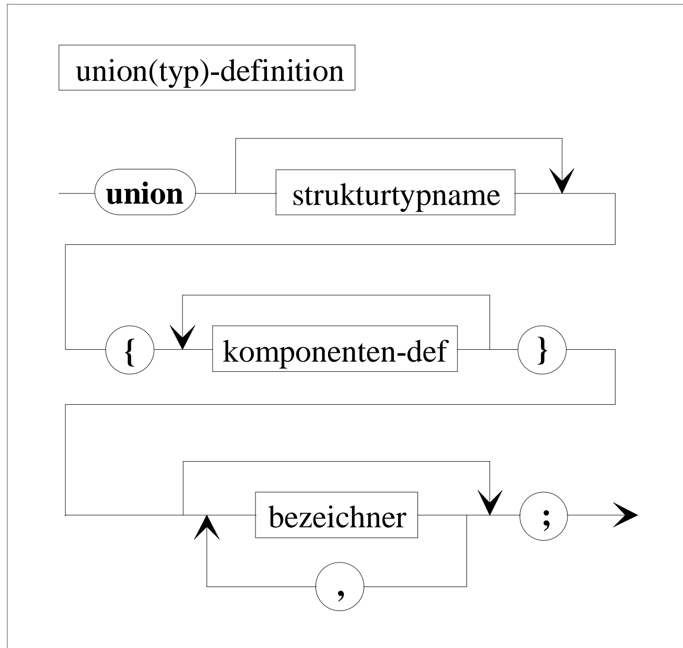
- <datentyp> ist unsigned oder int
- <größe> gibt die zu einer Komponente gehörende Anzahl von Bits an (1 .. 16)
- wird der Bitfeld-bezeichner weggelassen, so existiert das Feld zwar physisch, es kann aber nicht darauf zugegriffen werden
- der für eine Bitfeld-Struktur benötigte Speicherplatz wird in Blöck zu sizeof(int) allociert

Beispiel

```
struct patient
{
    char Name[20];
    struct
    {
        unsigned Mumps : 1;
        unsigned Masern : 1;
        unsigned Roeteln : 1;
    } Anamnese;
    int Alter;
};
patient p;
printf("Name:  ");
scanf("%s", p.Name);
printf("Anamnese\n");
{
    printf(Mumps (j/n):  ");
    char k=getchar();
    if(k=='j')
        p.Anamnese.Mumps=1;
    else
        p.Anamnese.Mumps=0;
. . .
}
```

Unions

- Unions sind Stukturen, bei denen nur jeweils einer ihrer Komponenten aktuell Werte zugewiesen werden können (Variante)



- die Größe des für eine Union allocierten Speicherplatzes entspricht der der größten Komponenten. Alle Komponenten teilen sich diesen Speicherplatz

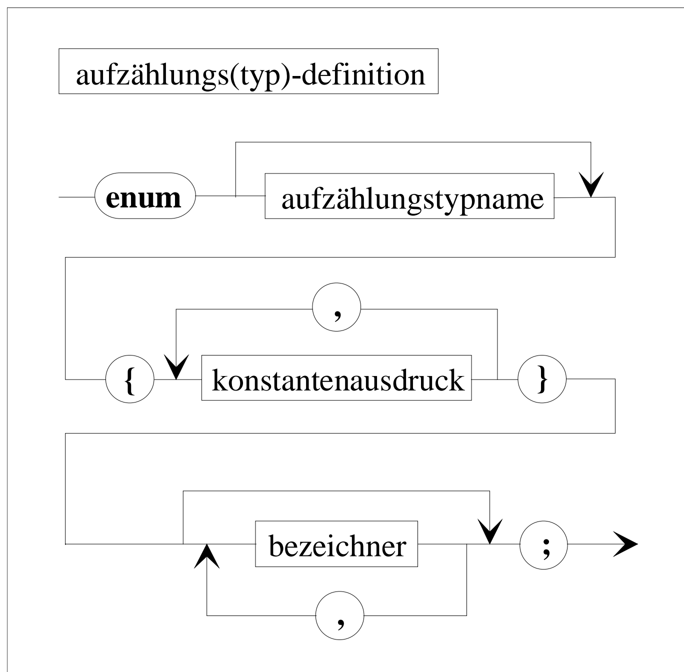
Beispiel

```
union
{
    int i;
    char s[10];
} variante;
printf("%d", sizeof(variante));
scanf("%d", &variante.i);
printf("%d", variante.i);
printf("%s", variante.s);
scanf("%s", variante.s);
printf("%s", variante.s);
printf("%d", variante.i);
```

- Unions können nur über die erste Komponeten initialisiert werden

2.6.3 Aufzählungen

- ein Aufzählungstyp definiert eine Menge von Bezeichnern (Konstanten) für konstante integer-Werte



- in C++ kann eine Variable (Bezeichner) vom Aufzählungstyp die in den Konstantenausdrücken vorgegebenen Werte annehmen
- Die Konstanten können nur integer-Werte annehmen. Sie können auch beliebigen Variablen zugewiesen werden
- Werden Konstanten nicht explizit Werte zugewiesen, erhalten sie in der Reihenfolge ihres Auftretens von liks nach rechts die Werte 0, 1, 2, ...

Beispiel

```
enum { a, b, c, d } x;
x=a;
printf("%d", x);
x=c+d;
int i=b;
printf("%d", i);
```

Beispiel

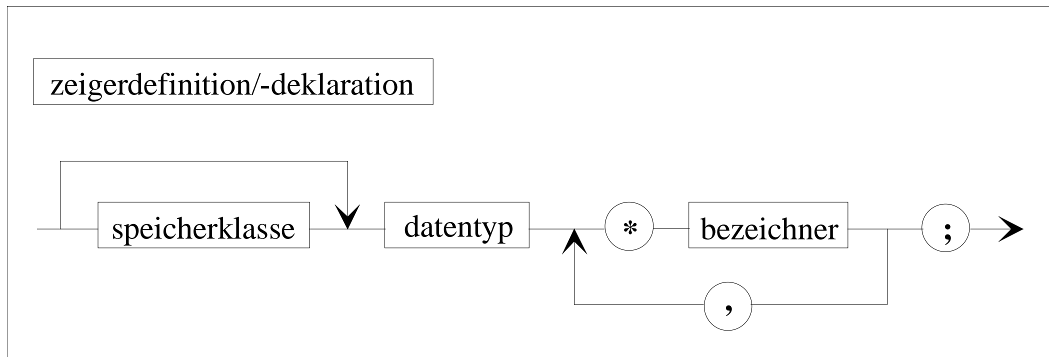
```
enum { einer=1, zweier, fünfer=5, zehner=2*fünfer, fünfziger=5*zehner } geld;
geld=einer+fünfer+zehner+zehner;
```

Beispiel

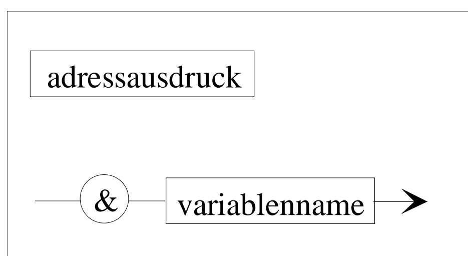
```
enum { nl='\n', tab='\t' } z;
switch(z)
{
   case  nl:    printf("new line");  break;
   case  tab:   printf("tabulator");
}
```

2.7 Zeiger

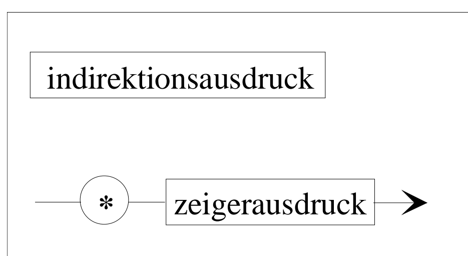
2.7.1 Definition und Operatoren



- Zeigervariablen zeigen auf Speicherplätze, deren Inhalte Werte eines bestimmten Datentyps sind (strenge Typprüfung)

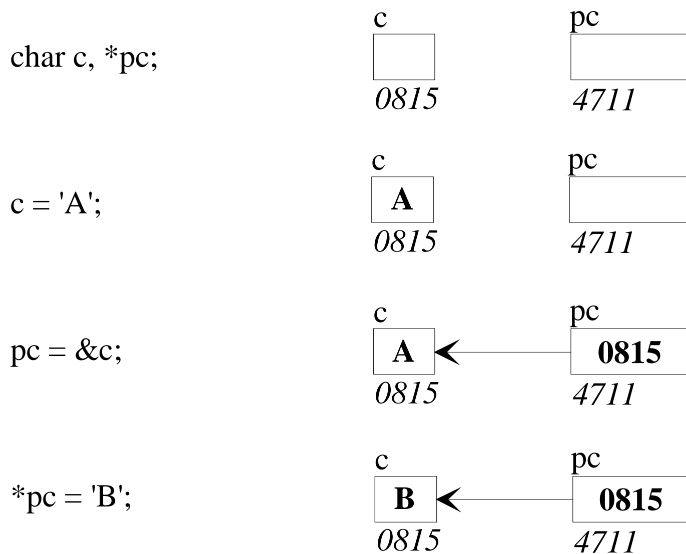


- der Adressoperator liefert die (Anfangs)-Adresse des Speicherplatzes, der der Variablen zugewiesen wurde



