

# Systemprogrammierung

## Teil 3: ANSI-C Anweisungen

### Ausdrücke / Operatoren / Ablaufsteuerung

## ANSI-C Anweisungen: Übersicht

Ein Programm besteht aus Anweisungen (*Statements*):

- Variablen-Definitionen

**Typ Name = Ausdruck;**    */\* nur global oder am Anfang eines Anweisungsblocks \*/*

- Ausdrücke mit darauf folgendem Semikolon

**Ausdruck;**    */\* besteht aus Literalen, Namen und Operatoren \*/*

**;**    */\* Spezialfall leere Anweisung \*/*

- Anweisungsblöcke in geschweiften Klammern

**{Anweisung Anweisung ...}**

**{ }**    */\* Spezialfall leere Anweisung \*/*

- Anweisungen zur Ablaufsteuerung (Kontrollstrukturen)

Verzweigungen: **if-else**    **switch-case-default**

Schleifen: **while**    **do-while**    **for**

Sprünge: **break**    **continue**    **return**    **goto**

## ANSI-C Operatoren: Übersicht (1)

- Operatoren mit einem Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang <sup>1</sup>
++, --	Postfix-Inkrement/Dekrement	unär	— <sup>2</sup>	1
. <i>Komponente</i>	Auswahl	unär	— <sup>2</sup>	1
-> <i>Komponente</i>	Inhalt mit Auswahl	unär	— <sup>2</sup>	1
[ <i>Index</i> ]	Indizierung	unär <sup>3</sup>	— <sup>2</sup>	1
( <i>Parameterliste</i> )	Methodenaufruf	unär <sup>3</sup>	— <sup>2</sup>	1
++, --	Präfix-Inkrement/Dekrement	unär	— <sup>2</sup>	2
+, -	Vorzeichen Plus/Minus	unär	— <sup>2</sup>	2
!	Logische Negation	unär	— <sup>2</sup>	2
~	Bitweise Invertierung	unär	— <sup>2</sup>	2
*	Inhalt	unär	— <sup>2</sup>	2
&	Adresse	unär	— <sup>2</sup>	2
( <i>Typ</i> )	Typanpassung	unär	— <sup>2</sup>	2
sizeof	Speicherplatzbedarf	unär	— <sup>2</sup>	2

<sup>1</sup> Sortierung vom höchsten Vorrang 1 bis niedrigstem Vorrang 15.

<sup>2</sup> Einstellige Operatoren haben keine Assoziativität. Sie werden von innen nach außen berechnet.

<sup>3</sup> In ( ) oder, [ ] geklammerte Parameter der Operatoren bleiben bei der Stelligkeit unberücksichtigt.

## ANSI-C Operatoren: Übersicht (2)

- Operatoren mit zwei Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang
*	Multiplikation	binär	links	3
/	Division	binär	links	3
%	Modulo	binär	links	3
+	Addition / Konkatination	Binär	links	4
-	Subtraktion	binär	links	4
<<	Links-Shift	binär	links	5
>>	Rechts-Shift	binär	links	5
<	kleiner	binär	links	6
<=	kleiner-gleich	binär	links	6
>	größer	binär	links	6
>=	größer-gleich	binär	links	6
==	Gleichheit	binär	links	7
!=	Ungleichheit	Binär	links	7
&	bitweises Und	binär	links	8
^	bitweises XOR	binär	links	9
	bitweises Oder	binär	links	10

