Systemprogrammierung

Teil 3: C Anweisungen

Ausdrücke / Operatoren / Ablaufsteuerung

Prof. Dr. H. Drachenfels
Hochschule Konstanz

Version 10.0
21.6.2024

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`

C Operatoren: Übersicht (1)

- Operatoren mit einem Operanden:

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

1 *Sortierung vom höchsten Vorrang 1 bis niedrigstem Vorrang 15.*

2 *Einstellige Operatoren haben keine Assoziativität. Sie werden von innen nach außen berechnet.*

3 *In ( ) oder, [ ] geklammerte Parameter der Operatoren bleiben bei der Stelligkeit unberücksichtigt.*

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	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

C Operatoren: Übersicht (3)

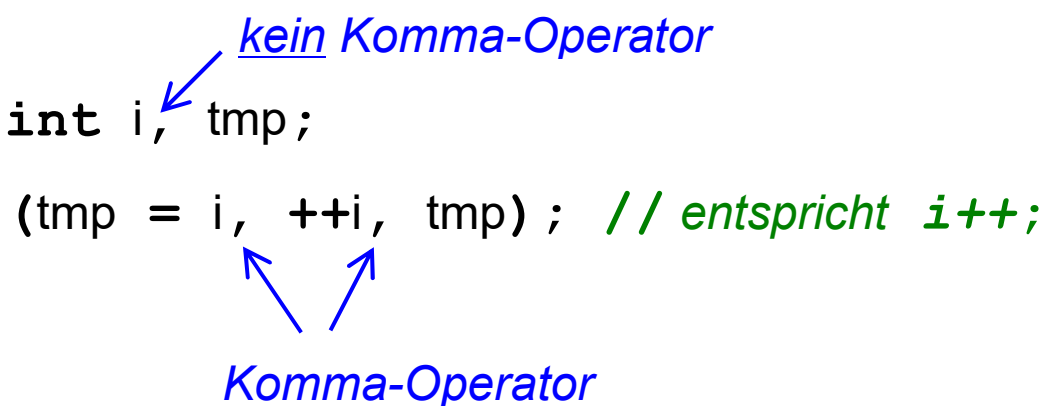
- weitere Operatoren mit zwei Operanden, bzw. in einem Fall mit drei Operanden:

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

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 !



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!*

C Operatoren: Vergleich mit Java

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

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

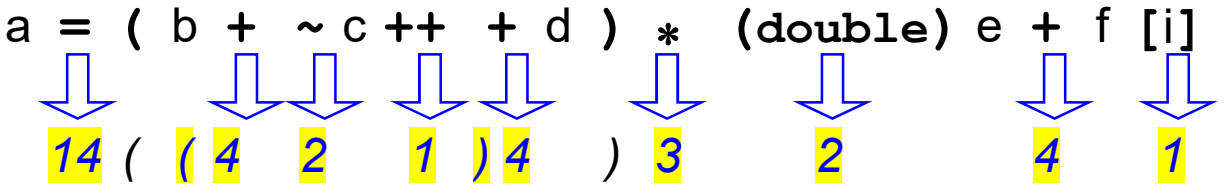
  In Java haben die Operatoren `&` und `|` je nach Operandentyp eine andere Bedeutung:

  im Fall von `boolean` sind es logische Operatoren mit voller Auswertung beider Operanden

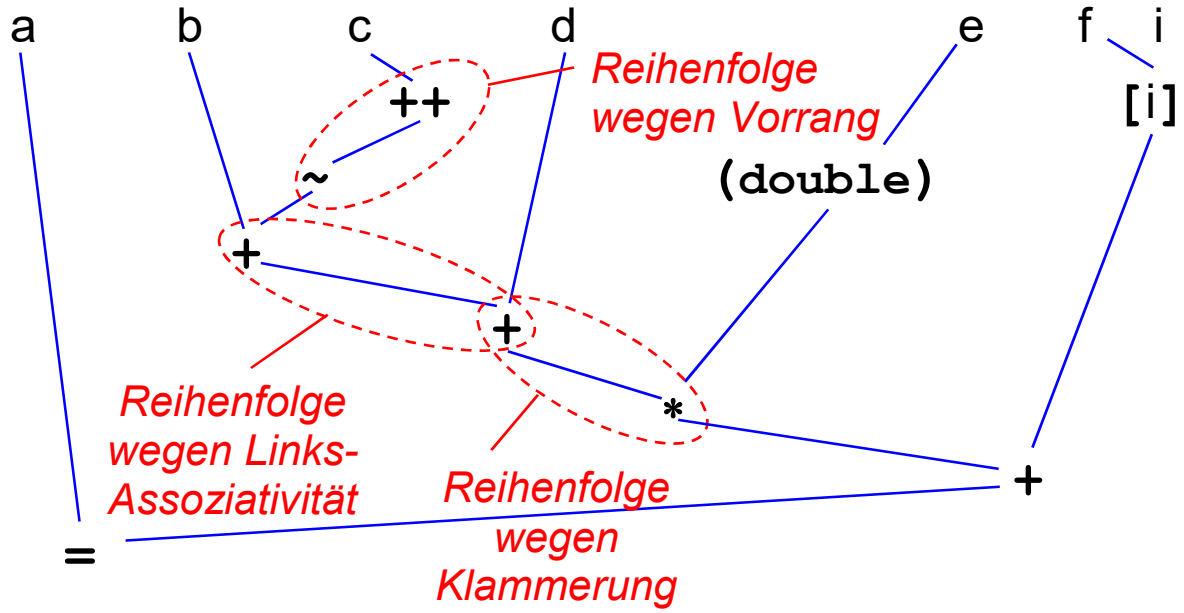
  im Fall von ganzen Zahlen sind es wie bei C arithmetische Bitoperatoren