- der Indirektionsoperator liefert den Inhalt (Wert) der aus dem Zeigerausdruck resultierenden Adresse

Beispiel



Zeiger und Typecasting

- Umwandlung des Typs einer Zeigervariablen oder Adresse in einen anderen Zeigertyp

- <zeigertyp> ::= <datentyp> *****
  <typecasting> ::= (<zeigertyp>) <zeigervariable>
  | (<zeigertyp>) <adresse>

Zeiger und Konstanten

1. Konstanter Zeiger
   Zeiger, dessen Wert (Adresse) nach der Initialisierung nicht mehr geändert werden kann
2. Zeiger auf Konstante
   Zeiger auf Speicherplatz, dessen Inhalt nicht verändert werden kann

Beispiel

```
int i;
int * const cpi = &i;
const int j = 123;
const int * pcj;
pcj = &j;
const int * const cpcj = &j;
```

2.7.2 Zeiger, Felder und Zeigerarithmetik

Zeiger auf Felder

- der Feldname selbt ist ein (konstanter) Zeiger auf den Feldanfang:
  <feldname> ⇔ &<feldname>[0]

Zeigerarithmetik

- Zeiger können in arithmetischen Ausdrücken verwendet und eingeschränkt manipuliert werden

Operationen auf Zeiger:

1. Addition/Subtraktion eines Integer-Wertes zu/von einem Zeiger bzw. einer Adresse
   <zeiger|adresse> +|- <integer-ausdruck>
   das Ergebnis ist eine Adresse, die sich aus
   <ergebnisadresse> = <adresse> +|- (<integer-ausdruck>*sizeof(datentyp der adresse))

2. Addition/Subtraktion zweier Zeiger bzw. Adressen gleichen Typs
   <zeiger|adresse> +|- <zeiger|adresse>
   das Ergebnis ist eine Adresse, die sich aus
   <ergebnisadresse> = <adresse> +|- (<adresse>*sizeof(datentyp der adresse))

3. Vergleich zweier Zeiger bzw. Adressen

Beispiel

```
float f[ ] = {1.23, 2.34, 3.45, 4.56, 5.67};
float *pf = f;
printf(„%f“, *pf);
pf = pf + 2;
printf(„%f“, *pf);
```

Beispiel

```
int f[10], *pf = f;
for(int i=0; i<10; i++)
    *(pf+i) = i;
```

Beispiel

```
int f[ ] = {3, 7, 12, 9, 22, 71};
int *pf = f;
int s = 0;
for(int i=0; i<6; i++)
    s += *pf++; printf(„%d“, s);
```

Beispiel

```
char s1[ ]=“Hallo world“;
char s2[20];
char *ps1=s1, *ps2=s2;
while(*ps1!=’\0’)
{
    *ps2=*ps1;
    ps1++;
    ps2++;
}
*ps2=*ps1;
```

Beispiel

```
char f[100], *pf=f;
scanf(„%s“, f);
while(*pf++);
printf(„%d“, pf-f);
```

Zeigerfelder

- Definition:
  <zeigertyp> <feldname> **[** <anzahl> **] ;**

Beispiel

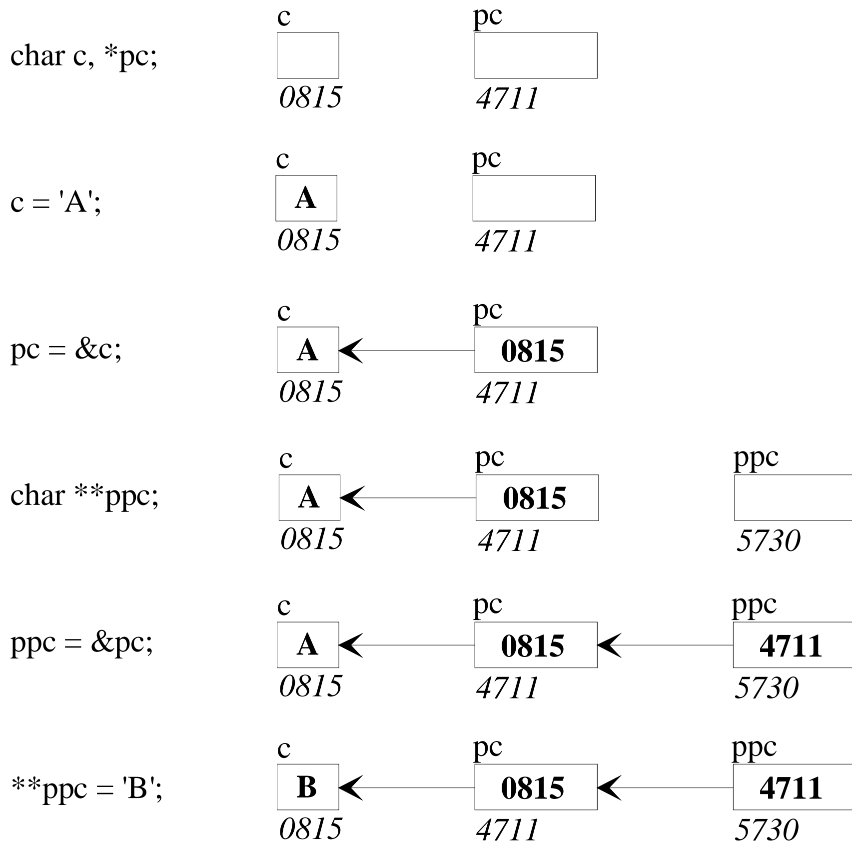
```
int nr;
char *monat[5]={“Illegal“, "Januar", "Februar", "März", "April"};
scanf("%d", &nr);
printf("Monat %s", (nr<1|| nr>4) ? monat[0] : monat[nr]);
```

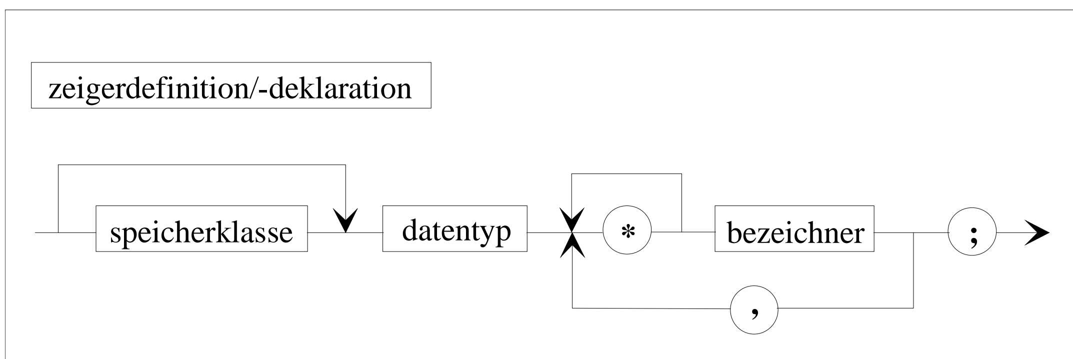
2.7.3 Zeiger auf Zeiger

- <datentyp> ::= **int**

| **char**
| . . .
| <datentyp> *   (Zeigertyp)

Beispiel





Beispiel

```
char *Monat[5]={ "Januar", "Februar", "März", "April"};
Monat[4] = NULL;
for(int i=0; Monat[i]!=NULL; i++)
    printf("\n%s", Monat[i]);
char **pMonat;
pMonat = Monat;
while(*pMonat)
    printf("\n%s", *pMonat++);
```

2.7.4 Zeiger und Strukturen

Zeiger auf Strukturen

- <datentyp> ::= **int**
  | **char**
  | . . .
  | <datentyp> ***** (Zeigertyp)
  | <strukturtyp>

- <zeiger-auf-struktur> ::= <strukturtyp> ***** <bezeichner> **;**

Zugriff auf Komponenten (Komponentenbezeichner)

- **Pfeiloperator: ->**
  (***** <zeiger>) **.** <komponente> ⇔ <zeiger> **->** <komponente

Beispiel

```
struct
{
    char Name[10];
    int Alter;
} Mitarbeiter[20], *pMit;
for(int i=0; i<20; i++)
{
    pMit = &Mitarbeiter[i];
    scanf("%s%d", pMit->Name, &pMit->Alter);
}
pMit = Mitarbeiter;
for(int i=0; i<20; i++)
{
    printf("\n%s  %d", pMit->Name, pMit->Alter);
    pMit++;
}
```