## ANSI-C Operatoren: Übersicht (3)

- weitere Operatoren mit zwei Operanden, bzw. in einem Fall mit drei Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang
&&	Logisches Und	binär	links	11
	Logisches Oder	binär	links	12
? :	Bedingung	ternär	rechts	13
=	Zuweisung	binär	rechts	14
+=	Additions-Zuweisung	binär	rechts	14
-=	Subtraktions-Zuweisung	binär	rechts	14
*=	Multiplikations-Zuweisung	binär	rechts	14
/=	Divisions-Zuweisung	binär	rechts	14
%=	Modulo-Zuweisung	binär	rechts	14
&=	Bitweise-XOR-Zuweisung	binär	rechts	14
^=	Bitweise-Oder-Zuweisung	binär	rechts	14
=	Bitweise-Und-Zuweisung	binär	rechts	14
<<=	Links-Shift-Zuweisung	binär	rechts	14
>>=	Rechts-Shift-Zuweisung	binär	rechts	14
,	Sequenz	binär	links	15

## ANSI-C Operatoren: Komma

Der **Komma-Operator** bildet eine Ausdrucks-Sequenz

- Syntax: `Ausdruck1, Ausdruck2`
- der **Datentyp** des Ausdrucks insgesamt  
ist der Datentyp des Ausdrucks hinter dem Komma
- der **Wert** des Ausdrucks insgesamt  
ist der Wert des Ausdrucks hinter dem Komma
- Achtung: nicht jedes Komma in einem C-Programm ist ein Komma-Operator !

`int i, tmp;`  
          ↑ *kein Komma-Operator*

`(tmp = i, ++i, tmp); /* entspricht i++; */`  
          ↑   ↑  
          *Komma-Operator*

# ANSI-C Operatoren: Logik und Vergleiche

---

**Logische Operatoren** `!` `&&` `||` verknüpfen Werte mit Zahl- und Zeigertypen.

Ein **Zahl- oder Zeigerwert 0** wird dabei als *false* interpretiert.

Jeder **Zahl- oder Zeigerwert ungleich 0** wird dabei als *true* interpretiert.

- der **Datentyp** eines logischen Ausdrucks ist `int`
- der **Wert** eines logischen Ausdrucks ist `0` oder `1`

*Achtung:* Durch den fehlenden Typ *boolean* können die logischen Operatoren `&&` und `||` leicht mit den arithmetischen Bitoperatoren `&` und `|` verwechselt werden!

Die **Vergleichs-Operatoren** `<` `<=` `>` `>=` `==` `!=` prüfen eine Relation zwischen zwei Werten.

- der **Datentyp** eines Vergleichs-Ausdrucks ist `int`
- der **Wert** eines Vergleichs-Ausdrucks ist `1`, wenn die Relation zutrifft, sonst `0`

*Achtung:* Durch den fehlenden Typ *boolean* kann in Bedingungen leicht der Gleichheitstest `==` mit der Zuweisung `=` verwechselt werden!

## ANSI-C Operatoren: Vergleich mit Java

---

Bei den Operatoren gibt es wenig Unterschiede zwischen ANSI-C und Java:

- der Sequenz-Operator `,` fehlt in Java
- der Rechts-Shift ohne Vorzeichen `>>>` fehlt in ANSI-C
- die Operatoren für Zeiger und Zeigerarithmetik `&`, `*`, `->`, `sizeof` sind in Java überflüssig
- der Typabfrage `instanceof` ist in ANSI-C überflüssig
- in ANSI-C gibt es logisches Und/Oder nur als `&&` bzw. `||` mit fauler Auswertung

*In Java haben die Operatoren `&` und `|` je nach Operandentyp eine andere Bedeutung:  
im Fall von `boolean` sind es logische Operatoren mit voller Auswertung beider Operanden  
im Fall von ganzen Zahlen sind es wie bei ANSI-C arithmetische Bitoperatoren*