C Ausdrücke: Auswertungs-Reihenfolge (1)

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



- eindeutig darstellbar als **Auswertungsbaum**:



Reihenfolge in jedem Ast von oben nach unten

Reihenfolge zwischen den Ästen in der Regel beliebig

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)

C Ausdrücke: Makros

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

- Definition eines Makros:

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

  die Parameterliste besteht 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(7, 8 | 9); // Verwendung des Makros
m = 2 * ((7) > (8 | 9) ? (7) : (8 | 9)); // expandiertes Makro
```

Achtung: unbedingt den ganzen Ausdruck und jeden Parameter klammern!

C Ausdrücke: Vergleich mit Java

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

- in C gibt es im Gegensatz zu Java mehrdeutige Ausdrücke
- in 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

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

Liest zwei ganze Zahlen ein und gibt deren Maximum aus.

```
#include <stdio.h>
int main(void)
{
    printf("Zwei Zahlen eingeben: ");
    int m, n;
    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;
}
```

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)

Gibt die Anzahl der Tage eines Monats aus.

```
#include <stdio.h>
int main(void)
{
     printf("Monat eingeben [1-12]: ");
     int month;
     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;
```

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;
}
```

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.

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 eine 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

Liest ganze Zahlen ein und gibt deren Summe aus.

```
#include <stdio.h>

int main(void)
{
    printf("Zahlen eingeben (Ende mit Strg-d): ");
    int sum = 0;
    int n;
    while (scanf("%d", &n) == 1)
    {
        sum += n;
    }
    printf("Summe: %d\n", sum);
    return 0;
}
```

Beispielprogramm do-Schleife

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

```
#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));

   ...
```

```
   ...

   // Binaerzahl ausgeben
   printf("       ");  // 7 Leerzeichen
   do
   {
      printf("%d\b\b", n % 2);
      n /= 2;
   }
   while (n > 0);

   printf("\n");

   return 0;
}
```

Beispielprogramm for-Schleife

Gibt Feldelemente aus.

```
#include <stdio.h>

int main(void)
{
    int an_array[] = {3082, 3101, 3275, 3436};
    const int array_size = (int) (sizeof an_array / sizeof *an_array);

    for (int i = 0; i < array_size; ++i)
    {
        printf("%d\n", an_array[i]);
    }

    return 0;
}
```

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)
    {
        ...
    }
}
```

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

C Ablaufsteuerung: Sprünge (3)

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

Genaueres 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)

Liest ganze Zahlen ein und gibt deren Summe aus.

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

int main(void)
{
    printf("Zahlen eingeben (Ende mit Strg-D): ");
    int sum = 0;
    while (1) // Endlos-Schleife, alternativ auch for (;;)
    {
        int n;
        int i = scanf("%d", &n);

        if (i == EOF) // Strg-D ?
        {
            fprintf(stderr, "*** Eingabeende\n");
            break; // hinter die Schleife springen
        }

        ...
```

Beispielprogramm Sprünge (2)

```
        ...
        else if (i == 0)
        {
            fprintf(stderr, "*** Eingabe ist keine ganze Zahl: ");
            int c;
            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
}
```

C Anweisungen: Empfehlungen (1)

- Leerzeichen machen Ausdrücke lesbarer, unnötige Klammern nicht unbedingt:

```
a + b * c          a+(b*c)        // Klammern unnötig
(a + b) * c        (a+b)*c        // Klammern notwendig
```