Beispiel

```
struct
{
    char Name[10];
    struct
    {
        char Ort[10];
        char Straße[20];
        int Nr;
    } Wohnung;
    int Alter;
} Person[10], *pPer = Person;
for(int i=0; i<10; i++)
{
    scanf("%s%d", pPer->Name, &pPer->Alter, pPer->Wohnung.Ort,
            pPer->Wohnung.Straße, &pPer->Wohnung.Nr);
    pPer++;
}
```

Zeiger als Strukturkomponenten

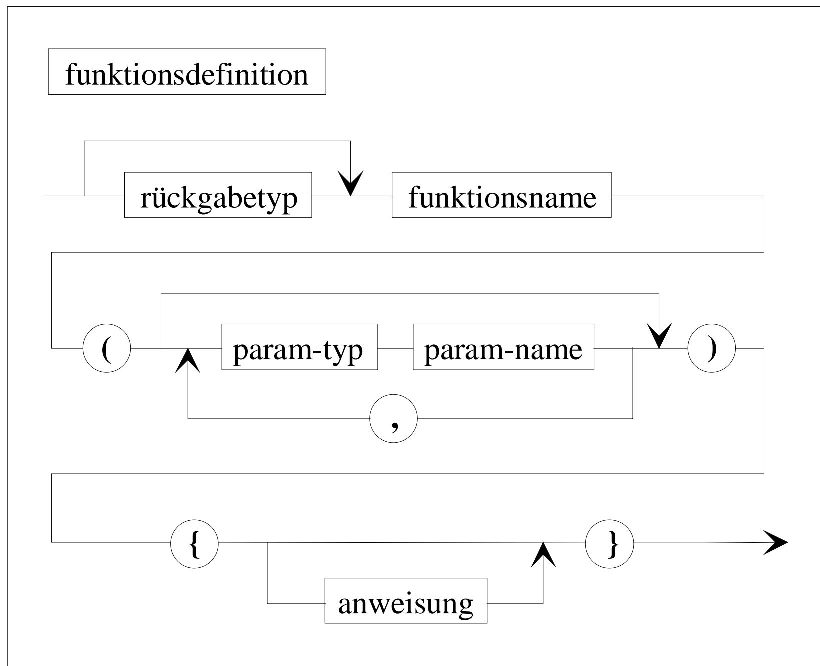
Beispiel

```
struct wohnort
{
    char Ort[10];
    char Straße[20];
    int Nr;
} Hauptwohnung, Nebenwohnung;
struct
{
    char Name[10];
    wohnort *Aktuell;
    int Alter;
} Person, *pPer = &Person;
scanf("%s%d", Person.Name, &Person.Alter);
scanf("%s%s%d", Hauptwohnung.Ort, Hauptwohnung.Straße, &Hauptwohnung.Nr);
scanf("%s%s%d", Nebenwohnung.Ort, Nebenwohnung.Straße, &Nebenwohnung.Nr);

pPer->Aktuell = &Nebenwohnung;
printf("%s wohnt zur Zeit in %s, %s %d", pPer->Name, pPer->Aktuell->Ort,
        pPer->Aktuell->Straße, pPer->Aktuell->Nr);
```

2.8 Funktionen

2.8.1 Definition und Deklaration

- Funktion: selbständiger Programmabschnitt, der eine spezifische wohldefinierte Task ausführt



- <rückgabetyp>:
  - Datentyp des Wertes, den die Funktion an das aufrufende Programm zurückgibt
  - wird kein Rückgabetyp angegeben, dann wird als Rückgabetyp **int** angenommen
  - soll die Funktion keinen Wert zurückgeben, ist der Rückgabetyp **void**

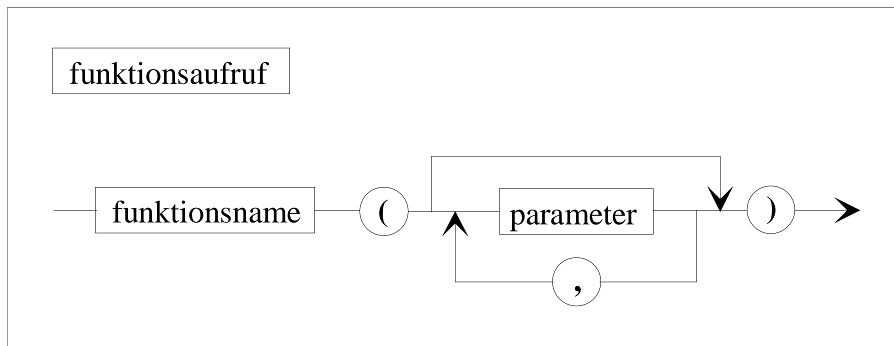
- <parameter-typ><parameter-name>:
  - eine Folge dieser Tupel bildet die Parameteriste, über die der Funktion beim Aufruf Werte übergeben werden
  - ist die Liste leer (keine Werteübergabe) kann auch **void** eingetragen werden

- <anweisungen>:
  - ist der Rückgabetyp nicht **void**, dann muß die Funktion mindestens eine Anweisung

    return <ausdruck> **;**

    enthalten.

- eine Funktionsabarbeitung wird beendet, wenn eine **return**-Anweisung oder das Ende des Funktionsrumpfes erreicht ist



- <parameter>:
  - Parameter sind Ausdrücke und können insbesondere
    Werte (Konstanten)
    Variablen
    Funktionsaufrufe, die Werte zurückgeben
    sein, die der Funktion beim Aufruf übergeben werden
  - gibt eine Funktion einen Wert zurück, dann kann sie an beliebigen Stellen im Programm, an denen Werte verlangt werden, aufgerufen werden. Im übrigen kann ein Funktionsaufruf an jeder Stelle, an der die Funktion gültig ist, als eigenständige Anweisung stehen

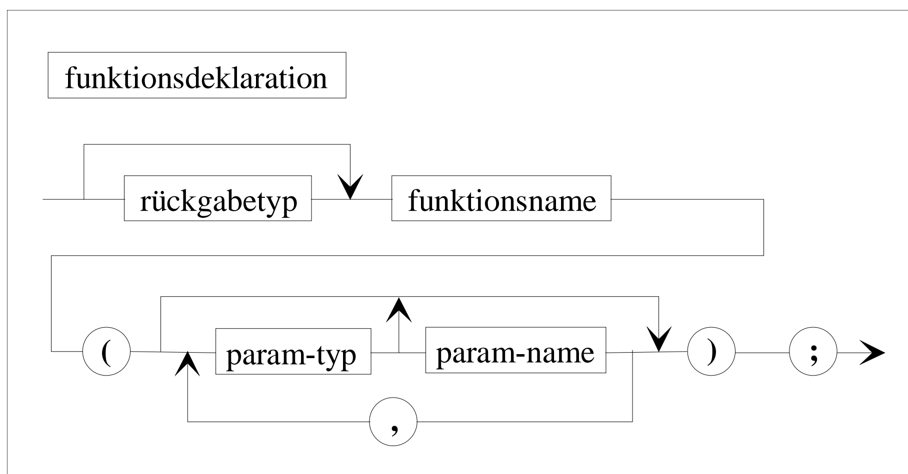
Beispiel

```
void alarm(void)
{
   for(int i=0; i<50; i++)
      putchar('\a');
}
void ausgabe_mit_alarm(char s[])
{
   printf("%s!", s);
   alarm();
   alarm();
   alarm();
}
main()
{
   int h, m;
   alarm();
   do
   {
      int fehler=0;
      printf("\nWie spät (hh.mm)?  ");
      scanf("%d.%d", &h, &m);
      if((h<0)||(h>23))
      {
         ausgabe_mit_alarm("Falsche Stunde");
         fehler=1;
```

```
        }
        if((m<0)||(m>59))
        {
            ausgabe_mit_alarm("Falsche Minute");
            fehler=1;
        }
    } while (fehler);
}
```

Beispiel

```
long x_hoch_y(int x, unsigned y)
{
    long z=1;
    if(!y)
        return z;
    for(int i=1; i<=y; i++)
        z*=x;
    return z;
}
main()
{
    long y;
    int x;
    unsigned n;
    scanf("%d%d", &x, &n);
    y = x_hoch_y(x, n) + x_hoch_y(x, n+1);
    printf("%d", y);
}
```



- Eine Funktion ist von der Stelle ihrer Deklaration bis zum Blockende/Programm-Datei-Ende bekannt (gültig/sichtbar)

- Eine Funktion kann als eigenständige Zeile an jeder Stelle im Programm (auch mehrmals) deklariert werden
- die Funktionsdeklaration muß im Rückgabetyp, Funktionsname und Anzahl/Typ/Reihenfolge der Parameter mit der zugehörigen Funktionsdefinition übereinstimmen