## ANSI-C Ausdrücke: Auswertungs-Reihenfolge (1)

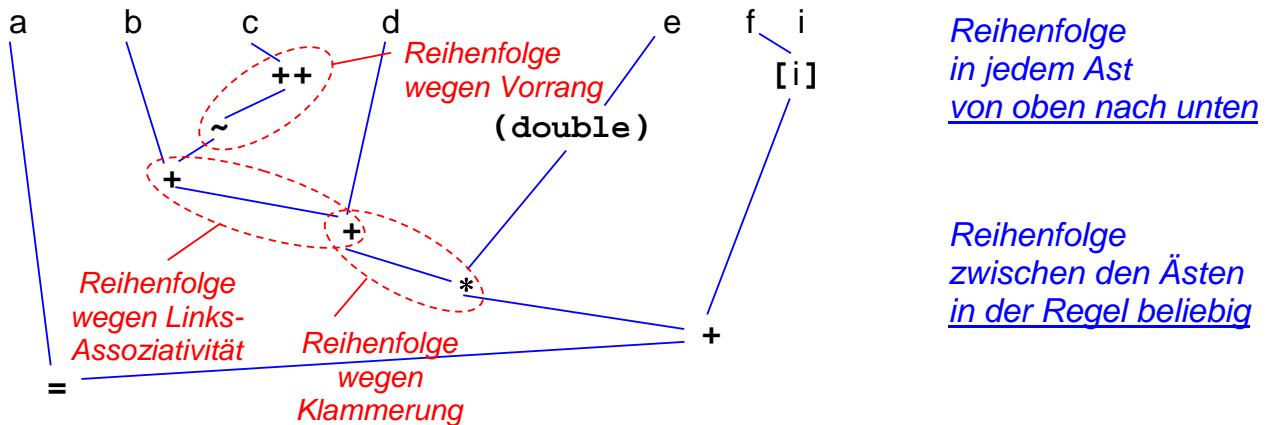
- Die **Auswertungs-Reihenfolge der Operatoren** eines Ausdrucks wird bestimmt von Vorrang, Assoziativität und Klammerung:

a = ( b + ~ c ++ + d ) \* (double) e + f [i]

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

14 ( 4 2 1 ) 4 ) 3 2 4 1

- eindeutig darstellbar als **Auswertungsbaum**:



## ANSI-C Ausdrücke: Auswertungs-Reihenfolge (2)

- Die **Auswertungs-Reihenfolge der Operanden** eines Operators ist in der Regel Compiler-abhängig.

Man kann insbesondere nicht erwarten, dass ein Ausdruck von links nach rechts abgearbeitet wird.

Beispiel: `i = 0, v[i] = ++i`

Der Compiler darf Code erzeugen, der `v[i]` vor oder nach `++i` auswertet. Je nachdem wird `v[0]` oder `v[1]` auf 1 gesetzt!

- nur bei den folgenden vier Operatoren ist verbindlich festgelegt, dass der linke Operand vor dem rechten ausgewertet wird:

Komma `,`

Bedingung `?:`

Logisches Und `&&`

Logisches Oder `||`

Bei den Operatoren `&&` bzw. `||` wird der rechte Operand gar nicht ausgewertet, wenn der linke 0 bzw. ungleich 0 ist (Lazy Evaluation)

## ANSI-C Ausdrücke: Makros

---

Der ANSI-C-Präprozessor erlaubt es, häufig benötigte Ausdrücke als **Makro** zu definieren.

- Definition eines Makros:

```
#define Name( Parameterliste ) Ausdruck
```

*die Parameterliste besteht nur aus durch Komma getrennten Namen ohne Typen*

- Benutzung eines Makros:

nach der Definition kann ein Makro wie eine Funktion "aufgerufen" werden

*das Makro wird beim Übersetzen vom Präprozessor expandiert, d.h. durch den Ausdruck mit eingesetzten Argumenten ersetzt*

- Beispiel:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

```
m = 2 * max(x + y, z); /* Benutzung des Makros */
```

```
m = 2 * ((x + y) > (z) ? (x + y) : (z)); /* expandiertes Makro */
```

*Achtung: unbedingt den ganzen Ausdruck und jeden Parameter klammern!*

## ANSI-C Ausdrücke: Vergleich mit Java

---

Bei den Ausdrücken gibt es wichtige Unterschiede zwischen ANSI-C und Java:

- in ANSI-C gibt es im Gegensatz zu Java mehrdeutige Ausdrücke
- in ANSI-C gibt es keine echten logischen Ausdrücke

*dadurch Verwechslungsgefahr zwischen Arithmetik und Logik, (speziell zwischen Zuweisung = und Test auf Gleichheit == sowie zwischen logischem Und/Oder und bitweisem Und/Oder)*

- in Java gibt es keine Makros

*dafür expandiert die Java Virtuelle Maschine zur Laufzeit automatisch häufig durchlaufene Aufrufe einfacher Methoden wie ein Makro*

# ANSI-C Ablaufsteuerung: Verzweigung

Eine **Verzweigung** ermöglicht optionale und alternative Anweisungen.

- Syntax:

```
if (Bedingung) Anweisung /* falls Bedingung erfüllt */
```

```
if (Bedingung)
    Anweisung1 /* falls Bedingung erfüllt */
else
    Anweisung2 /* falls Bedingung nicht erfüllt */
```

```
if (Bedingung1)
    Anweisung1 /* falls Bedingung1 erfüllt */
else if (Bedingung2)
    Anweisung2 /* falls nur Bedingung2 erfüllt */
else
    Anweisung3 /* falls keine Bedingung erfüllt */
```

Eine *Bedingung* ist ein Ausdruck mit arithmetischem Typ oder Zeigertyp.

Vorsicht bei geschachtelten Verzweigungen:

Ein `else`-Teil gehört immer zum letzten noch offenen `if`.

Eine andere Zuordnung muss mit geschweiften Klammern erzwungen werden:

```
if (Bedingung1) {
    if (Bedingung2) ...
} else {
    ...
}
```

## Beispielprogramm Verzweigung

```
#include <stdio.h>
int main(void)
{
    int m, n;
    printf("Zwei Zahlen eingeben: ");
    if (scanf("%d%d", &m, &n) < 2)
    {
        fprintf(stderr, "Eingabefehler !\n");
    }
    else if (m > n)
    {
        printf("Maximum: %d\n", m);
    }
    else
    {
        printf("Maximum: %d\n", n);
    }
    return 0;
}
```

Liest zwei ganze Zahlen ein und gibt deren Maximum aus.

# ANSI-C Ablaufsteuerung: Fallunterscheidung

Die **Fallunterscheidung** ist eine spezielle Schreibweise für Mehrfachverzweigungen.

- Syntax: 

```
switch (Ausdruck) {  
    case Wert1:  
        Anweisung1  
        break;  
    case Wert2:  
        Anweisung2  
        break;  
    default:  
        Anweisung3  
}
```

Im Prinzip gleichbedeutend mit:

```
if (Ausdruck == Wert1)  
    Anweisung1  
else if (Ausdruck == Wert2)  
    Anweisung2  
else  
    Anweisung3
```

Der *Ausdruck* muss einen ganzzahligen Typ haben.

Die **case**-Werte müssen dazu passende ganzzahlige Konstanten (bzw. **enum**-Werte) sein.

Der **default**-Fall wird ausgeführt, wenn der Ausdruck keinen der **case**-Werte hat.

Mit **break** wird die Fallunterscheidung verlassen. *Ohne **break** z.B. hinter Anweisung1 würde nach Anweisung1 die Anweisung2 ausgeführt*

## Beispielprogramm Fallunterscheidung (1)

```
#include <stdio.h>  
int main(void)  
{  
    int month;  
    printf("Monat eingeben [1-12]: ");  
    if (scanf("%d", &month) < 1)  
    {  
        month = 0;  
    }  
    switch (month)  
    {  
        case 2:  
            printf("28 oder 29 Tage\n");  
            break;  
        case 4:  
        case 6:  
        case 9:  
        case 11:  
            printf("30 Tage\n");  
            break;  
    }
```

*Gibt die Anzahl der Tage  
eines Monats aus.*



## Beispielprogramm Fallunterscheidung (2)

*/\* Fortsetzung ... \*/*

```
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
    printf("31 Tage\n");
    break;
default:
    fprintf(stderr, "Eingabefehler!\n");
}
return 0;
}
```

## ANSI-C Ablaufsteuerung: Schleifen (1)

Eine **Schleife** ermöglicht die wiederholte Ausführung einer Anweisung.