- Ausdrücke mit Seiteneffekten vermeiden:

```
a = b + c++;       // Seiteneffekt auf c

a = b + c;         // Aufteilung meistens besser
++c;
```

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

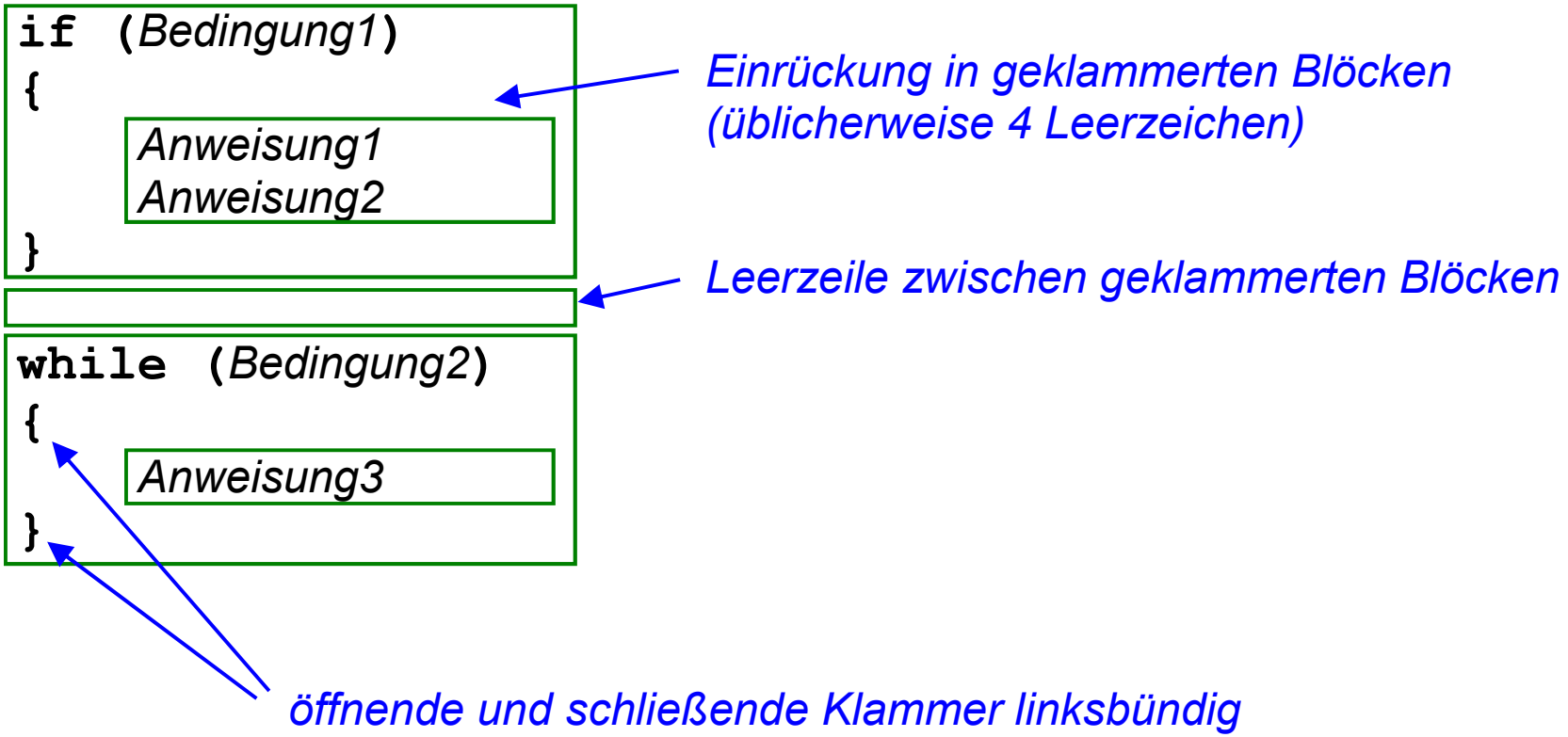
```
if (Bedingung)
{
    Anweisung;
}
```

```
if (Bedingung) Anweisung;
```

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

C Anweisungen: Empfehlungen (2)

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



C Anweisungen: Vergleich mit Java

Bei den Anweisungen gibt nur wenige Unterschiede zwischen C und Java:

- in C keine Ausnahmebehandlung mit **`try`** / **`catch`** / **`throw`**
- in C keine vereinfachte **`for`**`(`*T element* `:` *alleElemente*`)`-Schleife
- in C gibt es die Sprünge **`break`** und **`continue`** nur ohne Marke

  Marken können nur mit **`goto`** *angesprungen werden, was aber vermieden werden sollte*

C Anweisungen: Index