Beispiel (siehe oben)

```
void alarm(void);
void ausgabe_mit_alarm(char);
main()
{
    . . .
    alarm();
    . . .
            ausgabe_mit_alarm("Falsche Stunde");
    . . .
            ausgabe_mit_alarm("Falsche Minute");
    . . .
}
void alarm(void)
{
    . . .
}
void ausgabe_mit_alarm(char s[])
{
    . . .
}
```

2.8.2 Zeiger und Funktionen

Globale Variable

- Variable, die außerhalb aller Funktionen definiert wurden und deshalb überall sichtbar sind

Beispiel

```
int x, y;
void f(void)
{
    ++x;
    ++y;
}
main()
{
    int x=1, y=1;
    f();
    printf("%d %d", x, y);
}
```

Referenzieren über Zeiger

- Funktionsparameter: Zeiger auf Variable

Beispiel

```
void f(int *x, int *y)
{
    ++*x;
    ++*y;
}
main()
{
    int x=1, y=1;
    f(&x, &y);
    printf("%d %d", x, y);
}
```

Beispiel

```
int *Max(int *pos)
{
    int *max=pos;
    while(*pos)
    {
        if(*pos>*max)
            max=pos;
        pos++;
    }
    return max;
}
main()
{
    int f[20];
    for(int i=0; i<19; i++)
        f[i]=random(100)+1;
    f[19]=0;
    int *akt=f;
    while(akt)
    {
        int *max=Max(akt);
        int tmp=*akt;
        *akt=*max;
        *max=tmp;
        akt++;
    }
    for(i=0; i<19; i++)
        printf("%d\n", f[i]);
}
```

Beispiel

```
char (*ein(char (*z)[80]))[80]
{
    int i=0;
    while(((*z)[i++]=getchar())!='\n');
    (*z)[--i]='\0';
    return ++z;
}
main()
{
    clrscr();
    char f[5][80];
    char (*anfang)[80];
    char (*ende)[80];
    anfang=ende=f;
    ende=ein(ende);
    ende=ein(ende);
    ende=ein(ende);
    while(anfang<ende)
        printf("%s\n", *anfang++);
}
```

Zeiger auf Funktionen

- neuer Zeigertyp: Zeiger auf Funktion

  <datentyp> **(*** <funktionsname> **)(** <paramaterliste> **)**

Beispiel

```
int plus(int x, int y)
{
    return (x+y);
}
int mal(int x, int y)
{
        return (x*y);
}
void aufruf(int parameter1, int parameter2, int (*funktion)(int, int))
{
    printf("%d\n", (*funktion)(parameter1, parameter2));
}
main()
{
    aufruf(3, 4, plus);
    aufruf(3, 4, mal);
}
```

2.8.3 Stringfunktionen

char *strcpy(char *ziel, const char *quelle)

- kopiert einen String ab Adresse quelle bis einschließlich '\0' in einen Speicherbereich mit Adresse ziel
- gibt die Anfangsadresse ziel zurück

char *strcat(char *ziel, const char *quelle)

- hängt einen String ab Adresse quelle bis einschließlich '\0' an einen im Speicherbereich mit Adresse ziel bereits vorhandenen String an
- gibt die Anfangsadresse ziel zurück

char *strnset(char *s, char c, int n)

- füllt einen String ab Adresse s mit einer Anzahl n von Zeichen c
- gibt die Anfangsadresse s zurück

int strlen(const char *s)

- gibt die Anzahl der Zeichen eines Strings ab Adresse s (ohne abschließendes '\0') zurück

int strcmp(const char s1, const char s2)

- Vergleicht zwei Strings zeichenweise miteinander
- Rückgabewert < 0 wenn s1 kleiner s2
$= 0$ wenn s1 gleich s2
> 0 wenn s1 größer s2

char *strchr(const char s, char c)

- durchsucht eine String nach dem ersten Auftreten des Zeichens c
- liefert einen Zeiger auf das Zeichen oder NULL, wenn Zeichen nicht vorhanden

char *strstr(const char *s1, const char *s2)

- sucht des String s1 nach dem ersten Vorkommen des Teilstrings s2 ab
- liefert einen Zeiger auf den Anfang des Teilstrings s2 in s1oder NULL, wenn Zeichen nicht vorhanden

Beispiel

```
#include <stdio.h>
#include <string.h>
main()
{
    char text[200];
    char s[20];
    scanf("%s", s);
    strcpy(text, s);
    while(1)
    {
        strcat(text, " ");
        scanf("%s", s);
        if(!strcmp(s,"ENDE")) break;
        strcat(text, s);
    }
    printf("%s", text);
}
```

Beispiel

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
main()
{
    clrscr();
    char text[800];
    int i=0;
    while((text[i++]=getchar())!=EOF);
    text[--i]='\0';

    int anzahl=0;
    char *t=text;
    char string[20];
    printf("\n\nSuchstring: ");
    scanf("%s",string);
    t=strstr(t, string);
    while(t)
    {
        anzahl++;
        t=strstr(t+1, string);
    }
    printf("\nAnzahl: %d", anzahl);
}
```

int atoi(const char *s)

- konvertiert einen String in einen Integer-Wert und liefert diesen zurück
- String muß dem Format [<whitespace>][<vorzeichen>]<ziffernfolge> entsprechen
- Konvertierung endet beim ersten nicht interpretierbaren Zeichen

char *itoa(int i, char *s, int basis)

- konvertiert einen Integer-Wert i in einen String und speichert das Ergebnis in s
- basis: Basis der Konvertierung (2 .. 36), 10 dezimal
- liefert einen Zeiger auf s zurück

Beispiel

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char text[]="Ihr Gehalt beträgt 4500,- DM monatlich.";
    char *gehalt=text;
    char *ende;
    char zeichen;
    while((*gehalt<'0')||(*gehalt>'9'))
        gehalt++;
    ende=gehalt;
    while((*ende<='9')&&(*ende>='0'))
        ende++;
    zeichen=*ende;
    *ende='\0';
    itoa(atoi(gehalt)+atoi(gehalt)/100, gehalt, 10);
    *ende=zeichen;
    printf("\n\n%s", text);
}
```

2.8.4 Dateifunktionen

FILE *fopen(const char *dateiname, const char *modus)

- öffnet eine durch dateiname bezeichnete Datei und ordnet ihr einen Stream zu
- gibt den Zeiger auf den Steam zurück, bei Fehler NULL
- modus: Art der beabsichtigten Dateioperationen

"r" - nur Lesen
"w" - Erzeugen und Schreiben, falls Datei bereits existiert, wird sie überschrieben
"a" - Erzeugen und Schreiben, falls Datei bereits existiert, wird angefügt

int fclose(FILE *stream)

- schließt eine geöffnete Datei, Freigabe des Streams (der Puffer)
- liefert 0 bei fehlerfreier Ausführung, sonst EOF

int fprintf(FILE *stream, <formatzeichenkette> [**,** <argumentliste>] **)**

- gibt die Argumente in formatierter Form (wie printf) auf die durch stream bezeichnete Datei aus
- printf(format, argumente) ⇔ fprintf(stdout, format, argumente)

int fscanf(FILE *stream, <formatzeichenkette> [**,** <adressenliste>] **)**

- liest von der durch stream bezeichneten Datei entsprechend der Formatierung Daten in die angegebenen Adressen
- scanf(format, argumente) ⇔ fscanf(stdin, format, adressen)

void rewind(FILE *stream)

- Positionierung des Datenzeigers an den Dateianfang

Beispiel

```
#include <stdio.h>
#include <stdlib.h>
FILE *datei;
char wort[50];
main()
{
   if(!(datei=fopen("testdat.txt","w")))
   {
```

```
        printf("Error: Öffnen zum Schreiben");
        exit(0);
    }
    for(int i=0;i<3;i++)
    {
        scanf("%s",wort);
        fprintf(datei,"%s\n",wort);
    }
    fclose(datei);
    if(!(datei=fopen("testdat.txt","r")))
    {
        printf("Error: Öffnen zum Lesen");
        exit(0);
    }
    while(fscanf(datei,"%s",wort)!=EOF)
        printf("%s\n",wort);
    fclose(datei);
}
```

2.8.5 Die Funktion main()

- **int argc**
  Anzahl der Parameter, die dem Programm beim Aufruf im Betreibssystem übergeben wurden (incl. Programmname)

- **char *argv[ ]**
  Zeiger auf ein Zeichenkettenfeld, das jeden übergebenen Parameter als String enthält.

- **char *env[ ]**
  Zeiger auf ein Zeichenkettenfeld, das die aktuellen Umgebungs-Variablen als Strings enthält. Optional.