- Syntax der **while**-Schleife:

```
while (Bedingung)
    Anweisung
```

*Wiederholt die Anweisung, solange die Bedingung gilt.*

- Syntax der **do**-Schleife:

```
do
    Anweisung
while (Bedingung);
```

*Führt die Anweisung aus und wiederholt sie dann, solange die Bedingung gilt.*

Gleichbedeutend mit:

```
{
    Anweisung
    while (Bedingung)
        Anweisung
}
```

Eine *Bedingung* ist wie gehabt ein Ausdruck mit arithmetischem Typ oder Zeigertyp.

## ANSI-C Ablaufsteuerung: Schleifen (2)

Die **for**-Schleife ist eine spezielle Schreibweise für Schleifen mit Laufvariablen.

*Die Laufvariable muss vor der Schleife definiert werden.*

*for-Schleifen werden häufig benutzt, um Felder oder Listen (allgemein: Aggregate) abzulaufen. Dabei werden die aggregierten Elemente über eine Laufvariable angesprochen.*

- Syntax der allgemeinen **for**-Schleife:

```
for (Initialisierung; Bedingung; Fortschaltung)
    Anweisung
```

Die *Initialisierung* ist ein Ausdruck, der die Laufvariable auf das erste Element des Aggregats setzt.

Die *Fortschaltung* ist ein Ausdruck, der die Laufvariable auf das nächst folgende Element des Aggregats setzt.

Die *Bedingung* prüft, ob alle Elemente besucht wurden.

Gleichbedeutend mit:

```
{
    Initialisierung;
    while (Bedingung)
    {
        Anweisung
        Fortschaltung;
    }
}
```

## Beispielprogramm while-Schleife

```
#include <stdio.h>

int main(void)
{
    int sum = 0;
    int n;

    printf("Zahlen eingeben (Ende mit Strg-d): ");
    while (scanf("%d", &n) == 1)
    {
        sum += n;
    }

    printf("Summe: %d\n", sum);
    return 0;
}
```

*Liest ganze Zahlen ein und gibt deren Summe aus.*

## Beispielprogramm do-Schleife

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

```
int main(void)
```

```
{
```

```
    int n = 0;
```

```
    /* Dezimalzahl einlesen */
```

```
    do
```

```
    {
```

```
        printf( "Zahl zwischen 0 und 255 eingeben: " );
```

```
    }
```

```
    while (scanf( "%d", &n) == 1  
           && (n < 0 || n > 255));
```

```
    ...
```

Liest eine ganze Zahl ein und  
gibt sie in Binärdarstellung aus.

```
    ...
```

```
    /* Binaerzahl ausgeben */
```

```
    printf( "          "); /* 7 Leerzeichen */
```

```
    do
```

```
    {
```

```
        printf( "%d\b\b", n % 2);
```

```
        n /= 2;
```

```
    }
```

```
    while (n > 0);
```

```
    printf( "\n");
```

```
    return 0;
```

```
}
```

## Beispielprogramm for-Schleife

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

```
int main(void)
```

```
{
```

```
    int an_array[] = {3082, 3101, 3275, 3436};
```

```
    const int array_size = (int)(sizeof an_array / sizeof *an_array);
```

```
    int i; /* Laufvariable */
```

```
    for (i = 0; i < array_size; ++i)
```

```
    {
```

```
        printf( "%d\n", an_array[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

Gibt Feldelemente aus.

## ANSI-C Ablaufsteuerung: Sprünge (1)

- Eine **break**-Anweisung springt hinter die umgebende Fallunterscheidung / Schleife:

```
while (...)
{
    ...
    if (Bedingung) break;
    ...
}
... /* break springt hier hin */
```

Gleichbedeutend mit:

```
int stop = 0;
while (... && !stop)
{
    ...
    if (Bedingung)
        stop = 1;
    else
        ...
}
```

- Eine **continue**-Anweisung springt zum nächsten Schleifen-Durchlauf:

```
while (...)
{
    if (Bedingung) continue;
    ...
}
```

Gleichbedeutend mit:

```
while (...)
{
    if (!Bedingung)
    {
        ...
    }
}
```

## ANSI-C Ablaufsteuerung: Sprünge (2)

- Eine **goto**-Anweisung springt zu einer markierten Anweisung:

```
for (...) {
    for (...) {
        if (Bedingung) goto ende;
        ...
    }
}
ende:
...
```

*verlässt in einem Schritt die beiden ineinander geschachtelten Schleifen*

*goto-Anweisung sollten vermieden werden*

*den obigen Schleifenabbruch kann man ohne goto lösen, indem man die Schleifen in eine Funktion verlegt und diese per return verlässt*

## ANSI-C Ablaufsteuerung: Sprünge (3)

Eine **return**-Anweisung springt an die Aufrufstelle einer Funktion zurück.

*Genauerer später bei den Funktionen.*

- Innerhalb von **main** beendet **return** das Programm:

```
int main(void) {  
    ...  
    if (Bedingung) return 1;  
    ...  
    return 0;  
}
```

*Ein Rückgabewert 0 gilt als normales Programmende,  
ein Rückgabewert ungleich 0 gilt als Fehlerabbruch*

## Beispielprogramm Sprünge (1)

```
#include <stdio.h>  
#include <ctype.h>  
  
int main(void)  
{  
    int sum = 0;  
    int n;  
  
    printf("Zahlen eingeben (Ende mit Strg-D): ");  
    while (1) /* Endlos-Schleife, alternativ auch for (;;) */  
    {  
        int i = scanf("%d", &n);  
  
        if (i == EOF) /* Strg-D ? */  
        {  
            fprintf(stderr, "*** Eingabeende\n");  
            break; /* hinter die Schleife springen */  
        }  
  
        ...  
    }  
}
```

*Liest ganze Zahlen ein und  
gibt deren Summe aus.*

## Beispielprogramm Sprünge (2)