Beispiel

Datei: mainargv.cpp

```
#include <stdio.h>
main(int argc, char *argv[], char *env[])
{
    printf("Anzahl der Parameter: %d", argc);
    for(int i=0; i<=argc; i++)
        printf("\nargv[%d] = %s", i, argv[i]);
    while(*env)
        printf("\nenv = %s", *env++);
}
```

Beispiel

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    FILE *datei;
    char c;
    int Anzahl=0;
    datei=fopen(argv[1],"r"))
    while(fscanf(datei, "%c")!=EOF)
        Anzahl++;
    fclose(datei);
    printf("\nAnzahl: %d", Anzahl);
}
```

2.8.6 Erweiterungen für Funktionen

Inline-Funktionen

- neues Schlüsselwort vor der Deklaration/Definition einer Funktion:

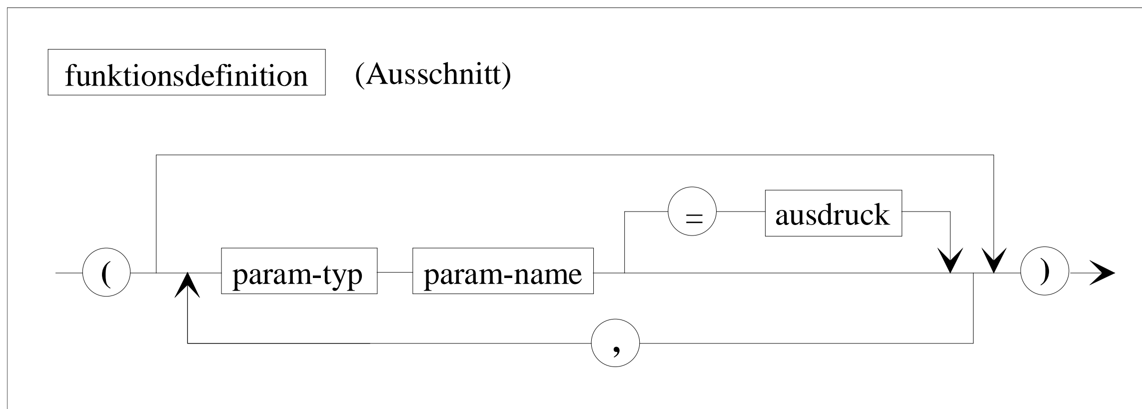
- bewirkt, daß Funktion wie Makro behandelt wird: Expansion des Quelltextes
- Volle Typprüfung wie bei Funktionen
- inline-Funktionen sollten nur 1..3 Instruktionen enthalten
- Compiler entscheidet selbständig, ob inline-Funktion als Makro expandiert wird oder nicht (z.B. nicht bei Steuerschleifen in der Funktion)

Beispiel

```
#include <stdio.h>
inline int d(int i)
{
   return 2*i;
}
main()
{
   int i=2;
   float f=2.2;
   char c='2';
   printf("%d  %f  %c", d(i), d(f), d(c));
}
```

Default-Parameter

- Möglichkeit, Argumenten von Funktionen bei der Definition bereits Werte zuzuweisen



- Hat eine Funktion Default-Argumente, könne diese beim Aufruf weggelassen werden. Die aktuellen Parameter werden dann mit den Default-Werten initialisiert
- Wird eine Funktion mit aktuellen Parametern anstelle der Default-Argumente aufgerufen, dann werden die Default-Werte durch die aktuellen Werte überschrieben

Beispiel

```
#include <conio.h>
void TOXY(int x, int y=wherey())
{
   gotoxy(x,y);
}
void TO_Ursprung(int x=1, int y=1)
{
   gotoxy(x,y);
}
main()
{
   TOXY(1);
   TOXY(12,40);
   TO_Ursprung()
}
```

- Existiert ein Default-Argument einer Funktion, dann müssen alle rechts davon auftretenden Argumente ebenfalls Default-Argumente sein

Beispiel

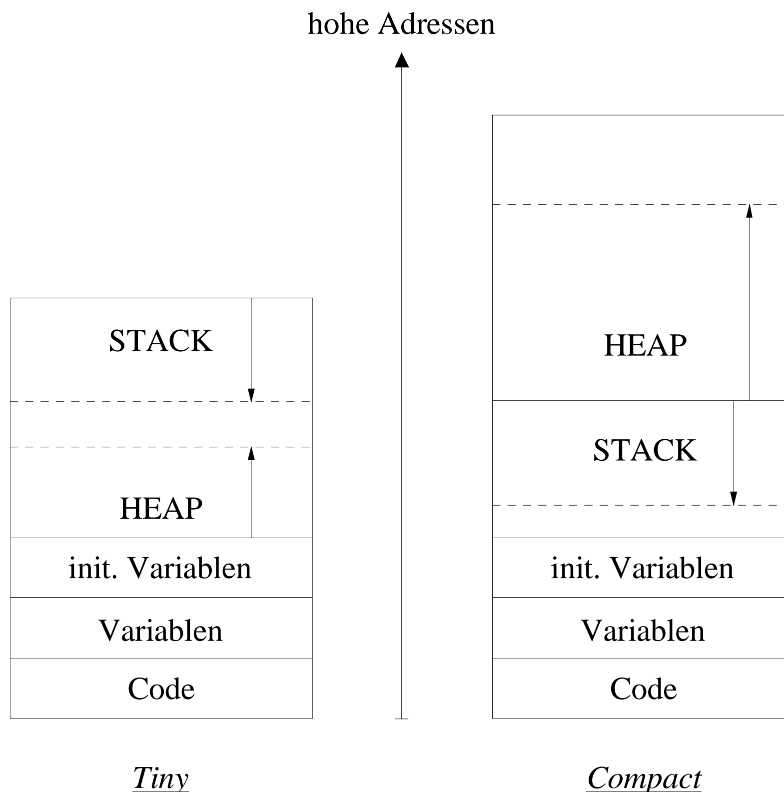
```
void TOXY(int x=wherex(), int y)
{
   gotoxy(x,y);
}
```

2.9 Dynamische Speicherverwaltung

- Variablendefinition: Reservierung von Speicherplatz durch den Compiler im *Statischen Speicherbereich*
- Freigabe des Speichers erst bei Verlassen des Blockes (auto, register) oder bei Programmende (static)
- Dynamische Speicherverwaltung: Reservierung und Freigabe von Speicherplatzt zur Laufzeit eines Programms im *Dynamischen Speicherbereich (Heap)*
- Voraussetzung dafür: Adressierung eines reservierten Speicherbereichs über Zeiger

Zeigertypen und Speichermodelle

- **near**-Zeiger
  Adressierung nur innerhalb des aktuellen Segments (Offset-Adresse), 16 Bit, Speichermodelle Tiny und Small
- **far**-Zeiger
  Adressierung über mehrere Segmente (Segment:Offset-Adresse), 32 Bit, Speichemodelle Medium, Compact, Large und Huge



2.9.1 Funktionen zur dynamischen Speicherverwaltung

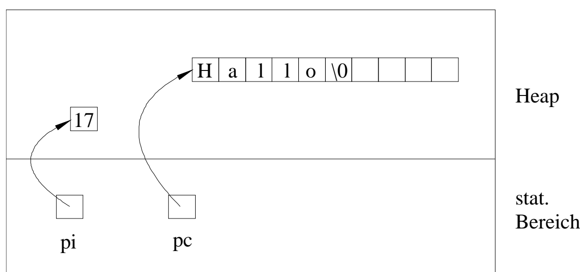
- Prototyping der Speicherfunktionen in alloc.h

a) Reservieren dynamischen Speicherbereichs

- **void *malloc(unsigned size)**
  - reserviert Speicherbereich der Größe size (in Byte) im heap
  - liefert den Zeiger auf den Anfang des reservierten Speicherbereichs zurück (generischer Zeiger)
  - ist size=0 oder Fehler (i.d. Regel kein ausreichender Speicherplatz) wird NULL zurückgegeben
  - Zuweisung des generischen Zeigers an eine Zeigervariable (eines Typs) durch typecasting
  - malloc deklariert in alloc.h

Beispiel

```
#include <stdio.h>
#include <alloc.h>
main()
{
    int *pi;
    pi=(int *)malloc(sizeof(int));
    *pi=17;
    char *pc;
    pc=(char *)malloc(10);
    scanf("%s", pc);
}
```



Beispiel

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main()
{
    char *text[5];
    char zeile[81];
    int laenge;
    printf("\n");
    for(int i=0; i<5; i++)
    {
        printf("EINGABE: ");
        char *pz=zeile;
        laenge=1;
        while((*pz++=getchar())!='\n') laenge++;
        *--pz='\0';
        text[i]=(char *) malloc(laenge);
        strcpy(text[i],zeile);
    }
    for(i=0; i<5; i++)
        printf("\n%s",text[i]);
    return 0;
}
```

Beispiel

```
#include <stdio.h>
#include <alloc.h>
main()
{
    struct person
    {
        char Name[10];
        int Alter;
    } *f[5];
    for(int i=0; i<5; i++)
    {
        f[i]=(person *) malloc(sizeof(person));
        printf("Name: "); scanf("%s", f[i]->Name);
        printf("Alter: "); scanf("%d", &f[i]->Alter);
    }
}
```

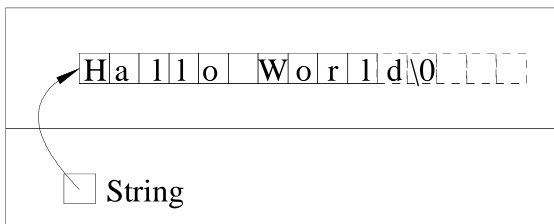
b) Erweiterung bereits vorhandenen dynamischen Speicherbereichs

- Problem: malloc reserviert nur neuen Speicherbereich, physische Lokalisation ist nicht beeinflußbar

- **void *realloc(void *block, unsigned size)**
  - verändert (vergrößert) einen bereits allocierten Speicherbereich block auf die Größe size (in Byte)
  - sonst wie malloc

Beispiel

```
main()
{
    char *String=(char *) malloc(10*sizeof(char));
    strcpy(String, "Hallo");
    printf("%s", String);
    String=(char *) realloc(String, 15*sizeof(char));
    strcat(String, "World!");
    printf("%s", String);
}
```



- bei der Rallocation wird
  1. neuer Speicherbereich allociert
  2. Inhalt umgespeichert
  3. alter Speicherbereich freigegeben

- ist block=NULL, dann wirkt realloc wie malloc:
  realloc(NULL, size) ≡ malloc(size)

Beispiel

```
#include <stdio.h>
#include <alloc.h>
main()
{
    int *f=NULL;
    for(int i=0;;i++)
    {
        f=(int *) realloc(f,(i+1)*sizeof(int));
        scanf("%d",&f[i]);
        if(f[i]<0) break;
    }
    while(*f)
        printf("\n%d",*f++);
```

```
    return 0;
}
```

Beispiel

```
main()
{
    char *a, *b;
    a=(char *)malloc(6);
    strcpy(a,"HALLO");
    printf("%s",a);
    b=a;
    printf("\n%s",b);
    a=(char *)realloc(a,7);
    printf("\n%s",a);
    printf("\n%s",b);
}
```

Beispiel

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
main()
{
    char **text=NULL;
    printf("\n");
    char zeile[81];
    do
    {
        static int i=0;
        printf("EIN: ");
        char *pz=zeile;
        int laenge=1;
        while((*pz++=getchar())!='\n') laenge++;
        *--pz='\0';
        text=(char **) realloc(text,(i+1)*sizeof(char *));
        text[i]=(char *) malloc(laenge);
        strcpy(text[i++], zeile);
    } while(*zeile);
    while(**text)
        printf("\n%s",*text++);
    return 0;
}
```

c) Freigabe allocierten dynamischen Speicherbereichs

- Speicherbereich automatisch nach Beendigung eines Programms freigegeben
- **void free(void *block)**
  - gibt an der Adresse block allocierten Speicherbereich wieder frei

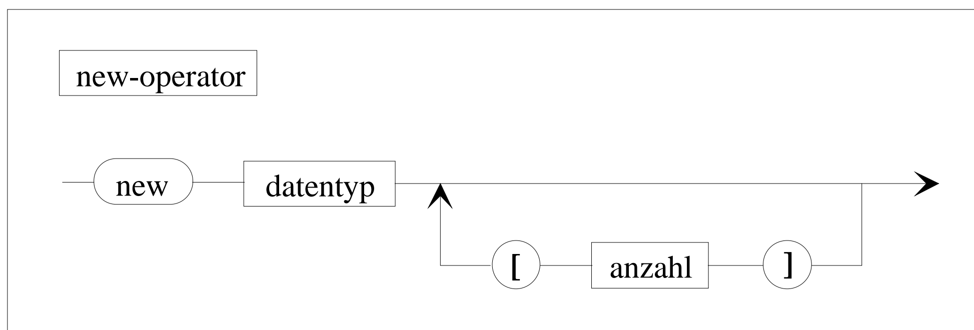
Beispiel

```
char *pc;
pc=(char *) malloc(100);
scanf("%s",pc);
printf("%s",pc);
free(pc);
```

2.9.2 Operatoren zur dynamischen Speicherverwaltung

- sind voll mit den Speicherfunktionen kompatibel (wechselseitige Verwendung möglich)

a) Reservieren dynamischen Speicherbereichs



- reserviert Speicherbereich der für den datentyp erforderlichen Größe im heap
- liefert den Zeiger auf den Anfang des reservierten Speicherbereichs zurück
- bei Fehler (i.d. Regel kein ausreichender Speicherplatz) wird NULL zurückgegeben
- Vorteil gegenüber malloc: kein Typecasting und keine Berechnung des erforderlichen Speicherplatzes
- Initialisierung durch '(<ausdruck>)' einfacher Datentypen möglich

Beispiel

```
char *pc1, *pc2, **pc3;
pc1=new char;
scanf("%c", pc1);
pc2=new char ('X');
pc3=new (char *) ("Hallo world!");
printf("%c %c %s",*pc1, *pc2, *pc3);
```

Beispiel

```
main()
{
    struct str
    {
        int i;
        char string[20];
```

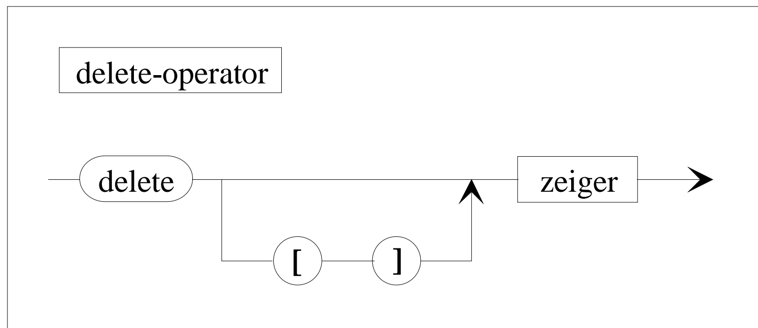
```
    };
    str *s;
    s=new str;
    s->i=2;
    strcpy(s->string, "Hallo");
    printf("\n%d %s",s->i, s->string);

}
```

- Allocation von Arrays durch Klammerung der Feldgröße

Beispiel

```
main()
{
    int (*f)[2];
    f=new int[3][2];
    int a=0;
    for(int i=0; i<3; i++)
        for(int j=0; j<2; j++)
            f[i][j]=a++;
    for(i=0; i<3; i++)
        for(int j=0; j<2; j++)
            printf("\n%d",f[i][j]);
}
```

b) Freigabe allocierten dynamischen Speicherbereichs



- gibt an der Adresse zeiger allocierten Speicherbereich wieder frei
- bei Freigabe von arrays sollten Klammern verwendet werden

Beispiel

```
int *pi;
char *pc;
pi=new int;
pc=new char[100];
. . .
delete pi;
delete[ ] pc;
```

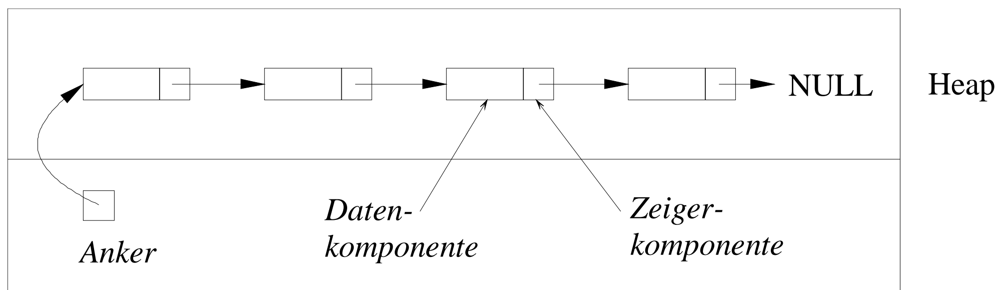
2.10 Rekursion

- Rekursion: Zurückführen einer Definition auf sich selbst.
- Ein Algorithmus heißt rekursiv, wenn er sich selbst als Einzelschritt enthält
- Grundlage der Strukturbildung

2.10.1 Rekursive Datenstrukturen

- Grundlage: Rekursive Datenstrukturen: Strukturen (struct), deren Definition auf sich selbst verweist
- weiterhin Bäume, Netze: Daten die in unterschiedlichster Form miteinander in Beziehung stehen

- Verkettete Liste: geordnete Folge von gleichartigen Datenelemente, die aufeinander zeigen



Erzeugen einer Liste

- Definition: Element einer einfach vorwärts verkettete Liste