```
...
else if (i == 0)
{
    char c;
    fprintf(stderr, "*** Eingabe ist keine ganze Zahl: ");
    while ((c = getchar()) != EOF && !isspace(c))
    {
        putc(c, stderr);
    }
    putc('\n', stderr);
    continue; /* zum naechsten Schleifendurchlauf springen */
}
sum += n;
}
printf("Summe: %d\n", sum);
return 0; /* normales Programmende */
}
```

## ANSI-C Anweisungen: Empfehlungen (1)

- Leerzeichen machen Ausdrücke lesbarer, unnötige Klammern nicht unbedingt:

a + b * c	<del>a+(b*c)</del>	<i>/* Klammern unnötig */</i>
(a + b) * c	<del>(a+b)*c</del>	<i>/* Klammern notwendig */</i>

- Ausdrücke mit Seiteneffekten vermeiden:

<del>a = b + c++;</del>	<i>/* Seiteneffekt auf c */</i>
a = b + c;	<i>/* Aufteilung meistens besser */</i>
++c;	

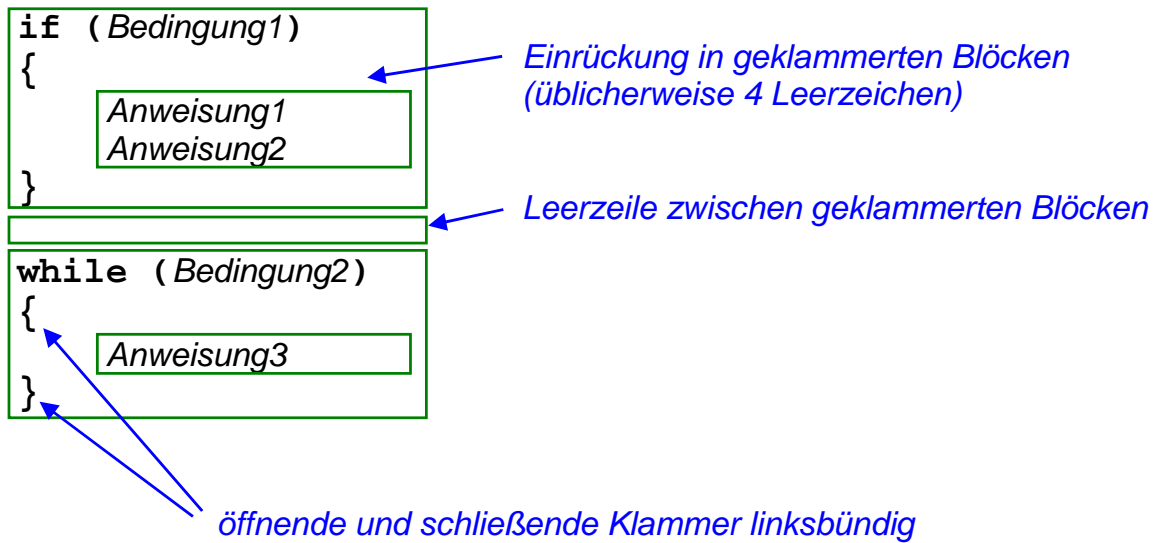
- Nur eine Anweisung pro Zeile schreiben, Kontrollstrukturen mehrzeilig schreiben.

<pre>if (Bedingung) {     Anweisung; }</pre>	<del><pre>if (Bedingung) Anweisung;</pre></del>
--	---

*Vereinfacht erheblich die Fehlersuche und Qualitätssicherung mit Werkzeugen wie Compiler, Debugger usw.*

## ANSI-C Anweisungen: Empfehlungen (2)

- Durch Zwischenraum (*Whitespace*), Klammerung und Einrückung die **Blockstruktur** der Ablaufsteuerung verdeutlichen:



## ANSI-C Anweisungen: Vergleich mit Java

Bei den Anweisungen gibt es einige Unterschiede zwischen ANSI-C und Java:

- in ANSI-C Variablen-Definitionen nur am Anfang eines Blocks
- in ANSI-C keine Ausnahmebehandlung mit `try/catch/throw`
- in ANSI-C keine vereinfachte `for (T element : alleElemente)`-Schleife  
*außerdem kann die Laufvariable einer `for`-Schleife nicht erst im Schleifenkopf, sondern muss vor der Schleife definiert werden*
- in ANSI-C gibt es die Sprünge `break` und `continue` nur ohne Marke  
*Marken können nur mit `goto` angesprungen werden, was aber vermieden werden sollte*

# ANSI-C Anweisungen: Index

---

Anweisung 3-1,3-27 bis 3-29  
Auswertungsreihenfolge 3-8,3-9  
**break** 3-1,3-14,3-22  
**case** 3-1,3-14  
**default** 3-1,3-14  
**continue** 3-1,3-22  
**do** 3-1,3-17  
**else** 3-1,3-12  
**for** 3-1,3-18  
**goto** 3-1,3-23  
**if** 3-1,3-12  
Kommaoperator 3-5  
logischer Operator 3-6  
Makro 3-10  
Statement 3-1  
**return** 3-1,3-24  
**switch** 3-1  
Vergleichsoperator 3-6  
**while** 3-1,3-17