```
struct element
{
   <datentyp> datum;
   element *next;
}
```

- Aufbau der Liste

```
(1)  element *anfang, *zeiger;
(2)  zeiger=(element *) malloc(sizeof(element));
(3)  anfang=zeiger;
(4)  zeiger->datum= ...;
(5)  zeiger->next=(element *) malloc(sizeof(element));
(6)  zeiger=zeiger->next;
(7)  zeiger->datum=...;
. . .
(n)zeiger->next=NULL;
```

Beispiel

```
main()
{
   struct element
   {
      int wert;
      element *next;
   };
   element *anfang, *zeiger;
   zeiger=(element *) malloc(sizeof(element));
   anfang=zeiger;
   for(int i=0;i<4;i++)
   {
      printf("Wert: ");
      scanf("%d",&zeiger->wert);
      zeiger->next=(element *) malloc(sizeof(element));
      zeiger=zeiger->next;
   }
   printf("Wert: ");
   scanf("%d",&zeiger->wert);
   zeiger->next=NULL;
}
```

Beispiel

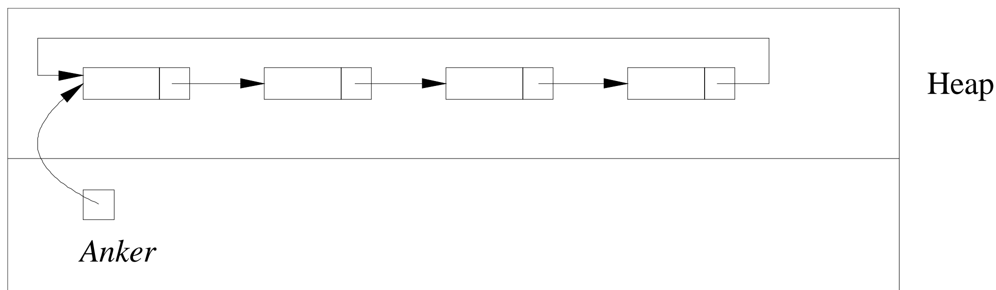
```
main()
{
   struct kind
   {
      struct
      {
         char name[10];
         int alter;
      } daten;
      kind *next;
   };
   kind *anfang, *zeiger;
   zeiger=(kind *) malloc(sizeof(kind));
   anfang=zeiger;
   for(int i=0;i<4;i++)
   {
      scanf("%s",zeiger->daten.name);
      scanf("%d",&zeiger->daten.alter);
      zeiger->next=(kind *) malloc(sizeof(kind));
      zeiger=zeiger->next;
   }
   scanf("%s",zeiger->daten.name);
   scanf("%d",&zeiger->daten.alter);
   zeiger->next=NULL;
}
```

Beispiel (zu oben)

```
int alter;
printf("\nALTER: ");
scanf("%d",&alter);
zeiger=anfang;
while(zeiger)
{
    if(zeiger->daten.alter==alter)
        printf("\nNAME: %s",zeiger->daten.name);
    zeiger=zeiger->next;
}
```

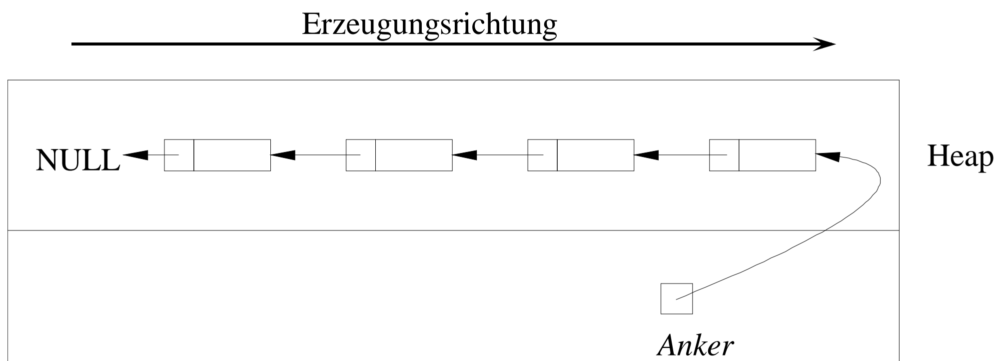
Weitere Listenformen

- Ringverkettete Liste

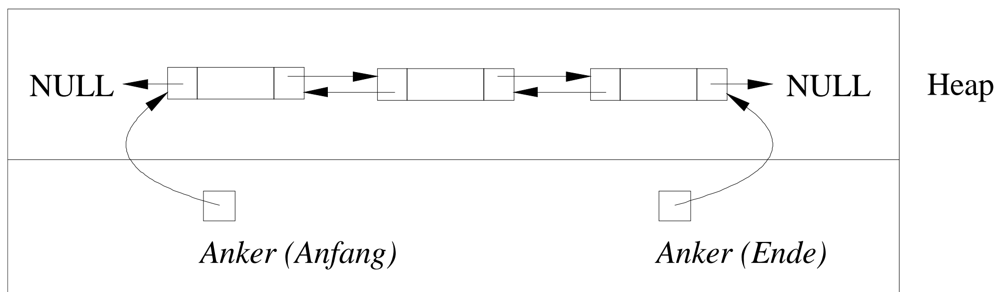


letztes Element: zeiger->next=anfang;

- Rückwärtsverkette Liste



- Doppeltverkettete Liste




```
struct element
{
    <datentyp> datum;
    element *next, *prae;
}
```

Grundoperationen auf verkettete Listen

- Anfügen vor den Anfang

```
zeiger=anfang;
anfang=(element *) malloc(sizeof(element));
anfang->datum=...;
anfang->next=zeiger;
```

- Anfügen an das Ende

```
zeiger=anfang;
while(zeiger->next)
    zeiger=zeiger->next;
zeiger->next=(element *) malloc(sizeof(element));
zeiger->next->datum=...;
zeiger->next->next=NULL;
```

- Einfügen in eine Liste

```
//zeiger zeigt auf das Element, hinter das eingefügt werden soll
element *tmp=(element *) malloc(sizeof(element));
tmp->datum=...;
tmp->next=zeiger->next;
zeiger->next=tmp;
```

- Löschen des ersten Elements

```
zeiger=anfang;
anfang=anfang->next;
free(zeiger);
```

- Löschen des letzten Elements

```
zeiger=anfang;
while(zeiger->next->next)
    zeiger=zeiger->next;
free(zeiger->next);
zeiger->next=NULL;
```

- Löschen innerhalb der Liste

```
//zeiger zeigt auf den Vorgänger des zu löschenden Elements
element *tmp=zeiger->next;
zeiger->next=zeiger->next->next;
free(tmp);
```

Allgemeine Fälle für Grundoperationen

a) Einfügen eines neuen Elements vor ein Element mit einem bestimmten Wert

```
int Wert;
element *tmp;
printf("\nWERT: "); scanf("%d",&Wert);
zeiger=anfang;
while(zeiger)
{
    if(zeiger->wert==Wert)
    {
        anfang=(element *) malloc(sizeof(element));
        printf("\NEUWERT: ");
        scanf("%d",&anfang->Wert);
        anfang->next=zeiger;
        break;
    }
    if(zeiger->next->wert==Wert)
    {
        tmp=(element *) malloc(sizeof(element));
        printf("\NEUWERT: ");
        scanf("%d",&tmp->Wert);
        tmp->next=zeiger->next;
        zeiger->next=tmp;
        break;
```

```
        }
        zeiger=zeiger->next;
    }
```

b) Löschen eines Elements mit einem bestimmten Wert

```
    int Wert;
    element *tmp;
    printf("\nWERT: "); scanf("%d",&Wert);
    zeiger=anfang;
    while(zeiger)
    {
        if(zeiger->wert==Wert)
        {
            anfang=zeiger->next;
            free(zeiger);
            break;
        }
        if(zeiger->next->wert==Wert)
        {
            tmp=zeiger->next;
            zeiger->next=zeiger->next->next;
            free(tmp);
            break;
        }
        zeiger=zeiger->next;
    }
```

Beispiel (zu Beispiel Kind oben)

```
int Alter;
kind *tmp;
printf("\nWERT: "); scanf("%d",&Alter);
zeiger=anfang;
while(zeiger)
{
    if(zeiger->daten.alter<Alter)
    {
        anfang=zeiger->next;
        free(zeiger);
        zeiger=anfang;
    }
    else break;
}
while(zeiger && zeiger->next)
{
    if(zeiger->next->daten.alter<Alter)
    {
```

```
        tmp=zeiger->next;
        zeiger->next=zeiger->next->next;
        free(tmp);
    }
    else zeiger=zeiger->next;
}
```

2.10.2 Rekursive Funktionen

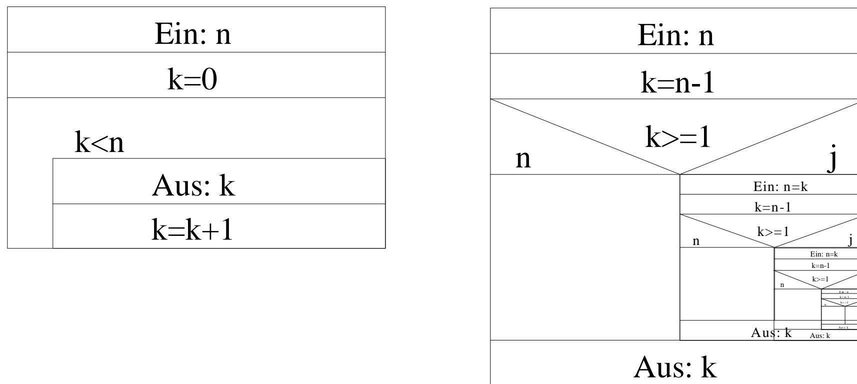
- Eine Funktion heißt rekursiv, wenn
  a) sie sich selbst aufruft (direkte Rekursion) oder
  b) sie eine andere Funktion aufruft, durch die sie aufgerufen wird (indirekte Rekursion)

- Eine rekursive Funktion terminiert, wenn
  a) eine Bedingung in der Funktion existiert, bei deren Erfüllung diese sich nicht selbst aufruft (Abbruchbedingung) und
  b) jeder erneute Aufruf der Funktion mit einer "Annäherung" an die Erfüllung dieser Bedingung verbunden ist

Beispiel

```
void iter(int);
void reku(int);
main()
{
    int z;
    scanf("%d",&z);
    iter(z);
    reku(z);
}
void iter(int n)
{
    int k;
    k=0;
    while(k<n)
        printf("%d",k++);
}
void reku(int n)
{
    int k;
    k=n-1;
    if(k>=1) reku(k);
    printf("%d",k);
}
```

- wärend sich die Iteration als PAP/Struktogramm darstellen läßt, ist das bei Rekursion nicht möglich:



- wärend Iteration grundsätzlich beliebig oft möglich (unendliche Schleifen), ist Rekursion wegen des begrenzten Stack ebenfalls begrenzt

Beispiel

```
void Stack(int i)
{
    printf("%d\n",i);
    Stack(++1);
}
main()
{
    Stack(1);
}
```

Beispiel

```
#include <stdio.h>
int fak_iter(int z)
{
    int i=1;
    while(z>0)
    {
        i*=z;
        z--;
    }
    return i;
}
int fak_reku(int z)
{
    if(z==0)
        return 1;
    else
        return fak_reku(z-1)*z;
}
```

```
void main()
{
    int z;
    scanf("%d",&z);
    printf("\n%d",fak_iter(z));
    printf("\n%d",fak_reku(z));
}
```

- Anwendungsgebiete der Rekursion: z.B Traversieren von Listen, Bäumen, Syntaxprüfung aber auch Programmierung bereits rekursiv definierter math. Funktionen

Beispiel

Fibonacci-Zahlen:

```
#include <stdio.h>
int fibo_iter(int n)
{
    int a,b,c;
    if((n==1)||(n==2)) return 1;
    a=1;
    b=1;
    for(int i=3; i<=n; i++)
    {
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
int fibo_reku(int n)
{
    if((n==1)||(n==2)) return 1;
    return fibo_reku(n-1)+fibo_reku(n-2);
}
main()
{
    int n;
    scanf("%d",&n);
    printf("\nIteration: %d",fibo_iter(n));
    printf("\nRekursion: %d",fibo_reku(n));
}
```

Beispiel

```
struct element
{
    int wert;
    element *next;
};
void loesche(element **pzeiger, int wert)
{
```

```
    if(*pzeiger)
        if((*pzeiger)->wert!=wert)
            loesche(&(*pzeiger)->next,wert);
        else
        {
            element *tmp;
            tmp=*pzeiger;
            *pzeiger=(*pzeiger)->next;
            free(tmp);
        }
}
void einfuege(element **pzeiger, int ein, int nach)
{
    if(*pzeiger)
        if((*pzeiger)->wert!=nach)
            einfuege(&(*pzeiger)->next, ein, nach);
        else
        {
            element *tmp;
            tmp=(element *) malloc(sizeof(element));
            tmp->wert=ein;
            tmp->next=(*pzeiger)->next;
            (*pzeiger)->next=tmp;
        }
}
main()
{
    . . .

    int wert;
    printf("\nWERT: "); scanf("%d",&wert);
    loesche(&anfang,wert);

    int ein, nach;
    printf("\nEIN: "); scanf("%d",&ein);
    printf("\nnNACH: "); scanf("%d",&nach);
    einfuege(&anfang, ein, nach);

    . . .
}
```

2.11 Präcompiler

Einfügen von Dateien

\# **include** <dateiname>

- fügt den Inhalt der angegebenen Datei an der Stelle der include-Anweisung in den Quelltext ein
- Datei wird in den mit Options/Project/Include Directories voreingestellten Verzeichnissen gesucht

\# **include** "dateiname"
\# **include** "pfad\dateiname"

- wie oben, Suche jedoch zuerst im aktuellen bzw. dem angegebenen Verzeichnis

Definition von Makros

\# **define** makro-name text

- jedes Auftreten des Makro-Namens im Quelltext wird durch den Text ersetzt

Beispiel

```
#define  YES  1
#define  NO   0
main()
{
. . .
   if(antwort==YES)
. . .
}
```

Beispiel

```
#define  LINKS_OBEN    1,1
#define  NACH_RECHTS_UNTEN   gotoxy(80,25);
main()
{
. . .
   gotoxy(LINKS_OBEN);
. . .
   NACH_RECHTS_UNTEN
. . .
}
```

Beispiel

```
#define  MORGENGRUSS      Morgen!
#define  ABENDGRUSS       "Abend!"
main()
{
   printf("MORGENGRUSS");
   printf(ABENDGRUSS);
}
```

 # define makro-name**(** argumentliste **)** sequenz

- argumentliste: durch Komma getrennte Folge von Argumenten wie bei Funktionen
- Aufruf des Makros im Quelltext:

  makro-name **(** aktuelle_argumentliste **)**

- Makroaufruf bewirkt zweimalige Ersetzung:
  1. makro-name **(** aktuelle_argumentliste **)** wird durch sequenz ersetzt
  2. alle in der sequenz enthaltenen Argumente der argumentliste werden durch die entsprechenden Argumente der aktuellen_argumentliste ersetzt

Beispiel

```
#define  SQARE(X)    X*X
main()
{
   int x, y;
   scanf("%d%d ", &x, &y);
   printf("%d", SQARE(x));
   printf("%d", SQARE(x+y));
}
```

Beispiel

```
#define  MAX(A,B)    (A>B ? A : B)
main()
{
   int x, y;
   scanf("%d%d", &x, &y);
   printf("%d", MAX(x,y));
   printf("%d", MAX(x=8,y=7));
   printf("%d", MAX(x++,y++));
}
```

 # undef makro-name

- schaltet die Definition eines Makros aus

Bedingte Übersetzung

 # if konstantenausdruck
anweisungen
 # endif

- ist konstantenausdruck wahr, dann werden anweisugen ausgeführt

 # if konstantenausdruck
anweisungen1
 # else
anweisungen2
 # endif

- ist konstantenausdruck wahr, dann werden anweisugen1 ausgeführt, sonst anweisungen2

 # if konstantenausdruck1
anweisungen1
 # elif konstantenausdruck2
anweisungen2
 # elif konstantenausdruck3
anweisungen3
. . .
 # endif

- die zu dem ersten wahren konstantenausdruck gehörenden anweisungen werden ausgeführt

 # if defined bezeichner ⇔ **# ifdef** bezeichner
 # if !defined bezeichner ⇔ **# ifndef** bezeichner

- ist wahr, wenn bezeichner (aktuell) definiert bzw. nicht definiert ist

Beispiel

```
#define  GROSS
#if defined  GROSS
#define   LEFT      100
#define   TOP       100
#define   RIGHT     500
#define   BUTTOM    400
#else
#define   LEFT      200
#define   TOP       400
#define   RIGHT     300
#define   BUTTOM    300
#endif
main()
{
rectangle(LEFT, TOP, RIGHT, BUTOM);
}
```

Beispiel

x.h

```
#include „y.h“
text_x
```

y.h

```
#include „x.h“
text_y
```

test.cpp

```
#include „x.h“
#include „y.h“
main()
{ }
```

x.h

```
#define  _x_h
#ifndef  _y_h
#include „y.h“
#endif
text_x
```

y.h

```
#define  _y_h
#ifndef  _x_h
#include „x.h“
#endif
text_y
```

test.cpp

```
#include „x.h“
#include „y.h“
main()
{ }
```